1975 SAGAMORE COMPUTER CONFERENCE

PROCEEDINGS

OF THE

1975 SAGAMORE COMPUTER CONFERENCE

ON

PARALLEL PROCESSING

SYRACUSE UNIVERSITY · FOUNDED A·D·1870 · SUOS CULTORES SCIENTIA CORONAT

Syracuse University

PROCEEDINGS
OF THE
1975 SAGAMORE COMPUTER CONFERENCE
ON
PARALLEL PROCESSING

Papers presented on
August 19-22, 1975

Department of Electrical & Computer Engineering

SYRACUSE UNIVERSITY

PREFACE

The Sagamore Computer Conference has been held annually for the past four years at the former Vanderbilt summer estate in the Central Adirondack Mountains. The Conference was originally conceived to provide a secluded environment, a 1300-acre preserve surrounding the private Sagamore Lake, for the participants with excellent opportunities for exchanging ideas and learning each other's research activities. Thus informative discussions may be made not only during the technical sessions but also throughout the various sports and social gatherings provided by the Conference.

The first Sagamore Computer Conference was held on August 23-25, 1972. The subject of that conference was on "RADCAP (Rome Air Development Center Associative Processor) and Its Applications". About 90 invited participants attended the 2-day conference to hear the 17 technical papers presented. The Conference Proceedings were published and distributed by RADC and Syracuse University, co-sponsors of the Conference. In 1973, the Conference broadened its scope to "Parallel Processing" and issued the Call-for-Papers announcement. Among the submitted papers, 34 were accepted and presented on August 22-24, 1973. The 1973 Conference was sponsored by Syracuse University in cooperation with IEEE and ACM. Copies of its Proceedings (IEEE Catalog Number 73 CH0812-8 C) may be available from any one of these institutions. The 1974 Conference was extended to three days (August 20-23, 1974) to provide the participants with more time for individual activities. In addition to a panel discussion, the participants heard the presentation of 35 technical papers. The Conference was again sponsored by Syracuse University, but the Proceedings were published and distributed by Springer-Verlag as Volume 24 in their series of Lecture Notes in Computer Science.

Since 1972, the number of submitted papers and, in particular, the number of requests to attend the Conference have increased significantly and persistently. In 1973 and 1974, the Minnowbrook Conference Center was opened to accommodate the overflow from Sagamore. But this year, even with this added facility, we had to turn down many requests to attend the Conference. Thus, the Conference had grown out of Sagamore (as well as Minnowbrook) accommodations, and, much to our regret, this year's Conference had to be the last one held at Sagamore.

The success of the Sagamore Computer Conferene is mostly due to the vigorous support of many individuals. For this year's conference we are grateful to Col. Robert D. Krutz of Rome Air Development Center for his enlightening Keynote Address. Also, at the request of many past Sagamore participants, this year we invited Professors Enslow, Kuck, Ramamoorthy, and Yau to write survey papers on various computer architectures related to Parallel Processing. Their time and effort in this respect are very much appreciated. We appreciate the efforts of the authors who submitted papers for consideration. We are also much indebted to all the reviewers who, in order to meet the stringent review deadlines, put aside their own busy work schedule to carefully evaluate the papers sent for their comments. The success of the Conference is also attributable to the generous help we received from the session chairpersons. In addition, a special acknowledgement is due to Elliott McCulley, Anne Woods, Mary Jo Fairbanks, and Angela Wisniewski for their administrative assistance and typing help.

Tse-yun Feng
Department of Electrical & Computer Engineering
Syracuse University

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT'D.)

TABLE OF CONTENTS (Cont'd.)

TABLE OF CONTENTS (CONT'D.)

ASSOCIATIVE PROCESSOR ARCHITECTURE - A SURVEY

S. S. Yau and H. S. Fung
Departments of Computer Sciences and Electrical Engineering
Northwestern University
Evanston, Illinois 60201 U.S.A.

(Invited Paper)

Abstract -- A survey of associative processor architecture is presented. Based on their architecture, associative processors are classified into four categories, namely fully parallel, bit-serial, word-serial and block-oriented. The fully parallel associative processors are divided into two classes, word-organized and distributed logic. The architecture of each of these categories is described.

Introduction

An associative processor can generally be described as a processor which has the following two properties: (1) Stored data items can be retrieved using their content or part of their content (instead of their addresses) and (2) data transformation operations, both arithmetic and logic operations, can be performed over many sets of arguments with a single instruction. Because of these parallel processing characteristics, associative processors have a much faster data processing rate than conventional sequential computers, and hence are more effective in handling many types of information processing problems, such as information storage and retrieval of rapidly changing data bases, fast search of a large data base, arithmetic and logic operations on large sets of data, control and executive functions in large-scale computer systems, radar signal processing and weather computations. However, because of their relatively high implementation cost, associative processors are usually used in conjunction with standard sequential computer systems so that many required high-speed parallel processing tasks which cannot be effectively executed by sequential processors are performed by associative processors. It is anticipated that associative processors will be used more extensively in the future for enhancing the performance of many special-purpose and general-purpose computer systems.

In this paper, we will present a survey of the architecture of various associative processors. We will first give a general description of associative processor architecture and then classify associative processors on the basis of their architecture. Then we will discuss each of the categories of associative processors in some detail.

General Description

In general, the architecture of an associative processor can be described as shown in Fig. 1, which consists of an associative memory, arithmetic and logic unit (ALU), control system, instruction memory, and an input/output interface. The major difference between an associative processor as shown in Fig. 1 and a standard sequential processor is the use of an associative memory instead of a location-addressed memory. Because of this difference, all the other blocks are also different between an associative processor and a standard sequential processor. Furthermore, the associative memory has a major impact on the architecture of an associative processor, and the associative processor architecture can be classified based on the organization of its associative memory.

An associative memory [1] - [8] can be defined as a memory system with the property that stored data items can be retrieved by their content or part of their content (that is, the first property of an associative processor). An associative memory is also called catalog memory [9], content-addressed memory [10], data-addressed memory [11], parallel search memory [12], search memory [13] - [15], search associative memory [16], content-addressable memory [17], distributed logic memory [18], associative pushdown memory [19], and multi-access associative memory [20].

From the hardware point of view, in order to retrieve stored data items by their content or part of their content, it is required that the memory words can be accessed by matching their content or part of their content with the given search key words, instead of by an address as in a location-addressed memory. The basic memory element of the associative memory is called the bit-cell, which has the property that one-bit information can be written in, read out, and compared with interrogating information. The search operations, which consist of masking and comparison, are executed in a fashion that depends on the organization of the associative memory. The search key word can be compared with all the words of the memory through the interrogating bit drives and comparison logic circuitry. The possibility of matching multiple words to a search key word requires the associative memory to have some method of tagging all the matched words. The tag function and matched-word indication are performed by the so-called word-match tag networks. Those matched words can be accessed with a single instruction [4], [6], [7], [21]. It is noted that a memory can have the associative property by performing either parallel comparison (word-parallel or word-serial) or serial comparison (bit-serial).

The first associative memory was developed by Slade and McMahon [9] in 1956 using cryotrons. Since then, associative memories have been implemented using tunnel diodes [22], [23], evaporated organic diode arrays [2], magnetic cores [22], [24] - [27], plated wire [22], [28], thin permalloy film on copper wire [22], semiconductors [29], transfluxors [30], biax cores [31], laminated ferrites [32], magnetic films [33], solenoid arrays [34], bicore thin film sandwiches [22], multi-aperture logic elements [35], and large-scale integrated (LSI) circuits [36]. The capacity of these associative memories is limited by factors such as half-select noise which limits the word length and interrogation drive problems which limit the number of words. Because of these limitations and because of high implementation cost, most associative memories in early years had small capacity, say up to 1K words with length up to 100 bits [26]. Recent associative memories have become larger and more flexible due to the development of new architectural concepts and the use of LSI technology. For example, in PEPE (parallel element processing ensemble) [37] - [42] there are a number of processing elements each of which contains a simple 1K $\times$ 32-bit random-access memory, called the element memory, which is shared on a cycle-stealing basis by the arithmetic unit, correlation unit and associative output unit in the processing element to perform associative processing. In each associative array module of STARAN [43] - [46], a so-called multi-dimensional access memory - implemented by a 256 $\times$ 256-bit random-access memory is used to accommodate both bit-slice accesses for associative processing and word-slice accesses for input/output.

An associative memory may perform the following comparison operations:

equal	not equal
less than	greater than
less than or equal	greater than or equal
maximum value	minimum value
between limits	not between limits
next higher	next lower

An associative processor normally performs other complicated data transformation operations. For instance, the matched words in the associative memory are retrieved serially to the ALU through the output circuit of the associative memory under the control of the control system. The ALU performs the specified data transformation operations and the result is then stored to the associative memory, if necessary.

The first associative processor was developed by Behnke and Rosenberger [47] in 1963 using cryotrons. Since then a number of laboratory models of associative processors have been built using various types of associative memories. However, associative processors have not been put to practical use until the development of PEPE (parallel element processing ensemble) [37] - [42] and STARAN [43]-[46].

The architecture of associative processors can generally be classified into four categories according to the comparison process of their associative memories. The four categories are fully parallel, bit-serial, word-serial, and block-oriented. These are two types of fully parallel associative processors: word-organized and distributed logic. The former has its comparison logic associated with each bit-cell of every word, and the latter has its comparison logic associated with each character-cell (for a fixed number of bits) or a group of character-cells. In a bit-serial associative processor, only one bit-column (bit-slice) of all the words is operated at a time. A word-serial associative processor essentially represents hardware implementation of a simple program loop for search. The important factor which contributes to the relative efficiency of this approach as compared with programmed search in a standard sequential processor is that the instruction decoding time is greatly reduced since the search operation requires only a single instruction in the word-serial associative processor. A block-oriented associative processor can be implemented by using a logic-per-track rotating memory which consists of a head-per-track disk with some logic associated with each track.

In the following sections, we will discuss each of these categories in more detail.

Fully Parallel Associative Processors

Fully Parallel Word-Organized Associative Processors

As mentioned before, the major characteristic of a fully parallel word-organized associative processor is that the comparison logic is associated with each bit-cell of every word of its associative memory. Thus, its comparison process is performed in a parallel-by-word and parallel-by-bit fashion. The general organization of the associative memory with the ALU in such an associative processor is shown in Fig. 2. In this figure each cross-point represents a bit-cell. Although the operations of a fully parallel word-organized associative processor are simplest and fastest compared to other types of associative processors, its hardware is also the most complicated because each bit-cell has to contain the comparison logic. Because of its hardware complexity, this type of associative processor was developed only during the early stages although many experimental models were developed and their fully parallel word-organized associative memory systems used cryogenic components [9], [11], [31], [48] - [53], magnetic cores [24], [25] and cut-point cellular logic [54].

Distributed Logic Associative Processors

A distributed logic associative processor is a fully parallel character-oriented associative processor whose memory (usually called distributed logic memory [18]) has its comparison logic associated with each character-cell or each group of character-cells. A number of distributed logic associative processors have been developed [37] - [42], [55] - [62]. The first associative processor of this type was proposed by Lee [55] and a number of its variations were presented later [56] - [61]. The most well-known associative processor system of this type so far developed is the PEPE [37] - [42] by Bell Laboratories for the U.S. Army Advanced Ballistic Missile Defense Agency. In this section, these associative processors will be briefly described.

Lee's Distributed Logic Associative Processor and Its Modifications. The distributed logic associative processor proposed by Lee [55] can be represented by the block diagram shown in Fig. 3. Each character-cell has a single cell state element (state part) S which may either be in an active state or in a quiescent state, and each character-cell also has a number of cell symbol elements (symbol part) $X_1, \ldots, X_n$ depending upon the size of the symbol alphabet. The cell state element or cell symbol element is a bistable device, such as a flip-flop. Each character-cell stores one character symbol of information and can communicate with its two neighboring character-cells as well as the control system. A string of information is therefore stored in a corresponding string of character-cells.

Each data block consists of a name string and an arbitrary number of parameter strings. Every name string is preceded by a tag α, and every parameter string is preceded by a tag β. When the input search key is a name string, the fully parallel distributed logic memory is expected to output all of the parameter strings associated with the name string. This is the so-called direct retrieval. On the other hand, when the input search key is a parameter string, the fully parallel distributed logic memory is expected to output all of the name strings associated with that parameter string. This is so called cross-retrieval. In order to perform direct retrieval and cross retrieval, each character-cell in the fully parallel distributed logic memory must have enough cell logic circuitry so that it can produce a "yes" or "no" answer to a simple question such as whether the symbol of the character-cell is A or not A. If we want to retrieve all of the parameter strings whose name is AB, we will ask each character-cell whether its character symbol is A. If a cell gives us an answer "yes," we also want each character-cell to have enough cell logic circuitry so that it can signal its next character-cell to be ready to determine whether the symbol of the character-cell is B. The character-cells which finally respond "yes" to the name string AB are now ready to signal all those character-cells storing all the parameter strings associated with the name string AB to output their contents.

Typical operations of a character-cell are changing state, transmitting state information to a neighboring character-cell, accepting data from the input bus, or putting its character symbol on the output bus. When a character-cell is in an active state and when the input signal lead is activated, the symbol which is carried on the input bus is then stored in that character-cell. When a character-cell is in an active state, an output signal causes that character-cell to read out its symbol through the output bus and store its symbol in the output symbol buffer. Comparison operation is controlled by the match signal through the comparison logic of each character-cell. The stored symbol of each character-cell is compared with the symbol carried on the input bus, and a signal from each matched character-cell is transmitted to one of its neighboring character-cells which then becomes active. The directions of transmission of the signals are controlled by the signal on the direction leads. All the character-cells evaluate and act according to the input conditions (given by the input and state buses) independently and simultaneously.

Lee's system has been experimented using cryogenics consisting of 72, 8-bit character-cells [62].

Several modifications of Lee's original system have been proposed. Lee and Paull [18] proposed a distributed logic memory using two cell state elements instead of one for each character-cell, more control bus leads and a threshold circuit. They defined the complex symbol of a character-cell which includes both the two cell state elements and the cell symbol elements of the character-cell. The matching process requires that an entire complex symbol be used for matching. They presented a more complicated design for a character-cell memory and an external control unit in order to have more capabilities to deal with problems such as cross retrieval, erasing, gap closing and preference which appear in information retrieval.

In order to overcome the propagation timing problems, Gaines and Lee [59] proposed to redesign the logic circuitry using two different-purpose cell state elements, called the match flip-flop and control flip-flop, and adding a mark line to simultaneously activate all cells to the right of each active cell up to the first cell whose control flip-flop is set. Due to the control of the propagation of the marking signal, this memory system is capable of performing two new simultaneous operations, shifting and marking strings.

Crane and Githens [60] extended Lee's

system to a two-dimensional distributed logic memory which could be used to perform highly parallel arithmetic operations through the use of a large number of identical processing units on many sets of data simultaneously while retaining content-addressing capability to these data sets. Such an extension can be illustrated by the block diagram shown in Fig. 4.

Parallel Element Processing Ensemble (PEPE). PEPE [37] - [41] is one of the two large-scale associative processors developed to date. Its basic concept was derived from Lee's distributed logic associative processor and was originally developed by Bell Laboratories for the U.S. Army Advanced Ballistic Missile Defense Agency [37] - [39]. A second model of PEPE with both architectural and circuit technology improvements is being developed by the Agency [40] - [42]. The description of PEPE presented in this section is primarily for the current model.

PEPE is composed of the following functional subsystems: an output data control, an element memory control, an arithmetic control unit, a correlation control unit, an associative output control unit, a control system, and a number of processing elements. Each processing element consists of an arithmetic unit, a correlation unit, an associative output unit and a 1024×32 bit element memory. In addition, there are primary power and signal distribution subsystems to convert and route power and control and data signals between various functional subsystems. It is noted that the number of processing elements used in the PEPE is variable and may be increased or decreased to meet the requirements of the application. This variability has no impact on PEPE system performance, except that enough processing elements must be available to accommodate the expected number of objects to be tracked. A PEPE with 288 processing elements organized in 8 element bays was presented in [40].

The block diagrams of PEPE and its processing elements are shown in Figs. 5 and 6, respectively. The processing elements are the main computational component of PEPE. Selected portions of the data processing load are loaded from the host computer (a CDC 7600) to the processing elements. The loading selection process is determined by the inherent parallelism of the task and the ability of PEPE's unique architecture to manipulate the task more efficiently than the host computer. Each processing element is delegated the responsibility of an object under observation by the radar system. Each processing element maintains a data file for specific objects within its memory and uses its arithmetic capability to continually update its respective file.

The processing element operation and control are directed by the three global control units as follows: All the processing element arithmetic units, correlation units, and associative output units are controlled simultaneously by the arithmetic control unit, correlation control unit and associative output control unit, respectively.

In applying PEPE to radar data processing, the data for each tracked object are stored in a separate processing element memory. The arithmetic units are used to execute tracking and other programs on all or a selected subset of the tracked objects simultaneously. The control units are used to input new radar data into the correct processing element memories, and radar orders are obtained from the processing element memories by the arithmetic output units for transmission to the radar interface computer, concurrent with the processing in the arithmetic units. Arithmetic programs, correlation programs and output programs can be executed independently and simultaneously.

Each arithmetic unit, correlation unit and arithmetic output unit contains a content-addressed tag register, and only those arithmetic units, correlation units and arithmetic output units whose tag register contents match the currently specified "activity" tags will be set to perform subsequently issued instructions. All these units have the capability to make less-than, greater-than, and equal to comparisons between their contents and data issued by the respective control unit. This capability provides content-addressed access to object data and is achieved through a set of registers whose contents can be compared with the input data stream. Only data meeting comparative criteria is stored in the processing element memory.

Arithmetic processes such as track updating, track prediction, discrimination, and interceptor guidance are performed in parallel by the arithmetic units. All the activated arithmetic units simultaneously execute instruction signals issued by the arithmetic control unit.

Input of new information to the processing element is handled by the correlation unit under control of the correlation control unit. The correlation processes of the correlation unit consist of comparing newly received object position information (as derived from radar returns) with predicted object position information generated by the processing element arithmetic units and transferred to the respective correlation units. Information on one object at a time is broadcast by the correlation control unit to all correlation units simultaneously, and all or a selected subset of correlation units compare their stored data with the broadcast data. The object information is input to the correlation unit (or correlation units), where correlation occurs, or into the first empty processing element memory in those cases where no correlation occurs.

Output information sent to the radar is handled by the associative output unit under the control of the associative output control unit and control system through the output data control. Allocation of pulses for the radar consists of ordering pulse requests generated by the processing elements as the result of object file updates calculated by the arithmetic units and stored in the respective element memories for access. The ordered retrieval of the pulse requests on one object at a time are handled by the associative output control unit using a maximum-minimum search for associatively addressing data.

Bit-Serial Associative Processors

Because of the expensive logic in each memory bit and the communication problems in fully parallel associative processors, the bit-serial word-parallel associative processor using the concept of parallel processing with vertical data (one bit-column of a large number of words is processing at a time) was introduced by Shooman in 1960 [63]. His system is essentially a hypothetical vertical data processing computer (referred to as an orthogonal computer) which embodies both vertical data processing and conventional (referred to as horizontal data processing) techniques. Shooman also gave descriptions and algorithms for several vertical data processing instructions. Since the number of words to be processed is usually larger than the number of bits in each word, this approach represents a compromise between fully parallel and word-serial associative processing.

Since then, this concept has resulted in many proposals for associative processors. Kaplan [14] proposed a bit-serial associative memory which he called a search memory; this memory may be used as a subsystem for a general-purpose computer. The main memory may communicate via a memory register with the search memory subsystem, accumulator, arithmetic unit, control unit and input/output unit. The match logic to execute search operations was placed in the search memory subsystem. Ewing and Davies [28] proposed the design logic of a bit-serial associative processor. The block diagram of a bit-serial associative memory with the ALU is shown in Fig. 7. In this memory, storage for one bit is provided at each intersection of a word line and a bit line, and only one bit-column is operated one at a time. The particular bit-column is selected by the bit-column select logic. A pulse on a bit line causes a signal to be emitted by each bit on each word line. The signals are transmitted through the word lines to the sense amplifiers. The word logic associated with each word line gives the ability to perform associative processing. This logic is identical for all words and consists of a sense amplifier, storage flip-flops, a write amplifier and control logic. The storage flip-flop remembers the match state from one interrogating bit to the next. The output of the sense amplifier determines the state of the storage flip-flops in various ways as determined by the control signals from the control unit. The capability of the storage flip-flops to act as a shift register provides the communication link between adjacent words. Such a bit-serial associative processor can be considered as an external logic associative processor, in contrast to a distributed logic associative processor.

Chu [3] proposed the implementation of a bit-serial associative memory which makes use of conventional destructive-readout magnetic memory elements. This memory has two-dimensional read/write capability, resulting in two word lengths: a short-word length which is the number of bits in a word and a long-word length which is the number of words in a bit-column because the number of bits in a word is usually smaller than the number of words. This memory can read or write in either a horizontal or vertical direction of the array, called the short-word mode or the long-word mode, respectively. The short-word mode is the conventional memory organization. The long-word mode is equivalent to bit-serial associative technique.

Bit-serial associative processing through the use of 2-1/2 D core search memory was reported by many researchers [64], [65]. Fulmer and Meilander [66] presented a modular plated-wire implementation of a bit-serial associative processor which has arithmetic capability as well as storage and logic capability at each word of the memory.

Goodyear Associative Processor (GAP), developed by Goodyear Aerospace Corporation [67], uses the processor modules as basic building blocks. Each processor module contains 256 plated-wire processing elements. Each plated-wire processing element consists of one plated-wire, which is a memory device for one 256-bit word, and one response store, whose function is to signal the matching of the word, stored in the plated-wire. The limit on the number of processor modules largely depends on the hardware physical size (a single plated-wire module occupies about 0.5 cubic foot) and the processor's speed requirements.

One of the two large-scale associative processors developed to data is Goodyear Aerospace Corporation's STARAN [43] - [46]. Because of its importance we are going to discuss it in more detail.

STARAN. The basic structure of STARAN is shown in Fig. 8. It consists of a control system and a number of associative array modules (current system configuration allows up to 32 such modules). Each associative array module contains a 256-word × 256-bit multi-dimensional access memory, 256 simple processing elements, a flip network (or called permutation network), and a selector as shown in Fig. 9. There is a simple processing element for each of the 256 words of the memory, and each simple processing element operates serially by bit on the data in the memory word. This operational concept is shown in Fig. 10. Using the flip network, the data stored in the multi-dimensional access memory can be accessed through the input/output channel either in the bit direction or the word direction. A flip network (or permutation network) is also used for shifting and rearranging of data in an associative array module so that parallel search, arithmetic or logical operations can be performed between words of the multi-dimensional access memory. By proper arrangements, the multi-dimensional access memory can be implemented using random access memory chips [45], [68].

To locate a particular data item, STARAN initiates a search by calling for a match against an input data item. All words in the memories of all the modules that satisfy the search criterion are identified by a single instruction. The simple processing elements simultaneously execute operations as specified by the associative control logic. Therefore, in one instruction execution, the data in all selected word in the memories of all the modules are processed simultaneously by the simple processing element at each word.

The interface unit involves interface with sensors, conventional computers, a signal processor, interactive displays and mass storage devices. A variety of I/O options are implemented in the custom interface unit, including the direct memory access (DMA), buffered I/O (BIO) channels, external function (EXF) channels and parallel I/O (PIO). Each associative array module can have up to 256 inputs and 256 outputs into the custom interface unit. They can be used to increase speed of inter-array data communication, allow STARAN to communicate with a high-bandwidth I/O device, and allow any device to communicate directly with the associative array modules.

A mass storage device, like a multihead disk, is connected to the associative array modules via the PIO. The information transfer rates obtained with this configuration depend on the cycle time and the number of heads on the disk being used. In requesting data from the disk, STARAN will send the disk one or more external functions specifying a starting sector address, the number of sectors, and the direction of transfer. The disk system may interrupt STARAN when the disk reaches the requested sector to initiate the transfer over PIO lines. The STARAN control instructions that actually read or write the PIO can be synchronized to the disk so that STARAN timing is slaved to the disk timing during the transfer.

In 1973, an operational associative processor facility, called RADCAP, was installed at Rome Air Development Center [69] - [72]. This facility consists of a STARAN and various peripheral devices, all interfaced with a Honeywell Information Systems (HIS) 645 sequential computer which runs under the Multics time-sharing operating system. The objective of the RADCAP facility is to explore various applications of the system to various real-time problems.

Other Bit-Serial Associative Processors. In addition to those presented before, several other bit-serial associative processors have been developed recently. Sanders Associates [73] developed the OMEN computer in which a conventional serial processor, such as DEC PDP-11, and a bit-serial associative processor both address an orthogonal memory, which has a capacity of 64 words $\times$ 16-bits. The associative processor contains 64 identical processors which form the vertical arithmetic unit that has bit-slice access to the orthogonal memory. These 64 processors perform the same operations at the same time under the control of masks.

Hughes Aircraft Co. [74] developed an associative processor which contains 10 bit-serial associative memories and an MOS shift-register bulk memory. The bulk memory consists of a set of MOS shift registers, each having, at least 16,000 bits each. The purpose of this configuration is to achieve efficient operation of an associative memory when the data base is stored in a large inexpensive mass storage device.

Raytheon Co. [36] developed the Raytheon Associative/Array Processor (RAP) which contains a processing element array and a direct array access channel which facilitates bulk data transfer to/from the processing element array. The function of the processing element is to perform search, arithmetic and logic operation on data stored in its own private memory. Each processing element can be thought of as a bit-serial microprocessor with associative capability.

It should be noted that byte-serial associative processors are conceived between bit-serial and fully parallel associative processors. For reasons of efficiency, at a reasonable cost, a byte-serial word-parallel associative processor, called the Associative Processor Computer System (APCS), was proposed by Linde, Gates and Peng [75] at System Development Corporation. APCS contains two associative processing units and a parallel input/output channel. The word logic consists of byte operation logic, instead of bit operation logic as in bit-serial associative processors, and tags.

Word-Serial Associative Processors

As mentioned before, a word-serial associative processor essentially represents hardware implementation of a simple program loop for search. The important factor which contributes to the relative efficiency of this approach as compared with programmed search in standard sequential processor is that the instruction decoding time is reduced since the search operation requires a single instruction in the word-serial associative processor.

In 1962 Young [76] proposed to use "circulating" associative memories to allow many memory words to time-share a single set of content-addressing logic. In 1966, Crofut and Sottile [77] presented a word-serial associative processor based on a word-serial associative memory using n ultrasonic digital delay lines operating at 100 MHz with 10 μsec delay time, where n is the number of bits of a word. Each delay line stores one bit of the word, and all bits of the stored word propagate down the delay lines synchronously. A stable oscillator (Stalo) was used to generate the synchronizing clock pulses for advancing the address counter. Individual words can be interrogated and updated when they appear at the output of delay lines. The rewrite control logic allows the delay-line system to select either recirculating information or new data inputs. The operational characteristics of such a memory resemble that of a drum or disk. Such a word-serial associative processor is shown in Fig. 11. In 1969, Rux [78] presented a word-serial associative memory with 35 glass delay lines storing 2046 bits per line at 20.48 MHz which was connected to a general purpose medium-speed sequential computer called NEBULA [79], [80].

Because of the slow speed of word-serial associative memories only experimental models have been developed for word-serial associative processors.

Block-Oriented Associative Processors

For applications such as information storage and retrieval where a large storage capacity is required, neither bit-serial nor word-serial associative processors are cost-effective. Bit-serial associative processors become too expensive, while word-serial associative processors require a long processing time. Thus, block-oriented associative processors, which use mass rotating storage devices, such as a disk to provide a limited degree of associative capabilities, have been proposed [81] - [85].

Slotnick [81] and Parker [82] presented the concept of logic-per-track devices which consist of a head-per-track disk memory with some logic associated with each track. Based on this concept and Lee's distributed logic memory for information storage and retrieval applications, Parhami [83] presented a block-oriented associative processor, called RAPID (stands for Rotating Associative Processor for Information Dissemination), which can be shown in Fig. 11. Since the data rates between head-per-track disks and distributed logic memory is high, the RAPID system is suitable for applications requiring a large storage capacity which presently suffer from the high cost of random access memories or from performance degradation due to the frequent transfers between primary and secondary memories.

Minsky [84] proposed the associativity on rotating memories either in the form of drums or disks. He defined the term "partially associative memory" by specifying the primitive structure of information (name part and data part) to be stored on it and the operational characteristics (predicates and instructions). The activity of the memory is supervised by a special processor, called controller. Instead of spending the time in looking for a given address, he proposed to use the delay time for a search for a content.

Another block-oriented associative processor has been proposed by Healy, Lipovski and Doty [85] based on storage and retrieval from a segmented sequential table data structure utilizing associative addressing.

Summary

In this paper, we have reviewed the architecture of various associative processors and classified them into four major categories based on the organization of their associative memories. Among these associative processors, PEPE and STARAN are most well-known. Lloyd and Merwin [86] have made an evaluation of the performance of PEPE, STARAN and others in a real-time environment. In general, fully parallel and bit-serial associative processors are used for high-speed parallel data processing which cannot be carried out effectively in ordinary sequential computers. However, their implementation costs are also higher. For low-cost associative processing which is required in large information storage and retrieval systems, block-oriented associative processors offer a promising architecture.

Acknowledgment

The authors would like to thank Professors T. Y. Feng of Wayne State University, M. J. Gonzalez, Jr. of Northwestern University and Dr. Richard E. Merwin of U.S. Army Ballistic Missile Defense Program Office for many helpful discussions. They also would like to thank Capt. A. R. Klayton of U.S. Air Force Rome Air Development Center for providing useful information on the STARAN/RADCAP system.

References

[1] R. R. Seeber, "Cryogenic Associative Memory," Proc. Assoc. Computing Machinery Nat. Conf., Aug. 1960.

[2] M. H. Lewin, H. R. Beelitz and J. A. Rajchman, "Fixed Associative Memory Using Evaporated Organic Diode Arrays," Proc. 1963 Fall Joint Computer Conf., pp. 101-106.

[3] Y. H. Chu, "A Destructive-Readout Associative Memory," IEEE Trans. on Electronic Computers, Vol. EC-14 (Aug. 1965), pp. 600-605.

[4] A. A. Hanlon, "Content-Addressable and Associative Memory Systems: A Survey," IEEE Trans. on Electronic Computers, Vol. EC-15 (Aug. 1966), pp. 509-521.

[5] A. Weinberger, "The Hybrid Associative Memory Concept," Computer Design (Jan. 1971), pp. 77-85.

[6] J. Minker, "An Overview of Associative or Content-Addressable Memory Systems and a KWIC Index to the Literature," Computing Reviews, Vol. 12 (Oct. 1971), pp. 453-504.

[7] B. Parhami, "Associative Memories and Processors: An Overview and Selected Bibliography," Proc. of the IEEE, Vol. 61 (June 1973), pp. 722-730.

[8] M. Quinones, "An Associative-Capacitive ROM for Reprogrammable Logic Applications," Computer Design (Jan. 1974), pp. 98-101.

[9] A. Slade and H. O. McMahon, "A Cryotron Catalog Memory System," Proc. 1956 East Joint Computer Conf., pp. 115-120.

[10] M. H. Lewin, "Retrieval of Ordered Lists from a Content-Addressed Memory," RCA Reviews, Vol. 23 (June 1962), pp. 215-229.

[11] V. L. Newhouse and R. E. Fruin, "A Cryogenic Data Addressed Memory," Proc. 1962 Spring Joint Computer Conf., pp. 89-99.

[12] A. D. Falkoff, "Algorithms for Parallel Search Memories," Jour. Assoc. Computing Machinery (Oct. 1962), pp. 488-511.

[13] E. C. Joseph and A. Kaplan, "Target-Track Correlation with a Search Memory," Proc. Nat. Conv. on Military Electronics (June 1962), pp. 255-261.

[14] A. Kaplan, "A Search Memory Subsystem for a General-Purpose Computer," Proc. 1963 Fall Joint Computer Conf., pp. 193-200.

[15] R. G. Gall, "A Hardware-Integrated GPC/ Search Memory," Proc. 1964 Fall Joint Computer Conf., pp. 159-173.

[16] G. G. Pick, "A Read-Only Multi-Megabit Parallel Search Associative Memory," The 26th Annual Meeting of American Documentation, Oct. 1963.

[17] A. Estrin and R. Fuller, "Algorithms for Content-Addressable Memories," IEEE Pacific Computer Conf. Proc. (March 1963), pp. 118-128.

[18] C. Y. Lee and M. C. Paull, "A Content Addressable Distributed Logic Memory with Applications to Information Retrieval," Proc. IEEE, Vol. 51 (June 1963), pp. 924-932.

[19] R. B. Derickson, "A Proposed Associative Pushdown Memory," Computer Design (March 1968), pp. 60-66.

[20] N. K. Natarajan and P. A. V. Thomas, "A Multiaccess Associative Memory," IEEE Trans. on Computers, Vol. C-18 (May 1969), pp. 424-428.

[21] Y. H. Chu, Computer Organization and Microprogramming, Prentice-Hall, Inc., 1972.

[22] R. H. Fuller, "Content-Addressable Memory Systems," Disser. Absts., Vol. 24 (Nov. 1963), p. 1960, 611 pp.

[23] S. Nissim, "Organizing the Nanophile Computers," Electronic Design, Vol. 11 (March 1963), pp. 44-53.

[24] W. L. McDermid and H. E. Peterson, "A Magnetic Associative Memory System," IBM J. Res. and Dev., Vol. 5 (Jan. 1961), pp. 59-62.

[25] J. R. Kiseda, H. E. Peterson, W. C. Seelback and M. Teig, "A Magnetic Associative Memory," IBM J. Res. and Dev., Vol. 5 (April1961), pp. 106-121.

[26] R. T. Hunt, D. L. Snider, J. Suprise and H. N. Boyd, "Study of Elastic Switching for Associative Memory Systems," U.S. Gov. Res. Repts., Vol. 39, p. 188(A), AD432041, May 20, 1964.

[27] E. L. Younker, C. H. Heckler, D. P. Masher and J. M. Yarbourough, "Design of an Experimental Multiple Instantaneous Reference File," Proc. 1964 Spring Joint Computer Conf., pp. 515-528.

[28] R. G. Ewing and P. M. Davies, "An Associative Processor," Proc. 1964 Fall Joint Computer Conf., pp. 147-158.

[29] E. S. Lee, "Associative Techniques with Complementing Flip-Flops," Proc. 1963 Spring Joint Computer Conf., pp. 381-394.

[30] R. R. Lussier and R. P. Schneider, "All Magnetic Content Addressed Memory," Electronic Indust., Vol. 22 (March 1963), pp. 92-98.

[31] J. E. McAteer, J. A. Capobianco and R. L. Koppel, "Associative Memory System Implementation and Characteristics," Proc. 1964 Fall Joint Computer Conf., pp. 81-92.

[32] M. F. Wolff, "What's New in Computer Memories," Electronics (Nov. 1963), pp. 35-39.

[33] J. I. Raffel and T. S. Crowther, "A Proposal for an Associative Memory Using Magnetic Films," IEEE Trans. on Electronic Computers (Short Notes), Vol. EC-13 (Oct. 1964), p. 611.

[34] G. G. Pick, "A Semipermanent Memory Utilizing Correlation Addressing," Proc. 1964

Fall Joint Computer Conf., pp. 107-121.

[35] G. T. Tuttle, "How to Quiz a Whole Memory at Once," Electronics (Nov. 1963), pp. 43-46.

[36] G. R. Couranz, M. S. Gerhardt and C. J. Young, "Programmable Radar Signal Processing Using the RAP," Proc. of the Sagamore Computer Conference on Parallel Processing (Aug. 1974), pp. 37-52.

[37] B. A. Crane, M. J. Gilmartin, J. H. Huttenhoff, P. T. Rux and R. R. Shively, "PEPE Computer Architecture," IEEE COMPCON (1972), pp. 57-60.

[38] D. E. Wilson, "The PEPE Support Software System," IEEE COMPCON (1972), pp. 61-64.

[39] J. A. Cornell, "Parallel Processing of Ballistic Missile Defense Radar Data with PEPE," IEEE COMPCON (1972), pp. 69-72.

[40] A. J. Evensen and J. L. Troy, "Introduction to the Architecture of a 288-Element PEPE," Proc. of the 1973 Sagamore Computer Conf. on Parallel Processing,(Aug. 1973), pp. 162-169.

[41] J. R. Dingeldine, H. R. Martin and W. M. Patterson, "Operating System and Support Software for PEPE," Proc. of the 1973 Sagamore Computer Conf. on Parallel Processing (Aug. 1973), pp. 170-178.

[42] C. R. Vick and R. E. Merwin, "An Architecture Description of a Parallel Element Processing Element," Proc. 1973 International Workshop on Computer Architecture.

[43] J. A. Rudolph, "A Production Implementation of an Associative Array Processor: STARAN," Proc. 1972 Fall Joint Computer Conf., pp. 229-241.

[44] K. E. Batcher, "Flexible Parallel Processing and STARAN," WESCON Tech. Papers, Sept. 1972.

[45] K. E. Batcher, "STARAN Parallel Processor System Hardware," Proc. 1974 National Computer Conf., pp. 405-410.

[46] E. W. Davis, "STARAN Parallel Processor System Software," Proc. 1974 National Computer Conf., pp. 17-22.

[47] E. A. Behnke and G. B. Rosenberger, "Cryogenic Associative Processor," IBM Final Report, Sept. 1963.

[48] A. E. Slade and C. R. Smallman, "Thin Film Cryotron Catalog Memory," Automatic Control, Vol. 13 (Aug. 1960), pp. 48-50.

[49] R. F. Rosin, "An Organization of an Associative Cryogenic Computer," Proc. 1962 Spring Joint Computer Conf., pp. 203-212.

[50] P. M. Davies, "A Superconductive Associative Memory," Proc. 1962 Spring Joint Computer Conf., pp. 79-88.

[51] R. W. Ahrons, "Superconductive Associative Memories," RCA Reviews, Vol. 24 (Sept. 1962), pp. 325-354.

[52] J. D. Barnard, F. A. Behnke, A. B. Lindquist and R. R. Seeber, "Structure of a Cryogenic Associative Processor," Proc. IEEE, Vol. 52 (Oct. 1964), pp. 1182-1190.

[53] S. S. Yau and C. C. Yang, "A Cryogenic Associative Memory System for Information Retrieval," Proc. National Electronic Conf., Vol. 22 (Oct. 1966), pp. 764-769.

[54] C. C. Yang and S. S. Yau, "A Cutpoint Cellular Associative Memory," IEEE Trans. on Electronic Computers, Vol. EC-15 (Aug. 1966), pp. 522-528.

[55] C. Y. Lee, "Intercommunicating Cells, Basis for a Distributed Logic Computer," Proc. 1962 Fall Joint Computer Conf., pp. 130-136.

[56] R. P. Edwards, "Content-Addressable Distributed-Logic Memories," Proc. IEEE, Vol. 52 (Jan. 1964), pp. 83-84.

[57] E. S. Spiegelthal, "A Content Addressable Distributed Logic Memory with Applications to Information Retrieval," Proc. IEEE, Vol. 52 (Jan. 1964), p. 74.

[58] C. Y. Lee, "Content-Addressable and Distributed Logic Memories," Applied Automata Theory, J. T. Tou, Ed., New York; Academic Press, 1968.

[59] R. S. Gains and C. Y. Lee, "An Improved Cell Memory," IEEE Trans. on Electronic Computers (Feb. 1965), pp. 72-75.

[60] B. A. Crane and J. A. Crithens, "Bulk Processing in Distributed Logic Memory," IEEE Trans. on Electronic Computers, Vol. EC-14 (April 1965), pp. 186-196.

[61] G. J. Lipovski, "The Architecture of a Large Associative Processor," Proc. 1970 Spring Joint Computer Conf., pp. 385-396.

[62] B. A. Crane and R. R. Laane, "A Cryoelectronic Distributed Logic Memory," Proc. 1967 Spring Joint Computer Conf., pp. 517-524.

[63] W. Shooman, "Parallel Computing With Vertical Data," Proc. 1960 East Joint Computer Conf., pp. 111-115.

[64] P. A. Harding and M. W. Rolund, "A 2-1/2D Core Search Memory," Proc. 1968 Fall Joint Computer Conf., pp. 1213-1218.

[65] H. S. Stone, "Associative Processing for General Purpose Computers Through the Use of Modified Memories," Proc. 1968 Fall Joint Computer Conf., pp. 949-955.

[66] L. C. Fulmer and W. C. Meilander, "A Modular Plated Wire Associative Processor," IEEE Int. Computer Group Conf. Proc. (June 1970), pp. 325-335.

[67] J. A. Rudolph, L. C. Fulmer and W. C. Meilander, "The Coming of Age of the Associative Processor," Electronics (Feb. 15, 1971), pp. 91-96.

[68] K. E. Batcher, "The Multi-Dimensional Access Memory in STARAN," Proc. of the 1975 Sagamore Computer Conf. on Parallel Processing.

[69] J. D. Feldman and L. C. Fulmer, "RADCAP - An Operational Parallel Processing Facility," Proc. 1974 National Computer Conf., pp. 7-15.

[70] J. D. Feldman and O. A. Reiman, "RADCAP: An Operational Parallel Processing Facility," Proc. of the 1973 Sagamore Computer Conf. on Parallel Processing (Aug. 1973), pp. 140-146.

[71] K. E. Batcher, "STARAN/RADCAP Hardware Architecture," Proc. of the 1973 Sagamore Computer Conf. on Parallel Processing (Aug. 1973), pp. 147-152.

[72] E. W. Davis, "STARAN/RADCAP System Software," Proc. of the 1973 Sagamore Computer Conf. on Parallel Processing, (Aug. 1973), pp. 153-159.

[73] L. C. Higbie, "The OMEN Computers: Associative Array Processors," IEEE COMPCON (1972), pp. 287-290.

[74] H. H. Love, Jr., "An Efficient Associative Processor Using Bulk Storage," Proc. of the 1973 Sagamore Computer Conf. on Parallel Processing (Aug. 1973), pp. 103-112.

[75] R. R. Linde, R. Gates and T. F. Peng, "Associative Processor Applications to Real-Time Data Management," Proc. 1973 National Computer Conf., pp. 187-195.

[76] F. H. Young, "Circulating Associative Memories," Dept. of Math. Rept., Oregon State University, 1962.

[77] W. A. Crofut and M. R. Sottile, "Design Techniques of a Delay-Line Content-Addressed Memory," IEEE Trans. on Electronic Computers (Aug. 1966), pp. 529-534.

[78] P. T. Rux, "A Glass Delay Line Contet-Addressable Memory System," IEEE Trans. on Computers (June 1969), pp. 512-520.

[79] J. A. Boles, P. T. Rux and W. Weingarten, Jr., "NEBULA: A Digital Computer Using a 20 Mc Glass Delay Line Memory," Comm. Assoc. Computing Machinery (July 1966), pp. 503-508.

[80] W. Weingarten, "On an Associative Memory for the NEBULA Computer," Dept. of Math. Rept., Oregon State University, 1964.

[81] D. L. Slotnick, "Logic per Track Devices," Advances in Computers, Vol. 10, New York: Academic Press, 1970, pp. 291-296.

[82] J. L. Parker, "A Logic Per Track Retrieval System," IFIP Congress, 1971.

[83] B. Parhami, "A Highly Parallel Computing System for Information Retrieval," Proc. 1972 Fall Joint Computer Conf., pp. 681-690.

[84] N. Minsky, "Rotating Storage Devices as Partially Associative Memories," Proc. 1972 Fall Joint Computer Conf., pp. 587-595.

[85] L. D. Healy, G. J. Lipovski and K. L. Doty, "The Architecture of a Context Addressed Segment-Sequential Storage," Proc. 1972 Fall Joint Computer Conf., pp. 691-701.

[86] G. R. Lloyd and T. E. Merwin, "Evaluation of Performance of Parallel Processors in a Real-Time Environment," Proc. 1973 National Computer Conf., pp. 101-108.

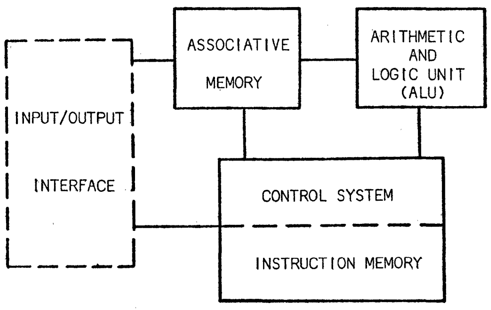


Fig. 1. An associative processor.

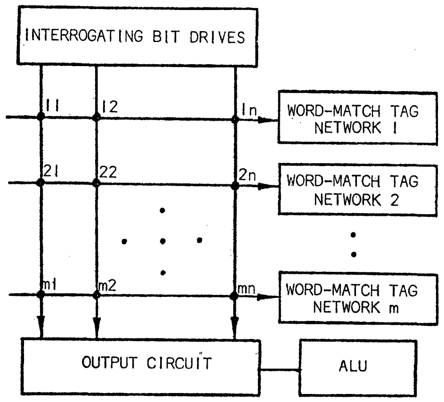


Fig. 2. A fully parallel word-organized associative memory and ALU.

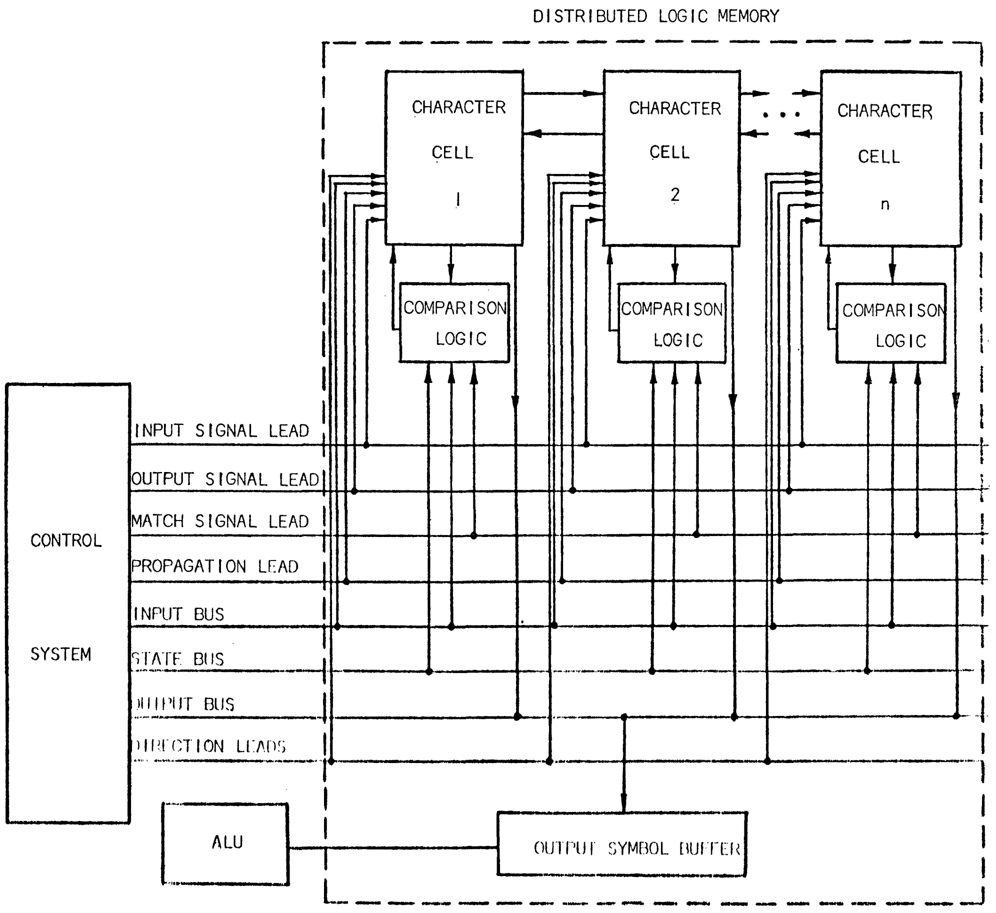


Fig. 3. A fully parallel distributed logic associative processor.

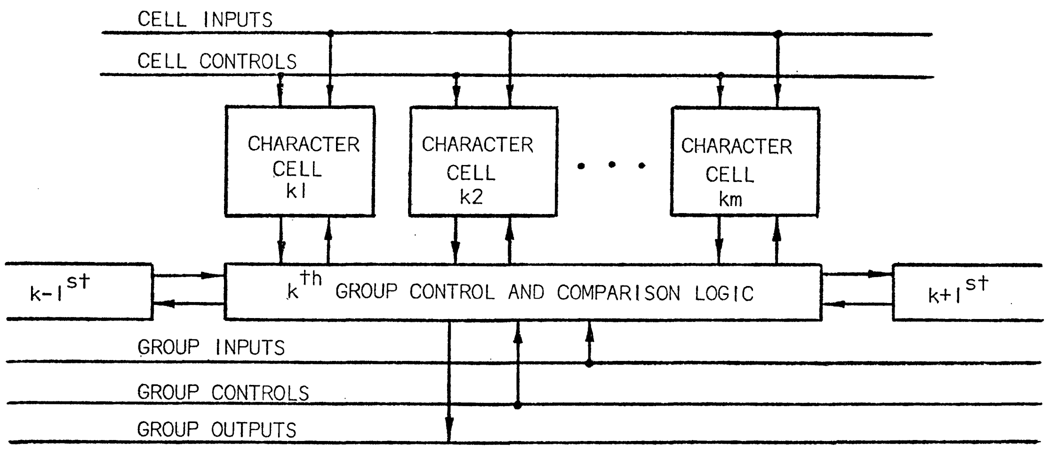


Fig. 4. A fully parallel two-dimensional distributed logic associative memory.

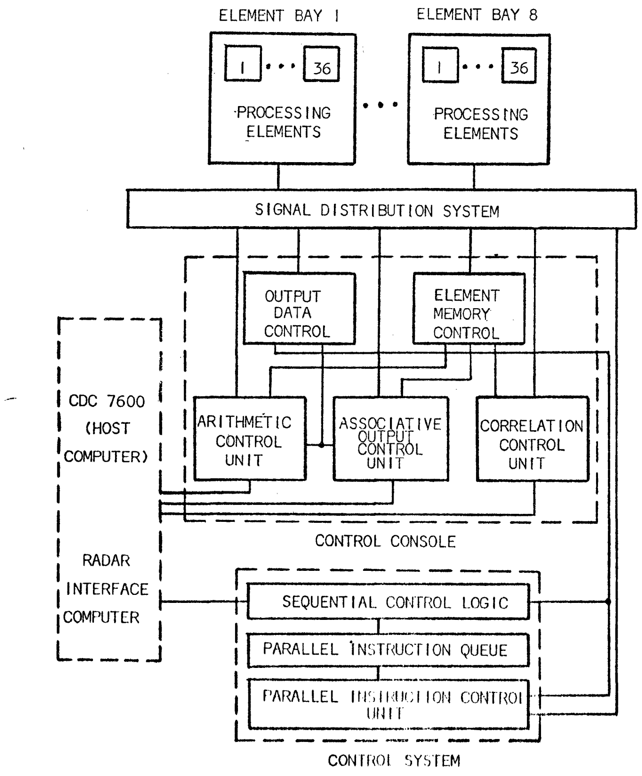


Fig. 5. The parallel element processing ensemble (PEPE) - a modified distributed logic associative processor.

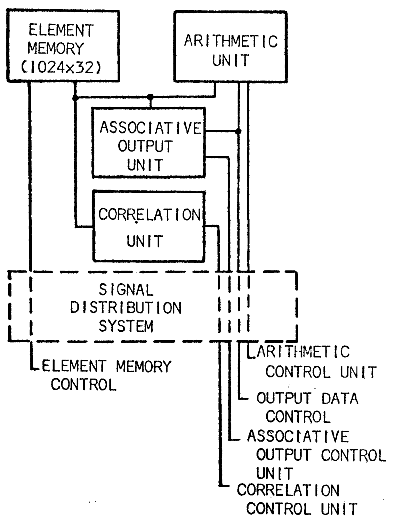


Fig. 6. A processing element of the PEPE.

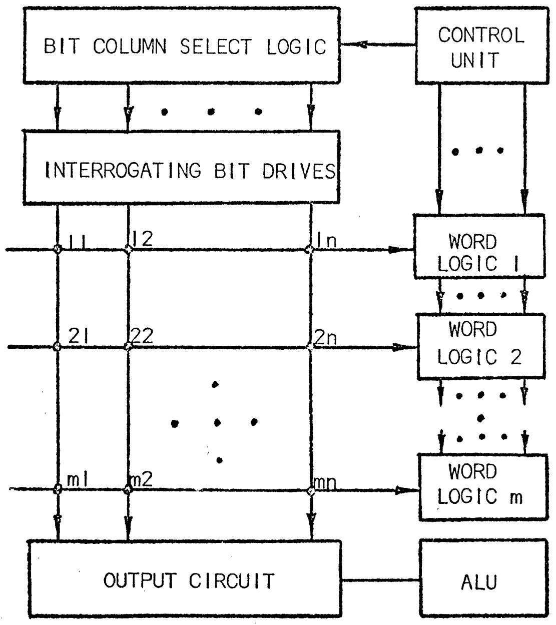


Fig. 7. A bit-serial associative memory and ALU.

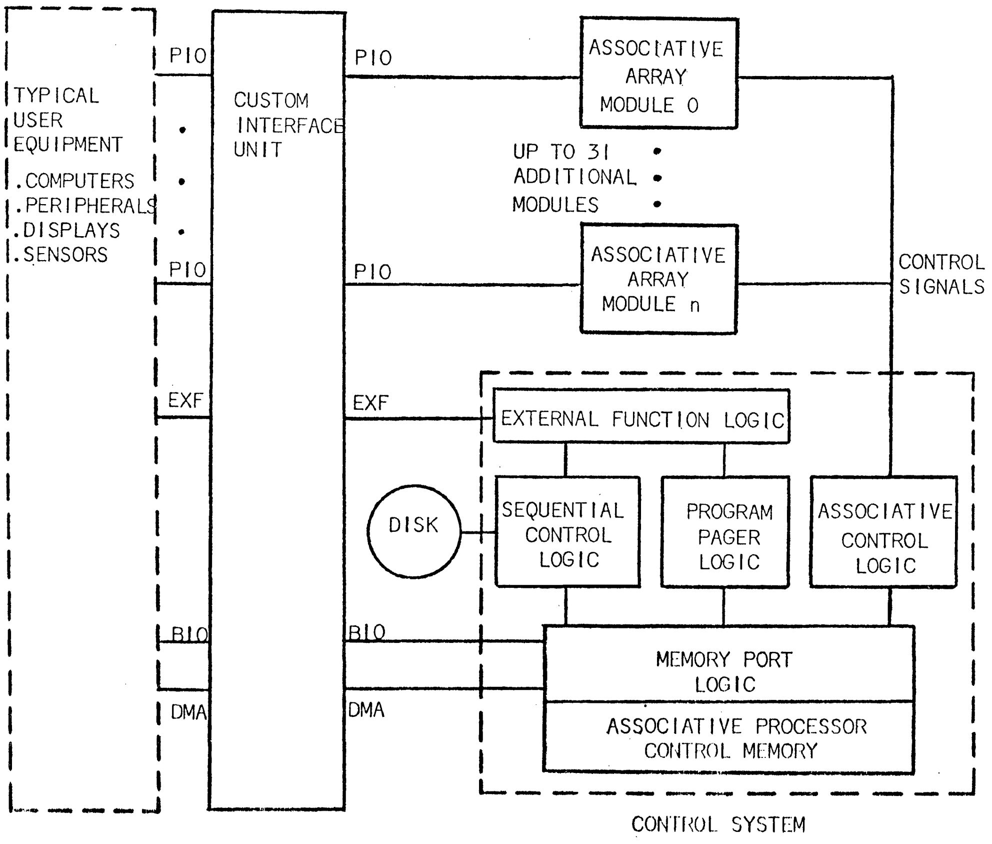


Fig. 8. The STARAN - a bit-serial associative processor.

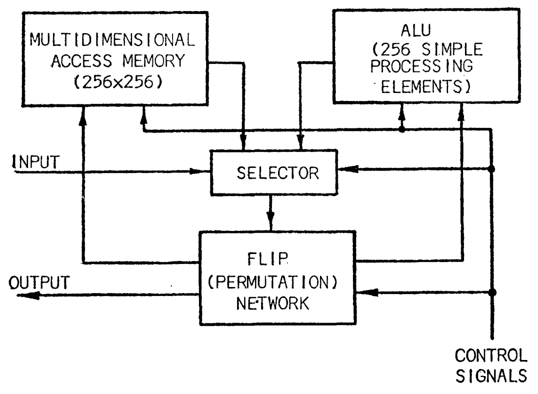

MULTIDIMENSIONAL ACCESS MEMORY (256x256)
ALU (256 SIMPLE PROCESSING ELEMENTS)
INPUT
SELECTOR
FLIP (PERMUTATION) NETWORK
OUTPUT
CONTROL SIGNALS

Fig. 9. An associative array module of the STARAN.

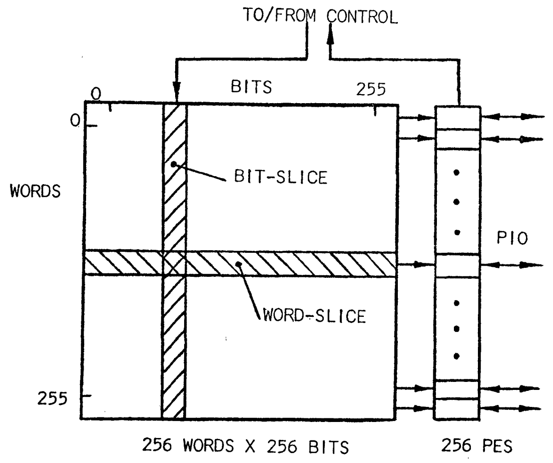


Fig. 10. The operational concept of a STARAN associative array module.

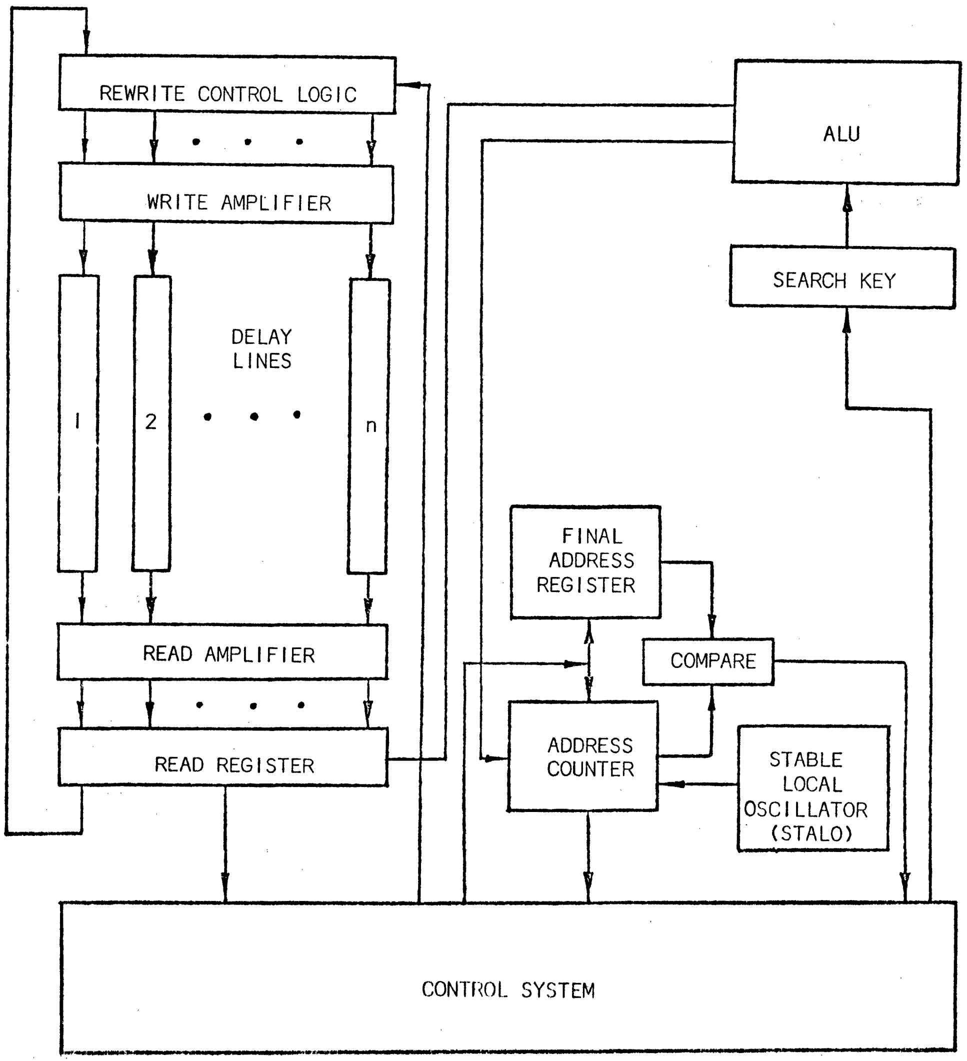


Fig. 11. A word-serial associative processor.

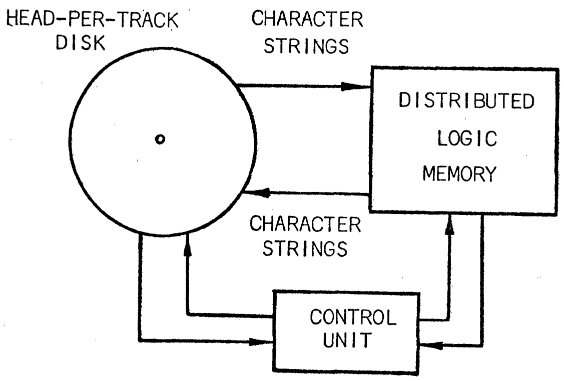


Fig. 12. The block-oriented associative memory used in the RAPID.

PARALLEL PROCESSOR ARCHITECTURE--A SURVEY*

David J. Kuck
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

(Invited Paper)

Abstract

This paper is a survey of parallel machine organizations and programming. We define parallelism in a broad sense, which encompasses the bit level, operation level, and algorithm level. A number of abstract and practical questions are discussed.

I. Introduction

We will interpret "parallel processor architecture" broadly in this paper. In the strict sense, we could limit our attention to a few machines, some already in operation, some being built, and others being proposed. But this would tend to perpetuate certain bits of folklore about parallel machines which puts them in a zoo somewhere between the chimera and the white elephant.

Indeed, some parallel (and other) computers probably belong there. But recent theoretical developments lead us to the conclusion that a broad interpretation of "parallel processing" is justified here. By taking this point of view, we will show that a number of previously diverse ideas can be unified.

It is certainly true that the term "parallel processing" has been used in many ways since the dawn of computer history. Twenty-five years ago it referred to arithmetic operations on whole words rather than one bit at a time. Now, it refers sometimes to multiple processors, other times to array-type machines (parallel or pipeline), and occasionally to multiprogrammed machines. In this paper we will touch on all of the above. Furthermore, we will not restrict our attention to processing the sense of logic and arithmetic. Memory access and data alignment problems are equally important.

Several theoretical results will be given and discussed. It turns out that by looking at things correctly, a few abstract ideas can be interpreted in several important practical ways. For example, by studying recurrence relations we can develop algorithms to transform sequential logic equations (e.g., for bit serial arithmetic) into combinational circuit diagrams (e.g., for fast carry lookahead arithmetic). Using exactly the same theory, we can develop compiler algorithms to transform standard serial programs (e.g., Fortran) into highly parallel programs (e.g., for array processors).

*This work was supported in part by NSF Grant DCR73-07980 A02.

We will also study tree-height reduction techniques which can be interpreted as logic design or compiler algorithms. As another example, we will present new results about the effective bandwidth of parallel memories. This can be interpreted for multiprocessors, associative processors, parallel or pipeline processors. Also, we will discuss data alignment techniques which have interpretations as bit shifters within one processor, as data alignment networks between several processors, or as merge networks between a secondary and primary memory.

Our approach is to study the structure of programs and the structure of certain machine parts to see how the two can be brought together. Our primary goal is to achieve high speed at low cost. Obviously, there are other very important considerations in machine design. But we choose this goal first, for several reasons. For one, it is widely used in practice and is thus practically important. For another, it has a great deal of theoretical importance as follows.

Computer speeds are determined by two basic limitations. One concerns the physics of the hardware being used, e.g., how fast do gates switch and how many inches of wire must a signal traverse? The other concerns the logic of machine organization and program organization. It is the latter question that we shall study.

Suppose that hardware speeds were fixed. Then, given a set of programs, how fast could they be executed? Theoretical lower bounds can be given on the number of time steps needed to compute certain functions. Once these are known for an algorithm or class of algorithms, then we can attempt to design "perfect" machine organizations for these algorithms. Such machines would execute the algorithms in the fastest possible way at the lowest cost. When we mention cost, logic design questions arise. How many gates or integrated circuits or microprocessors, etc., are needed? Real costs are measured in printed circuit bounds, power supplies, cabinets and so on, so "lowest cost" is difficult to define, but logic component counts tend to reflect overall hardware costs. We will deal at various points with bounds on gate, integrated circuit, processor and memory unit counts.

In this paper we will not give much attention to specific algorithms or to lower bounds. Rather, we will study Fortran-like programming languages to see what elementary set of language constructs may be universal to a large class of algorithms. Assignment statements, recurrences, and conditional statements together with program graphs will be discussed in detail. Instead of lower bounds, we will give upper bounds on time and components, all of which can be achieved by algorithms which support the bounds. In many cases, these upper bounds are not far from lower bounds obtained by simple fan-in arguments.

Overall, we are attempting to develop a better understanding of the structure of programs and of the relations of programs to machine organizations. By presenting bounds with constructive proofs, logic design automation or compiler algorithms follow. Furthermore, we repeat that a few theoretical ideas can be given a number of useful interpretations which provide insights in several areas. Finally, we will point out that a better understanding of some fundamentals about program and machine organization may lead to better understanding of structured programming and paging in the memory hierarchies of existing computer systems.

This paper is divided into three main parts. In section II, we give some theoretical background. Then, in section III we use this to study the overall structure of programs. We also use the theory in section IV for processor design. Section V discusses data handling; memory access and data alignment are both considered.

II. Theoretical Background

In this section we will be concerned with upper bounds on processing time and the number of processors required to achieve such time bounds. We will ignore for the moment memory activity, data alignment and control unit times, and assume an idealized multioperation machine. We will allow any number of arithmetic processors to be used at once (although we will bound this number). For expression evaluation, any processor may perform any of the arithmetic operations on any time step. This multiple instruction execution (c.f. MIMD [1,2]) is less desirable than having all processors execute the same instruction type (SIMD [1]). Our recurrence method requires only one instruction type at a time (SIMD). Clearly, expression evaluation is slowed down by at most a small constant if SIMD instructions are used. Further, we assume that each arithmetic operation takes one unit of time. Later in the paper we will discuss more realistic machine details.

If T_p is the number of unit time steps required to perform some calculation using $p \geq 1$ processors, we define the speedup of the p processor calculation over a uniprocessor as $S_p = \frac{T_1}{T_p} \geq 1$ and we define the efficiency of the calculation as $E_p = \frac{S_p}{p} \leq 1$ which may be regarded as actual speedup divided by the maximum possible speedup using p processors. For various computations we will discuss the maximum possible speedup known according to some algorithm and in such cases we use P to denote the minimum number of processors known to achieve this maximum speedup. In such cases we will use the notation T_P, S_P and E_P to denote the corresponding time, speedup and efficiency, respectively.

Time and processor bounds for some computation A will be expressed as $T_P[A]$ and $P[A]$ in the minimum time cases and $T_p[A]$ in the restricted processor ($p < P$) case. When no ambiguity can result, we will write $T[A]$ or just T in place of $T_P[A]$ and P in place of $P[A]$, for simplicity. We write log x to denote $\log_2 x$ and $\lceil x \rceil$ for the ceiling of x.

Arithmetic Expression Tree-Height Reduction

Now we consider time and processor bounds for arithmetic expression evaluation. We restrict our attention to transforming expressions using associativity, commutativity and distributivity which lead us to speedups of $O(\frac{n}{\log n})$. Since this is asymptotic to the best possible speedup, more complex transformations (e.g., factoring, partial fraction expansion seem unnecessary.
In section IV, we shall return to this subject in the form of Boolean expression evaluation for combinational logic circuits.

Definition 1

An arithmetic expression is any well-formed string composed of the four arithmetic operations (+,-,*,/), left and right parentheses, and atoms which are constants or variables. We denote an arithmetic expression E of n distinct atoms by E<n>.

If we use one processor, then the evaluation of an expression containing n operands requires n - 1 units of time. But suppose we may use as many processors as we wish. Then it is obvious that some expressions E<n> may be evaluated in $\log_2 n$ units of time as illustrated in Figure 1.

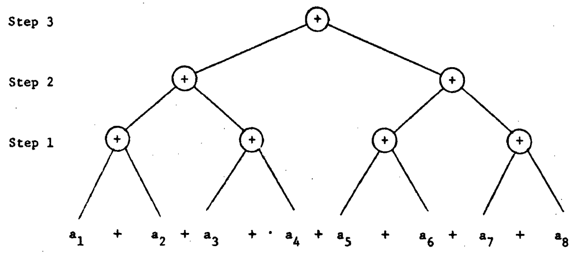

Step 3
Step 2
Step 1
+
a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8

Figure 1. Expression of 8 Atoms

In fact, we can establish, by a simple fan-in argument, the following lower bound:

Lemma 1 Given any arithmetic expression E<n>

$T[E<n>] \geq \lceil \log_2 n \rceil$.

On the other hand, it is easy to construct expressions E<n> whose evaluation appears to require O(n) time units regardless of the number of processors available. Consider the evaluation of a polynomial by Horner's rule:

$$p_n(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + xa_n) \ldots)). \qquad (1)$$

A strict sequential order is imposed by the parentheses in Eq. 1 and more processors than one are of no use in speeding up this expression's evaluation.

However, we are not restricted to dealing with arithmetic expressions as they are presented to us. For example, the associative, commutative, and distributive laws of arithmetic operations may be used to transform a given expression into a form which is numerically equivalent to the original but which may be evaluated more quickly. We now consider examples of each of these.

Figure 2a shows the only parse tree possible (except for isomorphic images) for the expression (((a + b) + c) + d). This tree requires three steps for its evaluation and we refer to this as a tree height of three. However, by using the associative law for addition we may rearrange the parentheses and transform this to the expression (a + b) + (c + d) which may be evaluated as shown in Figure 2b with a tree height of two. It should be noted that in both cases, three addition operations are performed.

Figure 3a shows a parse tree for the expression a + bc + d; again we have a tree of height three. In this case the tree is not unique, but it is obvious that no lower height tree can be found for the expression by use of associativity. But by use of the commutative law for addition we obtain the expression a + d + bc and the tree of Figure 3b, whose height is just two. Again we remark that both trees contain three operations.

Now consider the expression a(bcd + e) and the tree for it given in Figure 4a. This tree has height four and contains four operations. By use of associativity and commutativity, no lower height tree can be found. But, using the arithmetic law for the distribution of multiplication over addition we obtain the expression abcd + ae, which has a tree of minimum height three as shown in Figure 4b. However, unlike the two previous transformations, distribution has introduced an extra operation; the tree of Figure 4b has five operations compared to the four operations of the undistributed form.

Having seen a few examples of arithmetic expression tree-height reduction, we are naturally led to ask a number of questions. For any arithmetic expression, how much tree-height reduction can be achieved? Can general bounds and algorithms for tree-height reduction be given? How many processors are needed?

To answer these questions, we present a brief survey of results concerning the evaluation of arithmetic expressions. Details and further references may be found in the papers cited. Assuming that only associativity and commutativity are used to transform expressions, Baer and Bovet [3] gave a comprehensive tree-height reduction algorithm based on a number of earlier papers. Beatty [4] showed the optimality of this method. An upper bound on the reduced tree height assuming only associativity and commutativity are used, given by Kuck and Muraoka [5], is the following.

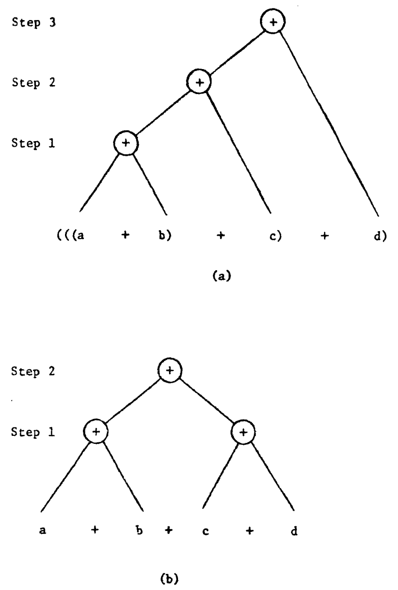


Figure 2. Tree-Height Reduction by Associativity

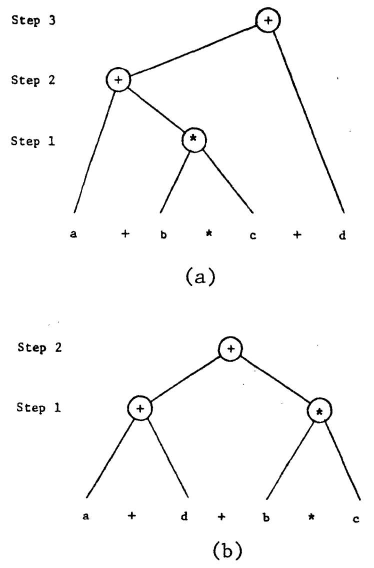
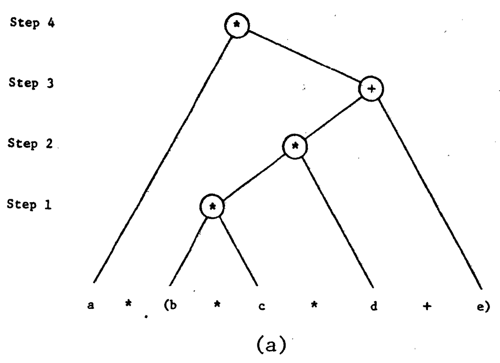


Figure 3. Tree-Height Reduction by Commutativity



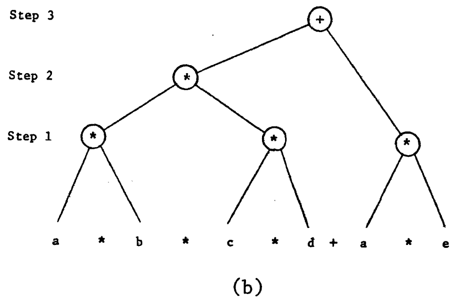


Figure 4. Tree-Height Reduction by Distributivity

Theorem 1 Let $E<n|d>$ be any arithmetic expression with depth d of parenthesis nesting. By the use of associativity and commutativity only, $E<n|d>$ can be transformed such that

$$T_P[E<n|d>] \leq \lceil \log_2 n \rceil + 2d + 1$$

with

$$P \leq \left\lceil \frac{n}{2} - d \right\rceil .$$

Note that if the depth of parenthesis nesting d, is small, then this bound is quite close to the lower bound of $\lceil \log_2 n \rceil$. The complexity of this algorithm has been studied in [6], where it is shown that in addition to the standard parsing time, tree-height reduction can be performed in O(n) steps. Unfortunately, there are classes of expressions, e.g., Horner's rule polynomials or continued fractions for which no speed increase can be achieved by using only associativity and commutativity.

Muraoka [7] studied the use of distributivity as well as associativity and commutativity for tree-height reduction and developed comprehensive tree-height reduction algorithms using all three transformations. An algorithm which considers operations which take different amounts of time is presented by Kraska [8].

Bounds using associativity, commutativity and distributivity have been given by a number of people [9,10,11]. In [10] the following theorem is proved.

Theorem 2 Given any expression $E<n>$, by the use of associativity, commutativity and distributivity, $E<n>$ can be transformed such that

$$T_P[E<n>] \leq \lceil 4 \log n \rceil$$

with

$$P \leq 3n .$$

The complexity of the algorithm of [10] has been studied in [6], where it is shown that tree-height reduction can be done using O(n log n) steps in addition to normal parsing. For expressions of special forms, better bounds can be given [9]. Also, if the number of processors is allowed to grow beyond O(n), the time coefficient of Theorem 2 can be reduced to 2.88 [11].

So we conclude that any arithmetic expression $E<n>$ can be evaluated in O(log n) time steps using O(n) processors. While this affords a nice speedup for large n, we rarely see expressions with n larger than five or six. For bigger speedups, in practice, we turn to the following.

Recurrence Relations

Linear recurrences share with arithmetic expressions a role of central importance in computer design and use. But they are somewhat more difficult to deal with. While an expression specifies a static computational scheme for a scalar result, a recurrence specifies a dynamic procedure for computing a scalar or an array of results. Linear recurrences are found in computer design, numerical analysis and program analysis, so it is important to find fast, efficient ways to solve them.

Recurrences arise in any logic design problem which is expressed as a sequential machine. Also, almost every practical program which has an iterative loop contains a recurrence. While not all recurrences are linear, the vast majority found in practice are, and we shall concentrate first on linear recurrences.

We shall begin with several examples. First, consider the problem of computing an inner product of vectors $a = (a_1,\ldots,a_n)$ and $b = (b_1,\ldots,b_n)$. This can be written as a linear recurrence of the form

$$x = x + a_i b_i, \quad 1 \leq i \leq n \tag{2}$$

where x is initially set to zero and finally set to the value of the inner product of a and b.

As another example of a linear recurrence which produces a scalar result, the evaluation of a degree n polynomial $p_n(x)$ in Horner's rule form can be expressed as

$$p = a_i + xp, \quad 2 \leq i \leq n \tag{3}$$

where p is initially set to a_1 and finally set to the value of $p_n(x)$.

Techniques to handle both of these

recurrences should be familiar from our discussion of expression evaluation. Note that Eq. 2 can be expanded by substituting the right-hand side into itself (statement substitution) as follows:

$$x = a_1b_1$$

$$x = a_1b_1 + a_2b_2$$

$$x = a_1b_1 + a_2b_2 + a_3b_3$$

$$\vdots$$

After n iterations we have an expression which can be mapped onto a tree similar to that of Figure 1.

Earlier, we have also discussed polynomial evaluation. Thus by carrying out a procedure similar to the above, we could obtain an expression which could be handled by tree-height reduction. Thus, we would expect that these and similar recurrences could be solved in $T_P = O(\log n)$ time steps using $P = O(n)$ processors.

But there are other, more difficult looking linear recurrences. For example, a Fibonacci sequence can be generated by

$$f_i = f_{i-1} + f_{i-2} \qquad 3 \leq i \leq n \tag{4}$$

where

$$f_1 = f_2 = 1.$$

As another example, consider the addition of two n-bit binary numbers $a = a_n \ldots a_1$ and $b = b_n \ldots b_1$. The propagation of the carry across the sum can be described by

$$c_i = y_i + x_i \cdot c_{i-1} \qquad 1 \leq i \leq n \tag{5}$$

where

$$c_0 = 0,\ x_i = a_i + b_i \text{ and } y_i = a_i \cdot b_i.$$

Here we use + to denote logical or and • to denote logical and. This is an example of a bit level linear recurrence, in contrast to our previous examples whose arguments were assumed to be real numbers.

In both Eq. 4 and Eq. 5 we are required to generate a vector result because of the subscripted left-hand side. This is in contrast to the scalar results of Eqs. 2 and 3. Because of this, we can expect a good deal more difficulty in trying to obtain a fast efficient solution to these recurrences. With the above as an introduction, we now turn to a formalization of the general problem. We will then give bounds for the solution of the general problem and several important special cases.

Definition 2

An m-th order linear recurrence system of n equations, R<n,m> is defined for $m \leq n$ by

$$x_i = 0 \qquad \text{for } i \leq 0$$

and

$$x_i = c_i + \sum_{j=i-m}^{i-1} a_{ij}\, x_j \qquad \text{for } 1 \leq i \leq n.$$

If m = n we call the system a general linear recurrence system and denote it by R<n>.

Note that we can express any linear recurrence system in matrix terms as

$$x = c + Ax$$

where

$$c = (c_1,\ldots,c_n)^t,\ x = (x_1,\ldots,x_n)^t$$

and A is a strictly lower triangular (banded if $m < n$) matrix with $a_{ij} = 0$ for $i \leq j$ or $i - j > m$. We refer to A as the coefficient matrix, c as the constant vector and x as the solution vector.

It should be observed that the constant vector and coefficient matrix generally contain values which can be computed before the recurrence evaluation begins. Thus, the x_i and y_i values of Eq. 5 would be precomputed from the a_i and b_i. We will assume that the elements of c and A are precomputed (if necessary) in all cases so that our bounds on recurrence evaluation can be simply stated, and that m and n are powers of 2.

How can we solve an R<n> system in a fast, efficient way using many simultaneous operations? The following is a straightforward way which uses O(n) processors to solve the system in O(n) steps.

Column Sweep Algorithm

Given any R<n> system, we initially know the value of x_1. On step 1 we broadcast this value, c_1, to all other equations, multiply by a_{j1} and add the result to c_j. Since we now know the value of x_2, this leads to an R<n-1> system which can be treated in exactly the same way. Thus after n - 1 steps, each of which consists of a broadcast, a multiply and an add, and each of which generates another x_i, we have the solution vector x. The method requires n - 1 processors on step 1, and fewer thereafter, so $T_P = 2(n - 1)$ with $P = n - 1$.

What speedup and efficiency have we achieved by this method? The time required to solve this system using a single processor which might sweep the array by rows or columns would be

$$T_1 = 2[1 + 2 + \ldots + (n - 1)]$$

$$= 2\left[\frac{n(n-1)}{2}\right] = n(n - 1).$$

Hence the above method achieves a speedup of

$$S_P = \frac{n(n-1)}{2(n-1)} = n/2$$

with an efficiency of

$$E_P = \frac{S_P}{P} = \frac{n}{2(n-1)} > \frac{1}{2}\ .$$

Thus we can conclude that the Column Sweep Algorithm is a reasonable method of solving an R<n> system. But how does it perform in the R<n,m> case for m<<n.

It can be seen that the Column Sweep Algorithm will achieve $S_p = O(m)$ for an R<n,m> system. So if m is very small, the method performs poorly, particularly if we have a large number of processors available. It should be noted that the m<<n case occurs very often in practice. Note that all of our examples (Eqs. 2-5) had $m \leq 2$.

What are our prospects for finding a faster algorithm. First, we observe that the total number of initial data values in an R<n,m> system is O(mn). This is the total of the constant vector c and the coefficient matrix A. Assuming that these numbers all interact in obtaining a solution, a fan-in argument [c.f. Lemma 1] indicates that we need at least O(log mn) steps to solve an R<n,m> system, since $m \leq n$, O(log mn) = O(log n). The Column Sweep Algorithm required O(n) steps, so we still have a big gap in time.

Fast Recurrence Method

The next theorem is based on an algorithm for the fastest known method of evaluating an R<n,m> system. For large m, the number of processors required is rather large, but for small m, the number of processors is quite reasonable. We also give bounds for the case of a small number of processors, Corollary 3 is particularly important in the case of m < p < P.

Theorem 3 Any R<n,m> can be computed in

$$T_P \leq (2 + \log m) \log n - \frac{1}{2}(\log^2 m + \log m)$$

with

$$P \leq m^2 n/2 + O(mn) \qquad \text{for } m<<n,$$

$$P \leq n^3/68 + O(n^2) \qquad \text{for } m \leq n.$$

The details of transforming a system to meet this bound are fairly straightforward [12]. We will give a simple example here as a basis for some intuition about how the technique of Theorem 3 works. Consider an R<4,2> system. This method would generate the following expressions for the evaluation of the x_i:

$$x_1 = c_1$$

$$x_2 = (c_2+a_{21}c_1)$$

$$x_3 = (c_3+a_{31}c_1) + a_{32}(c_2+a_{21}c_1)$$

$$x_4 = c_4 + (a_{42}+a_{43}a_{32})(c_2+a_{21}c_1) + a_{43}(c_3+a_{31}c_1).$$

Note that all of the parenthesized expressions can be computed simultaneously in two steps (there are just three distinct ones). Then x_4, the largest calculation, can be completed in three more steps for $T_P = (2+\log 2)(\log 4) - \frac{1}{2}(\log^2 2+\log 2) = 5$.

This time bound may be achieved using just three processors in this case. But as n grows larger, the number of processors required becomes very large as shown in the tables of [13].

In practice we may have a machine with a limited number of processors p < P so Theorem 3 cannot be used directly. Several schemes are available for mapping a computation onto a smaller set of processors and generally increasing the efficiency of the computation as well. While the techniques described below may be applied to arithmetic expressions as derived from Theorems 1 or 2, the expressions found in typical programs usually do not require enough processors to warrant such reductions [14].

First, we describe a folding scheme which reduces the number of processors at a much faster rate than the computation time increases. The P processor computation for R<n,m> resulting from Theorem 3 contains log n stages, each stage consisting of many independent tree computations of height (log m+1) resulting from inner products of two m-vectors. Such a tree of height t will contain 2^t-1 operation nodes and its evaluation requires 2^{t-1} processors. P is the maximum of the total number of processors used at each stage. It is easy to show that given such a tree its height increases only one step by halving the number of processors (called one fold), and after f folds ($f \leq t - 2$) are performed the tree height is $t+2^{f+1}-f-2)$ while the number of processors is reduced to $2^{t-1}/2^f$. If all trees at the same stage are folded uniformly, then this folding scheme can provide us T_p as stated below.

Corollary 1 Let R<n,m> and P be as in Theorem 3. Then if $f \leq \log m - 1$ and $p = \lceil P/2^f \rceil$, we have $T_p \leq T_P + (2^{f+1}-f-2) \log n$.

Another technique which is useful in mapping any computation onto a limited number of processors p < P is the sweeping scheme [15]. If the i-th step of any parallel computation requires O_i operations using P processors, it can be executed on p processors in $\left\lceil \frac{O_i}{p} \right\rceil$ steps. This observation leads to the following:

Lemma 2 [10] If a computation C can be completed in T_P with O_P operations on P processors, then C can be computed in $T_p \leq T_P + (O_P-T_P)/p$ for p < P.

To apply this technique directly on the algorithm of Theorem 3, the O_P value can be obtained by the summation of $2 \cdot p(k)$ for k = 2, 4, 8, ..., n, where p(k) is the number of processors required at each stage [12]. The result of this technique can be found in [17].

Our third scheme for reducing the number of processors required for an R<n,m> system is called the cutting scheme. The idea is to cut the original system into a number of smaller systems and evaluate these in sequence, using the algorithm underlying Theorem 3 on each such system. We have used this scheme in [13], [18], and a detailed proof is given in [17].

Corollary 2 Let R<n,m> and P be as in Theorem 3. Then any R<n,m> can be computed with $1 < p < P$ processors in

$$T_p \le 2[m/p](n-1) \qquad \text{for } 1 < p \le m,$$

$$\le \frac{\beta}{72} np^{\frac{-1}{3}}(\log^2 p + 27\log p + 144) + \frac{2mn}{p} \qquad \text{for } m < p < m^2$$

$$\le \frac{\beta}{72} np^{\frac{-1}{3}}(\log^2 p + 27\log p + 144) \qquad \text{for } m^2 \le p \le m^3,$$

$$\le \beta\frac{m^2 n}{p}(\log m \log p + 2\log_2 p - \frac{5}{2}\log^2 m - \frac{7}{2}\log m + 1) \qquad \text{for } m^3 < p < P,$$

where $\beta(m,n,p)$ is a small constant.

For most practical R<n,m> systems in which m is very small compared to n, if the number of processors is also very limited then a new computational algorithm developed in [12] can be used more efficiently. This method gives the following time bounds.

Corollary 3 Let R<n,m> and P be as in Theorem 3. If $m < p < P$, then any R<n,m> can be computed in

$$T_p \le (2m^2+3m)\frac{n}{p} + O(m^2\log(p/m)).$$

In summary, for $1 < p < P$ the time bound for evaluating a given R<n,m> system can be determined by choosing the minimum value obtained from Corollaries 1, 2 and 3.

III. Program Analysis

In this section we discuss program analysis techniques. These techniques can be used to compile programs for parallel or pipeline computers. They can also be used to specify machine organization for high speed computation. Since we are really just studying the structure of ordinary serial programs, our results have interpretations for ordinary virtual memory machines and structured programming as well.

Definition 3

An assignment statement is denoted by $x = E$, where x is a scalar or array variable and E is a well-formed arithmetic expression. A block of assignment statements (BAS) is a sequence of one or more assignment statements with no intervening statements of any other kind. Any BAS can be transformed by a process called statement substitution to obtain a set of expressions which can be evaluated simultaneously.

For example, the BAS

X = BCD + E

Y = AX

Z = X + FG

can be evaluated using one processor in 6 steps, ignoring memory activity. By statement substitution we obtain three statements which can be transformed by tree-height reduction to obtain:

X = BCD + E; Y = ABCD + AE; Z = BCD + E + FG.

Since the resulting expressions can be evaluated simultaneously in three steps, we obtain a speedup of 2. By properly arranging the parse trees it may be seen that just five processors are required. Thus we have efficiency $E_5 = 2/5$. In general, the number of processors required to evaluate a set of trees in a fixed number of steps may be minimized using an algorithm of Hu [19]. Note that the speedup here results from two effects: the simultaneous evaluation of independent trees and tree-height reduction by associativity, commutativity and distributivity.

Definition 4

An IF statement is denoted by $(C)(S_1,\ldots,S_n)$ where C is the conditional expression composed of arithmetic and logical operations and $S_1,\ldots,S_n$ are n different statements which may be assignment statements, IF statements, or loops such that control will be transferred to one of them depending on the value of C.

In many programs it is possible to find outside DO loops, rather large sets of statements consisting of many IF and GOTO statements with some interspersed assignment statements. Suppose we have a method of discovering sections of code in which the ratio of control (IF, GOTO) statements to arithmetic operations is greater than some small number. We call such a section of code an IF block. Given an IF block, it is straightforward to put it in a canonical form consisting of:

Step 1: A set of assignment statements, all of which may be executed simultaneously.

Step 2: A set of Boolean functions, all of which may be evaluated simultaneously.

Step 3. A binary decision tree through which one path will be followed for each execution of the program. No Boolean function or arithmetic expression evaluation is included in the tree.

Step 4: A collection of blocks of assignment statements, each with a single variable or constant on the right-hand side. One such block is associated with each path through the tree.

The details of an algorithm for the discovery and transformation of an IF block to this canonical form are given by Davis [20]. Note that the IF block may be a graph with or without cycles. Such graphs are converted to trees called IF trees in the cited references.

Definition 5

A loop is denoted by

$$L = (I_1 \leftarrow N_1,\ I_2 \leftarrow N_2,\ \ldots,\ I_d \leftarrow N_d)\ (S_1,\ S_2,\ \ldots,\ S_s)$$

or

$$= (I_1,\ I_2,\ldots,\ I_d)(S_1,\ S_2,\ \ldots,\ S_s)$$

where I_j is a loop index, N_j is an ordered index set, and S_j is a body statement which may be an assignment statement, an IF statement or another loop. We use $OUT(S_j)$ and $IN(S_j)$ to denote for S_j the LHS (output) variable name and the set of RHS (input) variable names, respectively. We will write $S_j(i_1,\ i_2,\ \ldots,\ i_d)$ to refer to S_j during a particular iteration step, i.e., when the index variables of S_j are assigned the specific values $I_1 = i_1,\ I_2 = i_2,\ \ldots,\ I_d = i_d$. If S_i is executed before S_j, we will write $S_i \underset{o}{<} S_j$. We say that the relation $\underset{o}{<}$ defines the execution order of the statements. If a loop execution leads to the execution of n statements, we sometimes denote their execution order by writing $Y_i{:}x_i = E_i,\ 1 \le i \le n$, implying that $Y_i \underset{o}{<} Y_{i+1},\ 1 \le i \le n-1$.

Definition 6

Given a loop $L = (I_1 \leftarrow N_1,\ \ldots,\ I_d \leftarrow N_d)(S_1,\ \ldots,\ S_s)$, all possible data dependencies between statement pairs S_i and S_j are given by $OUT(S_i(k_1,\ldots,k_d)) \cap IN(S_j(\ell_1,\ldots,\ell_d)) \neq \phi$ for $S_i(k_1,\ldots,k_d) \underset{o}{<} S_j(\ell_1,\ldots,\ell_d)$. Whenever this condition is satisfied, we say that S_j is data dependent on S_i, and is denoted by $S_i \delta S_j$. δ is a transitive relation. All of the data dependencies can be represented by a data dependence graph G_1 of s nodes for S_i, $1 \le i \le s$. For each $S_i \delta S_j$ there is an arc from S_i to S_j. Statement S_j is indirectly data dependent on S_i, denoted $S_i \Delta S_j$, if there exist statements $S_{k_1},\ \ldots,\ S_{k_m}$ such that $S_i \delta S_{k_1}\ \delta \ldots\ S_{k_m} \delta S_j$. Practical details on determining if $S_i \delta S_j$ can be found in [16].

Our definition of data dependence is much more delicate than the usual definitions [21,22]. These definitions include the condition $OUT(S_i)$ $IN(S_j) \neq \phi$, i.e., they ignore subscripts and only check variable names. Thus statements like S_i: $A(I) = A(I+1) + B$ are said to be data dependent $(S_i \delta S_i)$. However, by Definition 6 we would not say $S_i \delta S_i$ because the values of $A(I+1)$ are not those from $A(I)$.

In terms of Definitions 5 and 6, we can further classify loops as follows.

Definition 7

We use D for data dependence relation, to denote the set of loops with at least one $S_i \delta S_j$, $1 \le i,j \le s$. In other words, there is at least one E_k, $1 \le k \le n$, which is a function of x_{k-m_k}, for $m_k > 0$. If $L \epsilon D$ and none of its S_i is a nonlinear function of x_j, $1 \le j \le s$, we call it a linear dependence and write $L \epsilon LD$ $(LD \subseteq D)$. The complement of D is denoted by $\overline{D}$, for non-dependence relation.

Definition 6 can be applied to any (d-u+1), $1 \le u \le d$, innermost nest of L as it is also a loop. This is described below.

Definition 8

Let L^u be the (d-u+1) innermost nest of L, $1 \le u \le d$, i.e.,

$$L = (I_1,I_2,\ldots,I_d)(S_1,S_2,\ldots,S_s)$$
$$= (I_1,I_2,\ldots,I_{u-1})(I_u,I_{u+1},\ldots,I_d)(S_1,S_2,\ldots,S_s)$$
$$= (I_1,I_2,\ldots,I_{u-1})(L^u).$$

Then for fixed values of $I_1,\ I_2,\ \ldots,\ I_{u-1}$, we can obtain all pairs of data dependence for L^u according to Definition 7 (note that now $k_1 = \ell_1,\ \ldots,\ k_{u-1} = \ell_{u-1}$), which defines graph G_u.

Example 1   Given a loop

```
L:  DO  S2  I1 = 1, 10
        DO  S2  I2 = 1, 10
            DO  S2  I3 = 1, 10
S1:  A(I1,I2,I3) = B(I1-1,I2,I3)*C(I1,I2)+D*E
S2:  B(I1,I2,I3) = A(I1,I2-1,I3)*F(I2,I3),
```

The corresponding data dependence graphs G_1, G_2, and G_3 are

$$G_1: \begin{tikzcd} S_1 \arrow[d] \\ S_2 \arrow[u, bend left] \end{tikzcd} \qquad G_2: \begin{tikzcd} S_1 \arrow[d] \\ S_2 \end{tikzcd} \qquad G_3: \begin{tikzcd} S_1 \\ S_2 \end{tikzcd}$$

For the set of data dependent loops, we can easily distinguish two cases: acyclic and cyclic graphs. Formally, we define these as follows:

Definition 9

An acyclic dependence graph is a dependence graph of s nodes, S_i for $1 \leq i \leq s$, with no pair (S_i, S_j) such that $S_i \Delta S_j$ and $S_j \Delta S_i$. A dependence graph which is not acyclic will be called cyclic.

Given a data dependence graph, we wish to partition it into blocks that contain only one statement or a cyclic dependence graph. Formally, we define these as follows:

Definition 10

On each dependence graph, G_u, $1 \leq u \leq d$, for a given loop L, we define a node partition π_u of $\{S_1, S_2, \ldots, S_s\}$ in such a way that S_k and S_ℓ are in the same subset if and only if $S_k \Delta S_\ell$ and $S_\ell \Delta S_k$. On the partition $\pi_u = \{\pi_{u1}, \pi_{u2}, \ldots\}$ for $1 \leq u \leq d$, define a partial ordering relation α in such a way that $\pi_{ui}\ \alpha\ \pi_{ui}$ (reflexive), and for $i \neq j$, $\pi_{ui}\ \alpha\ \pi_{uj}$ iff there is an arc in G_u from some element of π_{ui} to some element of π_{uj}. The α relation is also anti-symmetric and transitive. The π_{ui} are called π-blocks.

Wave-Front Method

If there are cyclic dependencies in a DO loop, we may turn to our next method, the wave-front method. This is a well-known method which effectively extracts array operations from the loop and we can then apply the above bounds to these. If the maximum speedup given by the wave-front method is insufficient, i.e., if the available processors are not all being used, we may turn to the recurrence method which gives the fastest known speedup for such problems.

Example 2

```
L2:  DO  10  I = 1, N
     DO  10  J = 1, N
10   W(I,J) = A(I-1,J) * W(I-1,J) + B(I,J-1)
          * W(I,J-1)
```

For one or more assignment statements containing cyclic dependencies, the wave-front method yields moderate speedups with high efficiency. The idea of this method can be illustrated by the loop L2 of Example 2 in which statement 10 has a cyclic dependence in that the LHS depends on RHS values computed earlier in the loop. Note that generally, one or more statements may form a cyclic dependence. This method proceeds as follows: if W(1,1) is computed from boundary values, then we can compute W(2,1) and W(1,2) in terms of W(1,1) and boundary values. Next we can compute W(3,1), W(2,2) and W(1,3) and so on, as a wave-front passes through the W array at a 45° angle. Thus we can compute this loop in $O(N)$ steps instead of the $O(N^2)$ serial steps required. The wave-front method was first described in detail by Muraoka in [7] and was later used in [15] and was also implemented in [23]. The formalization below removes some of the restrictions included in the original formulation.

In [26] a revised wave-front algorithm is presented. This includes a method of determining the angle α at which the wave-front passes through the array. It also includes a method for computing the speedup as a function of α. Note that these ideas can be extended to arrays of hieher dimension, as well. However, the wave-front method is of no value in one-dimensional arrays, since it degenerates to a serial computation in this case. A similar thing happens if α is slightly greater than 0° or slightly less than 90°. In such cases we may treat the cyclic dependence as a linear recurrence (assuming it is linear).

Loop Speedup Hierarchy

With the above fundamentals, it is possible to give some easy bounds on overall loop speedup in terms of the uniprocessor time T_1. We will present a simple hierarchy here based on the maximum known speedups for various classes of programs. Sharper bounds will be presented later in the paper, based on more detailed loop parameters. The hierarchy of this section will provide good intuition for the following sections.

The simplest loop is $L\epsilon\overline{D}$ which by Definition 7 has no dependence relation between any pair of statements. Thus, following the notation of Definition 5, all $x_i = E_i$, $1 \leq i \leq n$, can be computed in parallel. The following loop, which performs matrix addition and scalar product has this property.

```
      DO  S2 I1 = 1, 10, 2
      DO  S2 I2 = 1, 10, 1
S1:   G(I1,I2) = A(I1,I2) + B(I1, I2)
S2:   Z(I1,I2) = C(I1,I2) * D(I1,I2)
```

The total time required by any $L\epsilon\overline{D}$ is, by Theorems 1 and 2, $T_p \leq O(\log e)$ where e is the maximum number of atoms in E_i, $1 \leq i \leq n$. Hence, we have for $L\epsilon\overline{D}$

$$S_p \geq \frac{T_1}{O(\log e)} = O(T_1).$$

Now, let us study a slightly more complicated loop $L\epsilon LD$ such as one that performs vector inner-product

```
      DO  5  I = 1, 10
   5  T = T + A(I)*B(I) .
```

For any LεLD, if we pre-compute simultaneously all subexpressions in E_i, $1 \le i \le n$, which do not depend on any computed value in the loop, i.e., any x_i for $1 \le i \le n$, then the resultant statements $x_i = E_i'$, $1 \le i \le n$, can be treated as an R<n,m> system where m<n is the maximum of m_i (see Definition 7) for all i. The total computation time of any LεLD with m<<n, or m independent of n, is therefore, any preprocessing time needed to obtain the coefficients in R<n,m>, which is 0(log e) time steps by Theorems 1 and 2, plus the time to solve an R<n,m> system which is stated in Theorem 3. Since $n \le T_1 \le ne$, we have a speedup for this subset of LD,

$$S_p \ge \frac{T_1}{0(\log m \log n) + 0(\log e)} = 0(\frac{T_1}{\log T_1}) .$$

Next, consider the subset of loops which has m $\simeq$ n or m a function of n. For example, given an upper triangular matrix A, to solve Ax = b by the traditional back-substitution method we may write a loop like

```
      DO  5  I = 10, 1, -1
      X(I) = B(I)/A(I,I)
      DO  5  J = I + 1, 10, 1
   5  X(I) = X(I) - (A(I,J)/A(I,I))*X(J).
```

In this example, if we preprocess B(I)/A(I,I) for all I, and A(I,J)/A(I,I) for all I,J, we obtain an R<n,n> system. Since m = n, this is the worst-case loop of LD. Hence, we can say that the computation time of any LεLD is less than 0(log e) plus the time stated in Theorem 3, i.e., for any LεLD

$$S_p \ge \frac{T_1}{0(\log^2 n) + 0(\log e)} = 0(\frac{T_1}{\log^2 T_1}) .$$

Finally, we study a simple looking, but more complicated loop:

```
      DO  5  I = 1, 10
   5  X(I) = (X(I-1) + A/X(I-1))/2 .
```

This is a familiar iterative program for approximating $\sqrt{A}$. For this loop, LεD but L$\not\varepsilon$LD. Muraoka [7] shows that by using statement-substitution any loop with E_i being a d-th degree polynomial of x_{i-1}, d > 1, can be speeded up at most by a constant factor. Later, Kung also studied this problem [24] in a similar way. However, since we have been able to linearize a number of nonlinear recurrences, it remains an open question which techniques besides statement-substitution may be used to speed up such loops.

Summarizing the above, we are able to classify all loops in terms of their best known speedups over serial computation time T_1, i.e.,

$$S_p = \frac{T_1}{\alpha_i(\log T_1)^i} \qquad \text{for } 0 \le i \le 2 \qquad (6)$$

We call a loop Type i, $0 \le i \le 2$, if its maximum speedup has the form of Equation 6, or Type 3 if its maximum speedup is of a lower order of magnitude. This was also discussed in [2].

By the wave-front method we are at best able to achieve $T_p = 0(T_1^{1/2})$, with $T_p = 0(T_1)$ in the worst case. Thus we have $S_p \le 0(T_1^{1/2})$. Since the wave-front method's speedup is always inferior to the recurrence method for such problems, this is consistent with our claim that Equation 6 represents a maximum speedup hierarchy.

Loop Distribution

Now we turn to the question of compiling array operations from serial programs. In order to achieve statement independence we use statement-substitution. This yields increased speedup, sometimes at the cost of redundant operations. It should be used with discretion, and only in machines with a high degree of parallelism. After describing this we give our loop distribution algorithm and an example. This is a key algorithm in loop compilation.

For acyclic graphs, it is easy to demonstrate that we can perform statement-substitution between any pair of nodes which have a dependence relation. As in a BAS (c.f. Definition 3), we substitute for each LHS variable of S_i on the RHS of S_j, which is the cause of a dependence relation, the corresponding arithmetic expression on the RHS of S_i with all subscript expressions properly shifted. By applying statement-substitution, the dependence relation is removed and a set of independent assignment statements results. Each of these represents a vector assignment statement, all of which can be executed simultaneously. Theorems 2 and 3 can be used to bound the time and processors.

In loops with acyclic graphs, it is possible to reduce the graph for the entire loop to a set of independent nodes representing simultaneously executable array statements. However, in general, we must deal with cyclic graphs containing several interdependent nodes. We will now present our loop distribution algorithm which will be useful in handling these cases. By loop distribution we mean the distribution of the loop control statements over individual or collections of assignment statements contained in the loop. The idea of loop distribution was introduced by Muraoka [7], and later was implemented in our Fortran program analyzer to measure potential parallelism in ordinary programs [15], [25].

The purpose of distributing a given Type i loop is to obtain a set of smaller size loops of Type j, $0 \leq j \leq i$, which upon execution give results equivalent to the original loop. This is essentially to reduce α_i in Equation 6 (and hence increase speedup) as much as possible. In fact, the loop distribution algorithm resembling the distribution algorithm for the reduction of tree height of an arithmetic expression, may introduce more parallelism into a program loop than that obtained from an undistributed one. We now give the algorithm to accomplish this distribution as presented in [26].

Loop Distribution Algorithm

Step 1 Given a loop

$$L = (I_1, I_2, \ldots, I_d)(S_1, S_2, \ldots, S_s),$$

by analyzing subscript expressions and indexing patterns, construct a dependence graph G_u (c.f. Definitions 6 and 8) for $1 \leq u \leq d$.

Step 2 On G_u, $1 \leq u \leq d$, establish a node partition π_u as in Definition 10.

Step 3 On the partition π_u, $1 \leq u \leq d$, establish a partial ordering relation as in Definition 10.

Step 4 Let the (d-u+1) innermost nest of L be L^u, $1 \leq u \leq d$, i.e.,

$$L = (I_1, I_2, \ldots, I_d)(S_1, S_2, \ldots, S_s)$$
$$= (I_1, I_2, \ldots, I_{u-1})\{(I_u, I_{u+1}, \ldots, I_d)(S_1, S_2, \ldots, S_s)\}$$
$$= (I_1, I_2, \ldots, I_{u-1})(L^u) .$$

Replace L^u according to π_u with a set of loops $\{(I)(\pi_{u1}), (I)(\pi_{u2}), \ldots\}$ where $(I) = (I_u, I_{u+1}, \ldots, I_d)$.

The condition of the partial ordering relation γ insures that data are updated before being used. Hence, any execution order of the set of loops which replaces L^u will be valid as long as this relation is not violated. Thus, for fixed values of I_1, I_2, ..., I_{u-1}, if $\pi_{ui} \; \gamma \; \pi_{uj}$ then loop $(I)(\pi_{ui})$ must be evaluated before $(I)(\pi_{uj})$, otherwise they may be computed in parallel. In general, we can also use statement substitution to remove this relation between some or all of the distributed loops. But, by not allowing statement substitution we have a somewhat simpler compiler technique; one which generally requires fewer processors and yields less speedup.

As an example of the use of our loop distribution, consider the following pseudo-FORTRAN program.

Example 3

```
          DO  10  I = 1, N
S1:           A(I) = B(I) * C(I)
              DO  20  J = 1, N
S2:                D(J) = A(I-3) + E(J-1)
S3:           20   E(J) = D(J-1) + F
              DO  30  K = 1, N
S4:           30   G(K) = H(I-5) + 1
S5:       10  H(I) = SQRT(A(I-2))
```

Following step 1 of the Loop Distribution Algorithm, we obtain a dependence graph as shown in Figure 5. We use brackets to denote loop nesting. For simplicity and speedup in this program, we only consider the case u = 1.

In step 2, we form the partition $\pi_1 = \{\pi_{11}, \pi_{12}, \pi_{13}, \pi_{14}\}$ where $\pi_{11} = \{S_1\}$, $\pi_{12} = \{S_2, S_3\}$, $\pi_{13} = \{S_4\}$, and $\pi_{14} = \{S_5\}$. These partitions are partially ordered on step 3 as follows: $\pi_{11} \; \alpha \; \pi_{21}$, $\pi_{11} \; \alpha \; \pi_{14}$ and $\pi_{14} \; \alpha \; \pi_{13}$. Since we are considering only the case u = 1 here, we ignore step 4.

The result of this transformation is shown in Figure 6. We could use this graph to compile array operations as follows. First, S_1 yields a vector multiply. Next, we can execute π_{12} or π_{14}. π_{12} leads to a linear recurrence of the form R<N,3> which can be solved by the method of Theorem 3, by combining the D and E arrays as an unknown vector in which x_1 represents D(1), x_2 represents E(1), x_3 represents D(2), x_4 represents E(2), etc. π_{14} leads to the execution of S_5 as a vector of square roots. Finally, S_4 may be executed for all I and K simultaneously. Note that this requires the broadcasting of elements of the H array to all elements in the columns of G.

Original G_1

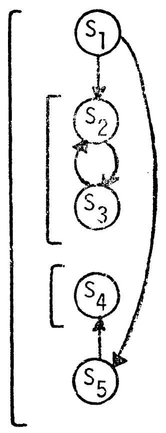


Figure 5

Distributed G_1

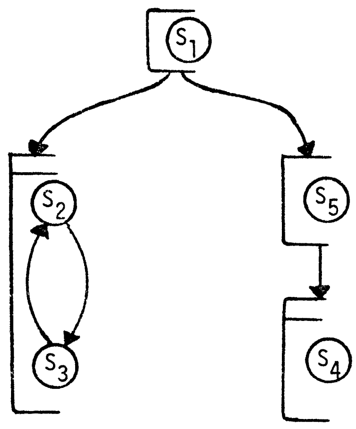


Figure 6

Here, the time required to execute π_{11}, π_{13}, and π_{14} is independent of N using O(N) processors. The overall execution time is dominated by π_{12} and is O(log N), so this is a type 1 loop. The number of processors required to achieve this time is O(N).

Notice that in this example we avoided statement substitution. Using statement substitution, we would have been able to obtain four π-blocks, all of which could be executed at once. This would require the execution of several different operations at one time, while the technique we used allows all operations at each step to be identical. Furthermore, very little additional speedup would be possible by this method since π_{12} dominates the time here.

IFs in Loops

To this point we have considered DO loops without conditional statements. The addition to DO loops of IF and GOTO as well as computed GOTO statements, can cause major problems. In particular, data dependencies can be changed at execution time by the existence of such conditional statements. Thus, knowledge at compile time of what can be executed in parallel may be difficult to obtain. In the worst case, we may be forced by not knowing about control flow, to compile loops for serial execution which in fact can be executed in a highly parallel way.

We have developed a good deal of background theory and have extended the loop distribution algorithm to handle DO loops containing IF, GOTO and computed GOTO statements [26], [27]. The material is somewhat lengthy, so we will not explore it here in detail. Rather, we will present a few examples to give an intuitive idea of how the procedures work. Basically, the procedures allow us to remove IFs from DO loops. Thus, we can compile code which is executed on arrays in an unconditional way.

An array computer is assumed to have a set of mode bits to indicate which array elements are to be operated on. The conditional testing is moved out of the program and into the data. Thus, our goal is to replace IFs in serial programs with mode bit vectors in array programs.

The easiest case to handle is the IF which depends on variables not set inside the loop. We call these type A IFs. Such IFs can be removed from loops trivially. However, good programmers almost never write such statements, so this is a moot point.

The next case is of the form

```
   DO  5  I = 1, N
   IF(I≤5) THEN A(I) = B(I) + C(I)
   ELSE A(I) = B(I)/C(I)
5  CONTINUE
```

Let $M_i[a,b]$ be a vector of mode bits denoting vector elements from a to b, inclusive. Then we can compile

```
    M1 = [1,5]
    M2 = [6,N]
DO  SIM{A(M1) = B(M1) + C(M1),
    A(M2) = B(M2)/C(M2)}
```

where DO SIM indicates that the bracketed statement can be executed as array statements and simultaneously. This is an example of what we call a prefix type B IF.

We also define two other types of IFs. A postfix type B IF requires that both sides of the IF be computed for the full array. We then give a fast method to make parallel tests and merge the two sets of data to obtain a correct result. We obtain the same speedups as in the prefix type B, but with lower efficiency. Finally, the type C IF is handled serially.

As a test of the usefulness of our methods, we have analyzed the 16 Fortran programs which appeared in 1973 in the CACM Algorithms Section. Nested DOs were counted as one loop at the outermost level. There were a total of 124 such DO loops. Each loop was characterized in terms of the worst recurrence type (c.f. Equation 6) and worst IF type it contained.

We observe that type A and prefix B IFs together with recurrence types 0 and 1 can be handled with good speedup and efficiency. This accounts for 85% of the loops. Four programs (type 3 and type C) are disasters for our methods and in part must be handled serially. The remaining programs can be handled by the postfix and wave-front (type 2) methods. Overall, this seems to imply that the methods given would be very effective for the general mix of CACM Fortran algorithms.

		Recurrence Type			
		0	1	2	3
	No IF	51	24	2	1
	A	0	0	0	0
	Prefix B	24	6	1	1
	Postfix B	3	8	1	0
IF Type	C	0	1	1	0

Table 1. 1973 CACM DO Loop Summary

Algorithm Analysis

We have also used the techniques of this section and section II to analyze several standard numerical algorithms abstractly, rather than in the form of programs. In particular, we have studied the solution of linear systems, the computation of tridiagonal matrix eigenvalues, and the solution of Poisson's equation.

Tridiagonal linear systems were studied in [28] for the special case where pivoting is not required. In [29], we improve this result slightly and also give a more stable method for solving any nonsingular tridiagonal system using Givens' transformations. This solves the system in $T = O(\log n)$ with $P = O(n)$. Furthermore, an orthogonal factorization method is given which solves any nonsingular linear system in $T_P = O(n)$ using $P = O(n^2)$ so $E_P = O(1)$. Attempts to go faster have all led to nonlinear recurrences which we have not been able to linearize.

In [30] and [31], the parallel computation of matrix eigenvalues is studied. In [31] we treat tridiagonal matrices and we show a parallel QR-algorithm which requires $O(\log n)$ steps and uses $O(n)$ processors per iteration. This yields a speedup of $O(n/\log n)$ and an efficiency of $O(1/\log n)$.

The Poisson equation is treated in [32] and [18]. In [18], it is proved that on an nxn grid, a finite difference approximation can be solved by a direct method in $T_P = O(\log n)$ with $P = O(n^2)$ so $E_P = O(1)$, for Dirichlet, Neumann, and periodic boundary conditions. We also treat the biharmonic equation.

Extensions

The results discussed in this section can be interpreted in several ways besides those already mentioned. The π blocks produced by these techniques could be used as tasks for separate processors in a multiprocessor computer. In virtual memory machines, the π blocks could be used as pages, with the α transitions representing page faults. Preliminary studies of space, time products with limited page allotments are quite encouraging. Substantial improvements in address localization have been obtained using this technique as compared to standard compilation techniques. Finally, the transformation of loops containing IFs can be thought of as transformation of serial programs to DO WHILE loops. In [27] we discuss further the interpretation of these results in a structured programming sense.

IV. Processors

Traditionally, computers have executed one operation at a time. Obviously, program speedups may be achieved by performing more than one operation simultaneously. This idea seems to be at least 130 years old, since we read in the October 1842 publication of Menabrea's description of Babbage's lectures [33]: "...when a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes." It should be pointed out that elsewhere (e.g., [33, p. 261]) remarks can be found indicating that at most one pair of operands are used at once. In any case, the idea seems to have been clear at the time, even if it was not part of the design.

In modern history, the Bell Telephone Laboratories' Model V system built by Stibitz and Williams in the late 1940s had two processors

[34]. Designs of machines capable of simultaneously executing several operations for solving partial differential equations began to appear in the 1950s. In 1952, Leondes and Rubinoff [35] proposed a multi-operation processor oriented around a drum memory. In 1958, the German computer pioneer Zuse proposed a drum-oriented parallel machine [36]. Subsequently, a number of abstract and real computers have been proposed and built that are capable of executing more than one operation at once. Most manufacturers now provide two processor general multiprocessor systems and some have up to four processors which operate from a shared memory. Others provide several arithmetic units in one processor. Pipeline and parallel systems currently appearing promise to make even more simultaneous operations available for special problem classes.

Since the early 1960s, we have seen a sequence of high speed machines which have some kind of multioperation capability. The CDC 6600 of the 60s was succeeded by the 7600 in the early 70s. IBM introduced the 360/91 and its successors. The 7600 and /91 are both pipelined machines and achieve high performance by operating on arrays of data. Their instruction sets are rather traditional, however. In contrast, the pipelined Control Data STAR [37] and Texas Instruments' ASC [38], both have vector instruction sets, which makes compilation for them substantially easier.

On the other hand, the Burroughs' ILLIAC IV [66] is a parallel array machine, but its instruction set is also traditional in nature. Vectors must be broken up into partitions of size 64 and loops performed over a sequence of such partitions. The Goodyear Aerospace STARAN IV [39] is a parallel array of processors, each of which operates in a bit-serial fashion. This is an example of an associative processor.

It is interesting to note that the highest speed pipeline processors, the STAR and ASC, both resort to parallelism by providing several parallel pipelines to achieve their desired operating speeds.

The processing speedups achieved by all of these machines are due to parallelism between operations as well as parallelism between memory and processor activities. We shall discuss memories, alignment networks, and control units later. Our point here is that in order to compile ordinary serial languages for these processors, two things are desirable: 1) Powerful translation techniques to detect parallelism, and 2) Array type machine languages.

The main contributions to program speedup discussed in section III arise from our loop distribution procedure. This leads to array operations and recurrences. Both of these are well suited for computation on machines which must perform the same operation on many data elements to achieve high performance. Thus, the methods of section III could serve as compiler algorithms for such machines.

Some time ago, we implemented a comprehensive analyzer of Fortran programs. It used algorithms like those of sections II and III, although some of the techniques were much more primitive than those discussed here. Details of our algorithms and results may be found in [7], [15], [25], and [2]. We will summarize a few points very briefly.

Altogether some 140 ordinary Fortran programs, gathered from many sources, were analyzed. The programs ranged from numerical computations on two-dimensional arrays (e.g., EISPACK) to essentially nonnumerical programs (e.g., Fortran equivalents of GPSS blocks). We set all loops to 10 or fewer iterations and analyzed all paths through the programs, computing T_1, T_p, S_p, E_p, etc. These were averaged over all traces and also over collections of programs. A plot of our results for S_p vs. p is shown in Figure 7. Some of the points are labelled with the names of a collection of programs; ALL represents a global average.

Thus we observe that for these simple programs, about 35 parallel processors could deliver a speedup of about 10 for an efficiency of greater than 30%.

Furthermore, we took a subset of the programs, again a random cross-section, and varied the DO loop limits from 10 to 40. The points 10, 20, 30, 40 correspond to the results. We conclude that for our sample of ordinary Fortran programs, speedup is a linear function of T_1 and hence p. This is quite different from some of the folklore which has arisen about parallel computation [40], [41], [42] (e.g., $S_p = O(\log p)$). For more discussion of this, see [2], [25]. Using the methods known now, we are implementing a new analyzer and expect better results.

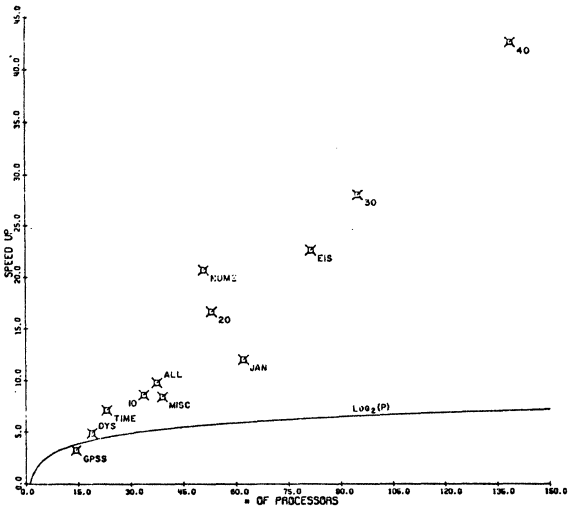


Figure 7. Speedup vs. Processors

Combinational Logic Design

We are also interested in the logic design of the processors themselves. Parallelism at the bit level as compared to the word level has been under consideration for many years in arithmetic unit design. Indeed, Babbage was greatly concerned with anticipatory carry-adder schemes [33] and was quite proud of his invention of a fast adder.

More recently, lower bounds were given on the time required for addition [43] and multiplication [44]. It was shown that the bounds could be met, but non-standard arithmetic was used. Binary addition was considered in [45]. These papers did not consider such questions as gate (or component) counts, fan-out considerations, or more general logic design problems.

In [46] these matters are discussed in some detail and we will briefly survey them below. We emphasize that each bound given is proved by a constructive algorithm. These algorithms can be used to transform any linear sequential logic into combinational logic. Since it is usually quite easy to write down sequential (bit serial) equations for a logical function, this seems to be a useful way to design fast parallel combinational circuits.

The theoretical results of section II were given at the level of operations on computer words. But, it is obvious that operations on bits are quite similar. The main difference between bit level and word level operations is in the fanning out of intermediate values. In section II we assumed that a number could be broadcast for operations with several other numbers without paying a gate or time penalty. This is reasonable because the actual time and gate count for broadcasting is negligible compared to arithmetic times.

However, at the gate level, add and multiply are interpreted as or and and, respectively. Thus, the fan-in of n bits through combinational logic may be done using n - 1 two input gates, whereas, fanning out a bit to n destinations also requires n - 1 two output gates. Thus, the fan-out gates may become nonnegligible.

In the following, we ignore the gates which are the source of signals since they are counted as the destination of some other signal. First, we give a lemma about signal fan-out.

Lemma 3 [46] An e way fan-out can be accomplished using gates with fan-out of $f \geq 2$ in

$$T_G \leq \lceil \log_f e \rceil - 1$$

with

$$G < \frac{e-1}{f-1} .$$

Next, we bound the gates and time in the combinational part of any logic circuit.

Lemma 4 [5], [47] Any Boolean expression E<e> of e atoms can be realized using gates of fan-in 2 in

$$T_G[E<e>] \leq \begin{cases} 1 + 2d + \lceil \log e \rceil & \text{if } d < \frac{3}{2}\log e \\ \lceil 4\log e \rceil & \text{otherwise,} \end{cases}$$

with

$$G[E<e>] \leq \begin{cases} e-1 & \text{if } d < \frac{3}{2}\log e \\ 2(e-1) & \text{otherwise,} \end{cases}$$

where d is the depth of parenthesis nesting in E. In the following, we assume for simplicity that n and m are powers of 2. A third useful result for recurrences is

Lemma 5 [46] Any m-th order linear Boolean recurrence R<n,m> can be solved using gates of fan-in 2 and fan-out $f = 2^q$, $q \geq 1$, in

$$T_G \leq (\frac{5}{2} + \log m + \frac{1}{2}\log_f n)\log n - \frac{1}{2}(\log^2 m + \log m)$$

with

$$G \leq \frac{1}{2}[m^2(2 + \frac{1}{f-1}) + m(1 + \frac{1}{f-1})]\, n \log n + O(m^3 n) .$$

Counting gates and gate delays is a useful measure, even though most present logic design is carried out in terms of integrated circuits, since gates are an absolute measure. But we are also interested in specifying the role of integrated circuits in solving recurrences. First, we define two integrated circuit types. Since the vast majority of real logic design problems are R<n,1> systems, we now restrict our attention to the case m = 1.

Definition 11

We define two types of integrated circuit packages.

a) $IC_{R<n,1>}$ is a package which accepts input atoms c_i for $1 \leq i \leq n$, and a_i for $2 \leq i \leq n$. It computes the outputs x_i for $1 \leq i \leq n$ according to the recurrence relation $x_i = c_i + a_i x_{i-1}$, where $x_0 = 0$.

b) $IC_{U<n>}$ is a package which may accept input atoms a_i and b_i for $1 \leq i \leq n$, and c and d. It computes the outputs x_i for $1 \leq i \leq n$, according to

$$x_i = v_i w_i + y_i z_i ,$$

where either

i) $v_i = a_i$, $w_i = c$, $y_i = b_i$ and $z_i = d$,

$1 \leq i \leq n$

or

ii) $v_i = a_i$, $w_i = b_i$, $y_i = \bar{a}_i$ and $z_i = \bar{b}_i$,

$1 \leq i \leq n$.

In general, we denote the total number of integrated circuits in some logical circuit by IC. Now we can prove the following.

Lemma 6 [46] Any first-order linear recurrence R<n,1> can be solved in time

$$T_{IC} \leq (2 \frac{\log n}{\log h} - 1)$$

using a total package count of

$$IC \leq 6 \frac{n}{h} + 4 \frac{\log n}{\log h} - 7$$

with package types $IC_{R<h,1>}$ and $IC_{U<h-1>}$ for $h \geq 2$.

Using the above, we can solve a number of useful logic design problems. To illustrate the method, consider the binary addition of two numbers $a = a_n \ldots a_1$ and $b = b_n \ldots b_1$. We can generate the sum digits $s = s_n \ldots s_1$ and carry digit c_n as follows:

$$s_i = (a_i b_i + \bar{a}_i \bar{b}_i)\ c_{i-1} + (\bar{a}_i b_i + a_i \bar{b}_i)\ \bar{c}_{i-1}$$

where $1 \leq i \leq n$ and c_i is specified in Equation 5.

By the use of Lemma 4 and Lemma 5, we can easily prove the following.

Theorem 4 [46] Two $n = 2^t$, $t \geq 0$, digit binary numbers can be added in

$$T_G \leq \frac{1}{2}(5 + \log_f n) \log n + 4$$

with

$$G \leq (\frac{3}{2} + \frac{1}{f-1})\ n \log n + (8 - \frac{1}{f-1})\ n + (\frac{2}{f-1}) \log n + 2 .$$

Furthermore, using Lemma 6 we can prove

Theorem 5 [46] Two $n = 2^t$, $t \geq 0$, digit binary numbers can be added in time

$$T_{IC} \leq (2 \frac{\log n}{\log h} + 1)$$

using a total package count of

$$IC \leq 9\frac{n}{h} + 4 \frac{\log n}{\log h} - 7$$

with package types $IC_{R<h,1>}$ and $IC_{U<h>}$ for $h \geq 2$.

Using these results, we consider a practical example.

Example 4 Consider the problem of adding two 32-bit binary numbers using gates with fan-in 2 and fan-out 8. By the method of Theorem 4, the sum can be formed in at most 21 gate delays since

$$T_G \leq \frac{1}{2}(5 + \log_8 32) \log 32 + 4 < \frac{1}{2}(\frac{20}{3})\ 5 + 4 = \frac{100}{6} + 4 < 21 .$$

The number of gates required is at most

$$G \leq (\frac{3}{2} + \frac{1}{7})\ 32 \cdot 5 + (8 - \frac{1}{7})\ 32 + 2 \cdot 5 + 2 < \frac{23}{14} \cdot 160 + \frac{55}{7} \cdot 32 + 12 = 527$$

On the other hand, if integrated circuit packages are available which handle 8 bits at a time, h = 8, we have the following. The total package count is

$$IC \leq 9 \frac{32}{8} + 4 \frac{\log 32}{\log 8} - 7 < 37$$

and the number of package delays is

$$T_{IC} \leq (2 \frac{\log 32}{\log 8} + 1) < 5 .$$

These results can be used to transform any linear sequential logic to combinational logic. Furthermore, [46] shows how certain nonlinear cases can be handled as well. Thus, we have a uniform way to design circuits which perform such functions as binary addition, counting the number of ones to the right of each position in a binary word, binary multiplication, digital filtering, and so on.

Control Units

A well-designed control unit is one which never gets in the way of the processor(s) and memories. In other words, it operates fast enough to be able to supply instructions whenever they are needed in the processing and moving of data. Control units tend to become complex, mainly in a timing sense, because they may have a number of tasks to control.

One way to ease some control unit difficulties is to use parallelism at the control unit level. A multiprocessor is an example; several complete control units are used. This may be rather expensive, and the multifunction, pipeline and parallel processor machines use one shared control unit. Such control units often contain a number of independently operating parts. For example, the first use of pipelining was in control units [69]. A detailed study of the control unit of any high-speed computer will reveal a number of simultaneously operating, independent functions. While this may allow the functions to operate more slowly, it also causes some sychronization problems.

As we mentioned in our processor discussion, the level of machine language is very important in modern, high-speed computers. Vector instruction sets make compiler writing easier. They also focus control unit design on the correct questions, namely, to execute vector functions at high speed.

Control units for high-speed computers must handle the traditional functions, including instruction decoding and sequencing, I/O and interrupt handling, and address mapping and memory indexing. In addition, we can list several new functions. For one, memory indexing becomes somewhat more complex when whole arrays are being accessed in large parallel memories. Also, array computers (parallel or pipeline) often rely on the control unit for scalar computations. Broadcasting of scalars to an array must also be handled. Special control features such as IF tree processing [48] can be effectively handled in the control unit. Special stacks and queues may be required to handle a number of processors and programs in rapid succession. Indeed, instruction level multiprogramming may even be attempted.

Rather than discuss any of these in detail, we simply refer the reader to a detailed study of the several high-speed machine papers mentioned earlier.

We conclude this section with the computer organization of Figure 8. The control unit can really be regarded as four control units, one for each of the four other major subsystems shown. The operation of this machine can be regarded as a pipeline from memory to memory. For move instructions (memory to memory) the processors can be bypassed.

Figure 8 represents parallelism at a number of levels: within the control unit, processors, memories and data alignment networks. Also, it contains parallelism in the simultaneity of operation of each of these which forms a pipeline. Note that pipelining can also be used within each of the five major subsystems to match bandwidths between them.

The details of accessing parallel memories and of aligning the accessed data will be discussed in detail in the next section.

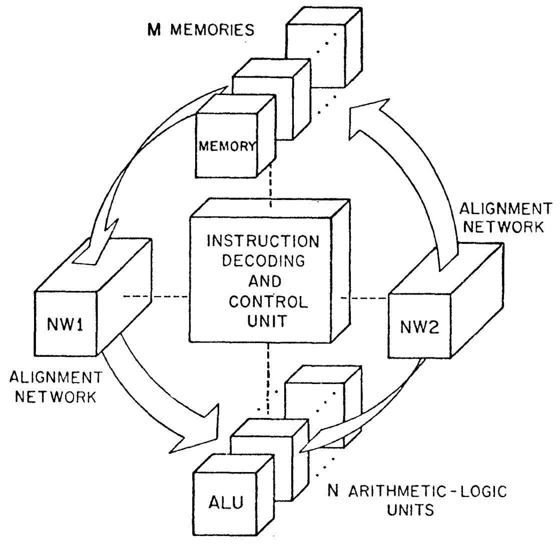


Figure 8. Overall Machine Organization

V. Parallel Memory Access and Data Alignment

As effective speeds of processing units have increased, memory speeds have been forced to keep up. This has partly been achieved by new technologies (magnetic cores to semiconductors). But technology has not been enough, as evidenced by the fact that in 1953, the first core memory operating (in Whirlwind I) had an 8 μs memory cycle time. Today, most computer designers cannot afford to use memories much faster than several hundred nanoseconds. Certainly, two orders of magnitude increase in memory speed is an upper bound, over the past twenty odd years.

In the same period, the fastest processor operation times have advanced from a few tens of microseconds to a few tens of nanoseconds; or three orders of magnitude. Memory system speeds have kept up with processors only through the use of parallelism at the word level. In the late 1950s, ILLIAC II and the IBM STRETCH introduced the first two-way interleaved memories. At the present time, high speed computers have on the order of 100 parallel memory units. If a word can be fetched from each of m memory units at once, then the effective memory bandwidth is increased by a factor of m.

Array Access

Parallel memories are particularly important in array computers (parallel or pipeline). Thus, if a machine has m memory units we can store one-dimensional arrays across the units as shown in Figure 9, for m = 4. While the first m operands are being processed, we can fetch m more, and so on. But, if the array is indexed such that, say, only the odd elements are to be fetched, then the effective bandwidth is cut in half due to access conflicts as shown in the underlined elements of Figure 9. These conflicts can be avoided by choosing m to be a prime number. Then, any index distance relatively prime to m can be accessed without conflicts.

Many programs contain multidimensional arrays. These can lead to more difficult memory access problems, since we may want to access rows, columns, diagonals, back diagonals (as in the wave-front method of section III), square blocks, and so on. For simplicity, consider two-dimensional arrays and assume we want to access n element partitions of arrays from parallel memories with m units.

Memory Units

$\underline{a_1}$	a_2	$\underline{a_3}$	a_4
$\underline{a_5}$	a_6	$\underline{a_7}$	a_8
a_9	a_{10}	a_{11}	•
•	•		

Figure 9. One-Dimensional Array Storage

Consider the storage scheme shown in Figure 10 where m = n = 4. Clearly, using this storage scheme we can access any row or diagonal without conflict. But all the elements of a column are stored in the same memory unit so accessing a column would result in memory conflicts, i.e., we would have to cycle the memories n times to get the n elements of a column.

In order to allow access to row and column n-vectors, we can <u>skew</u> the data as shown in Figure 11 [49]. Now, however, we can no longer access diagonals without conflict. It can be shown, in fact, that there is no way to store an mxm matrix in m memories, when m is even, so that arbitrary rows, columns, and diagonals can be fetched without conflicts. However, as we shall see, by using more than m memories we can have conflict-free access to any row, column, or diagonal, as well as other useful m-vectors. First, we will generalize the idea of skewed storage.

Let m be the number of memories and let δ_i be the skewing distance in the i-th dimension. Thus for a two-dimensional (mxm) matrix, each successive element of the first dimension (column) is stored δ_1 (mod m) memories away from the previous element. Similarly for the second dimension (rows) and δ_2. This is called a (δ_1,δ_2) skewing scheme. Thus, Figure 11 shows a (1,1) skewing scheme. For m = 5, Figure 12 shows a (2,1) skewing scheme. Clearly, this generalizes to matrices with k dimensions (i.e., $(\delta_1,\delta_2,\ldots\delta_k)$ skewing) and matrices whose dimensions are larger than m.

Define a <u>d-ordered n-vector</u> (mod m) as a vector of n elements whose i-th logical element is stored in memory unit $\mu = di + c \pmod m$ where c is an arbitrary constant. A sufficient condition for conflict-free access to a d-ordered n-vector (mod m) is:

$$m \geq n \gcd(d,m) \qquad (7)$$

where gcd(d,m) is the greatest common division of d and m. This follows from the fact that the set of memory units $\{\mu | \mu = di + c (\mathrm{mod}\ m),\ 0 \leq i \leq n - 1\}$ must contain n <u>distinct</u> elements. That is, the memories in which the n elements of the d-ordered n vector are stored must be distinct.

If we use a (δ_1,δ_2) skewing scheme, then clearly columns will be δ_1-ordered, and rows will be δ_2-ordered. Similarly, diagonals will be $\delta_1 + \delta_2$ ordered. Thus, in order to access these three types of n-vectors the following conditions must hold:

$$m \geq n \gcd(\delta_1,m) \qquad \text{(columns)}$$

$$m \geq n \gcd(\delta_2,m) \qquad \text{(rows)}$$

$$m \geq n \gcd(\delta_1+\delta_2,m) \qquad \text{(diagonals)}$$

Clearly, if m = n then $\gcd(\delta_1,m)$, $\gcd(\delta_2,m)$ and $\gcd(\delta_1+\delta_2,m)$ must equal 1 if these conditions are to hold. If m is even, then δ_1, δ_2 and $\delta_1+\delta_2$ must be odd for this to hold. But it is easy to show that δ_1, δ_2 and $\delta_1+\delta_2$ cannot all be odd. Thus we cannot have conflict-free access to rows, columns, and diagonals if m = n and m is even.

If we turn to memory systems where m > n, we can obtain conflict-free access to many partitions. In addition to rows, columns and diagonals mentioned above, let us consider back diagonals which are $\delta_1 - \delta_2$ ordered. Then it is easy to show that if $m = 2^{2k} + 1$, for any integer k, and $(\delta_1,\delta_2) = (2^k,1)$, we have conflict-free access (by Condition 7) to rows, columns, diagonals and back diagonals. Square blocks can also be accessed. For an example with k = 1, $\delta_1 = 2$, and $\delta_2 = 1$, see Figure 12. This and other similar results are discussed in [50].

If m is not a power of two, certain difficulties arise in indexing the memory. Also, note that the elements of various partitions are accessed in scrambled order. The question of unscrambling the accessed elements is discussed by Swanson [51].

In order to simplify indexing and unscrambling, systems of the form m = 2n were considered by Lawrie in [52] and [60]. He shows that conflict-free access to a number of partitions is possible using such a memory. We illustrate this in Figure 13 with $m = 2n = 8$, $\delta_1 = \sqrt{n} + 1$, and $\delta_2 = 2$. We will give a general discussion of data alignment networks for such unscrambling later in this section.

Memory Unit

0	1	2	3
a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

Figure 10. Straight Storage ($m = 4$)

Memory Unit

0	1	2	3
a_{00}	a_{01}	a_{02}	a_{03}
a_{13}	a_{10}	a_{11}	a_{12}
a_{22}	a_{23}	a_{20}	a_{21}
a_{31}	a_{32}	a_{33}	a_{30}

Figure 11. Skewed Storage ($\delta_1 = \delta_2 = 1$, $m = 4$)

Memory Unit

0	1	2	3	4
a_{00}	a_{01}	a_{02}	a_{03}	
a_{13}		a_{10}	a_{11}	a_{12}
a_{21}	a_{22}	a_{23}		a_{20}
	a_{30}	a_{31}	a_{32}	a_{33}

Figure 12. Skewed Storage ($\delta_1 = 2$ and $\delta_2 = 1$ with $m = 5$)

Memory Unit

0	1	2	3	4	5	6	7
a_{00}		a_{01}		a_{02}		a_{03}	
	a_{13}		a_{10}		a_{11}		a_{12}
a_{21}		a_{22}		a_{23}		a_{20}	
	a_{30}		a_{31}		a_{32}		a_{33}

Figure 13. Skewed Storage ($\delta_1 = 3$, $\delta_2 = 2$, $m = 2n = 8$)

In order to implement a skewing scheme, we must have a properly designed parallel memory system. In particular, each of the m memory units must have an independent indexing mechanism. This allows us to access a different relative location in each memory unit. It is interesting to observe that several presently existing high speed computers have handled their parallel memories in different ways.

The Control Data STAR and the Texas Instruments' ASC do not allow independent indexing of each memory unit. Instead, they provide transpose instructions by which arrays can be physically transposed in memory to provide access to, say, rows and columns. The transpose time is essentially wasted time and some algorithms for these machines are slowed down by as much as a factor of two in this way.

ILLIAC IV has independent index registers and index adders on each of $m = 64$ memories. Since it has 64 processors, access to partitions of $n = 64$ elements is usually required. Thus the (1,1) skew scheme of Figure 11 is easily implemented. Of course, since m is even, conflict-free access to rows, columns, and diagonals is impossible. But as Figure 11 shows, diagonals may be accessed in just two memory cycles.

STARAN IV has an associative memory. The key to implementing such a memory (which was for many years their downfall) is the ability to access individual words, as from a standard memory, on some occasions, and on other occasions to access the same bit from all words. Access in the bit direction allows searches to be made and access to be based on the contents of the memory. For example, a programmer may request all words whose second byte contains all zeros.

The arrival of semiconductor memories permitted an elegant implementation of an associative memory in STARAN IV by skewing the storage of words at the bit level. Thus if we interpret the a_{ij} of Figure 11 as bits, we have a 4-word memory with 4 bits per word. STARAN IV does not use programmable index registers, but rather has special hard-wired indexing functions which are available to the programmer. From a programmer's point of view, he can address a word, a bit slice across words, a particular byte across a subset of words, and so on. Physically, the memory units are arrayed as a set of m semiconductor chips each organized as 1 bit by m words. This results in an mxm bit memory, and a complete system contains several of these.

Random Access

The execution of Fortran-like programs frequently leads to memory access requirements which include one-dimensional arrays and various partitions of multidimensional arrays, as we have been discussing. However, we sometimes face access problems which have much less regularity.

For example, consider the subscripted subscript case:

```
DO  I = 1, N
    X(I) = A(B(I)) .
```

Here, we have no idea at compile time about which elements of A are to be fetched, assuming that B was computed earlier in the program. This easily generalizes to multidimensional arrays. Frequently, table lookup problems are programmed in this way.

To deal with this kind of memory access problems, is in general to deal with random access to a parallel memory. Note that this is a problem which has been given a good deal of attention in multiprocessor systems.

One of the great advantages of multiprocessor organizations is that the relatively expensive primary and secondary memory can be shared by several relatively inexpensive processors. A primary weakness in such systems is that as the number of memories and processors grows, the effective system bandwidth drops off due to memory conflicts. In commercially available systems, four processors and perhaps eight to sixteen memories have been limiting numbers in contrast to much higher numbers of memories in array machines.

In the past ten years, a variety of analytical results have been obtained concerning parallel memories. Most of the results were presented in a rather abstract way, without any clearly stated machine interpretations. We will interpret them below and also sketch some new results.

There are two key questions on which the validity and usefulness of these models turns. They are:

1) What kind of data dependence is assumed in the memory access sequence?

2) What kind of queueing mechanism is assumed for retaining unserviced accesses?

There are several other questions which relate to the usefulness of the model, but are of less importance in determining the general form of the results. These include how control dependence is handled and whether we study the steady state or transient memory bandwidth. These are interrelated questions, and control dependence is also related to data dependence.

In these terms, we briefly summarize some of the results. Hellerman's model [53] can most reasonably be interpreted to assume no data dependence between successive memory accesses and to have no provision to queue conflicting addresses. It is also a steady state model, ignoring control dependence. Thus, it scans an infinite string of addresses, blocking when it finds the first duplicate memory unit access request.

In various models, Coffman and his coworkers [54, 55, 56] extended the above to include a type of queueing and to separate data accesses from instruction accesses. These papers further introduced address sequences which were not necessarily uniformly distributed. These models also assumed that no data dependencies existed in the address sequence.

Ravi [57] introduced a model which was more realistic for multiprocessor machines. He allows each processor to generate an address and computes the number of them which can be accessed without conflict, in a steady state sense. Effectively he assumes a sequential data dependence in the addresses generated by each processor.

In [58] the above results are extended in several ways. First, it is shown analytically that the model of [57] yields an effective memory bandwidth which is linear in the number of memory units. Several models are given with queues in the processors and in the memories, to show the differing effects on bandwidth of such queues and methods used for managing the queues. Several types of data dependencies are assumed to exist; some as in the Ravi model and others which include dependencies between the processors. In all of these models, we show that the effective bandwidth of m memories can be made to be $O(m)$. The models are useful for either multiprocessor or parallel machines.

Thus, we conclude that for parallel or multiprocessor machines, the proper use of m parallel memories can lead to effective bandwidths which are $O(m)$. This is much more encouraging than the $O(m^{1/2})$ which was derived from earlier, more naive models.

Alignment Networks

Now we turn to the question of interconnecting the processors and memories we have been examining. We shall consider various alignment networks to handle the task. In existing computers, the data alignment problem is handled in a number of different ways; we will survey these and some new possibilities.

At the bit level, data shifting is done in all computers. The necessities of aligning and normalizing floating-point numbers or packing and unpacking bytes are well known. Usually, uniform shifts in which all bits shift an equal distance can be performed in a single instruction. Various implementations are discussed in [59].

At the word level, data alignment requirements depend on the machine organization. A simple way to connect several memories to a processor is to use a shared bus. For higher speed operation, multiprocessors often use a crossbar switch which allows each processor to be connected to a different memory simultaneously. In the ILLIAC IV array, the i-th processor can pass data to processors $i \pm 1$ and $i \pm 8$, modulo 64. Here all processors must route data the same distance in a uniform way.

None of the above techniques is well suited to a high performance parallel computer. Indeed, the alignment network should be driven by an independent control unit, to operate concurrently with the processor and memory operation. The requirements include more than uniform shifts and at times even more than permutations. Often broadcasts are needed, including partial and multiple simultaneous broadcasts, e.g., $n^{1/2}$ numbers, each broadcast to $n^{1/2}$ processors for matrix multiplication in an n processor machine [60].

The alignment network should be able to transmit data from memory to memory and processor to processor as well as back and forth between memories and processors. The connections it must provide are derived from two sources. For one, it must be able to handle the indexing patterns found in existing programs, for example, the uniform shift of 5 necessary in A(I) + A(I+5). For another, it must be able to scramble and unscramble the data for memory accesses. For example, to add a row to a column, one of the partitions must be "unskewed". More details can be found in [2], [52] and [60].

Many possible solutions exist for this problem. Given n processors and n memories, we will now outline a few details of some possible alignment networks.

A crossbar switch is an obvious candidate. With it we can perform any one-to-one connection of inputs to outputs, and with some modification we can also make one-to-many connections for broadcasting. The switch can be set and data can be transmitted in O(log n) gate delays. However, the cost of such a switch is quite high, namely, $O(n^2)$ gates. Thus for large systems, a crossbar alignment network is out of the question.

Another possibility is the rearrangeable network developed over many years in telephone switching theory. It is shown by Beneš [61] that such a switch, with the same connection capabilities as a crossbar, can be implemented using only O(n log n) gates. The time required to transmit data through the network is just O(log n). Unfortunately, the best known time to set up the network for transmission is O(n log n) [62]. This control time renders the network impractical as an alignment network, unless all connection patterns could be set up at compile time.

The Batcher sorting network [63] is another possibility. Not only can it perform the connections of a crossbar switch, it can also sort its inputs, if desired. This network has $O(n \log^2 n)$ gates, so it is an improvement over the crossbar. However, it requires time of $O(\log^2 n)$ gate delays for control and data transmission, making it faster than the Benes approach.

As a final possibility, we discuss the Ω network [60] proposed specifically for this purpose. This network can be controlled and transmit data in O(log n) gate delays, but contains only O(n log n) gates. Thus it has the speed of a crossbar with the cost of a Beneš network. Its shortcoming is that it cannot perform arbitrary interconnections. However, as discussed above, we seek an alignment network which can handle the requirements posed by program subscripts and memory skewing schemes. Lawrie has examined a number of such questions and the Ω network satisfies many of them.

It is interesting to note that the Ω network consists of a sequence of identical interconnection paths called <u>shuffles</u>, see e.g., [64], [60]. We call transmission from left to right a shuffle and from right to left an <u>unshuffle</u>. It can be shown that the Beneš and Batcher networks, as well as the Ω network, can all be constructed from a series of shuffle and unshuffle interconnections of 2x2 switching elements. The switching elements are basically 2x2 crossbars. In the Batcher network, they have the further capability of comparing their inputs and switching on this basis. In the Ω network they can also broadcast either of their inputs to both outputs.

File Processing

Many computation problems are nonnumerical in nature. In a number of file processing and information retrieval applications, the requirement for merging a number of long lists arises. Consider the problem of satisfying information retrieval queries posed using logical expressions of search terms. If the data is stored in inverted files, the retrieval can be handled conveniently by first merging index files of the search terms. Then the logical connections of the search term expression can be applied to the merged list to determine which stored data to retrieve.

In [65] such a system was designed and simulated. It involves a Batcher merge network [63] which can merge two lists of n terms in O(log n) gate delays using O(n log n) gates.

This merge network is followed by a coordination network of similar complexity, which handles the logical functions mentioned above. These networks together with a buffer memory can be located between a high-speed secondary memory (head-per-track disk or shift register type) and a buffer memory which holds intermediate search results. For file processing problems which are dominated by I/O and merges, such a configuration can achieve speedups proportional to the number of parallel inputs to the merge/coordination unit.

Similar applications are being made of the STARAN IV in a number of real world applications [67]. A number of other efforts have been carried out in the area of nonnumerical processing. When nonnumerical computation is as well understood as numerical computation is now--in the sense of standard languages and algorithms--one can expect results about time and component complexity to develop as it has for numerical computation.

VI. Conclusions

We have surveyed a number of aspects of "parallel computation" in the broad sense. It is clear that by viewing matters abstractly, several diverse design problems can be handled by identical methods.

Parallelism in machine organization has been used since the time of Babbage. It will continue to be exploited, together with hardware speed improvements, to build faster computers. If hardware costs continue to improve at a faster rate than hardware speeds, parallelism will be used even more. For example, processors which use hundreds of microprocessors as components are easy to imagine. The methods outlined here show how large numbers of them could be exploited.

The goal is to show that linear speed improvements can be achieved with linear (or almost linear) increases in component count. If one defines speedup/cost as a criterion function, it can be shown [68] that by using parallel processors at the algorithm level, better cost-effectivness can be achieved than by using parallel gates at the arithmetic level as we have been doing for the past 25 years.

Compilation for parallel machines has been a stumbling block. But the methods outlined have been used to transform a number of serial programs to highly parallel form. While it is easy to write programs which cannot be speeded up, due to nonlinear recurrences or difficult conditional statements, such constructs are seldom used in real programs.

Thus, by studying the structure of programs and the structure of machines, we can attempt to design machines which are well matched to the programs they are to execute. As side benefits, we may be able to transform programs into more understandable forms for programmers (e.g., IFs in loops become DO WHILEs). Also we may be able to improve the paging behavior of a standard program on any machine with virtual memory.

References

[1] M. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol. C-21, 9 (Sept., 1972), pp. 948-960.

[2] D. Kuck, "Multioperation Machine Computational Complexity," Proceedings of Symposium on Complexity of Sequential and Parallel Numerical Algorithms, Academic Press, (1973), pp. 17-47.

[3] J. L. Baer, and D. P. Bovet, "Compilation of Arithmetic Expressions for Parallel Computations," Proc. of IFIP Congress, North-Holland, Amsterdam, (1968), pp. 340-346.

[4] J. C. Beatty, "An Axiomatic Approach to Code Optimization for Expressions," J. Assoc. Comput. Mach., 19 (1972), pp. 613-640.

[5] D. Kuck, and Y. Muraoka, "Bounds on the Parallel Evaluation of Arithmetic Expressions Using Associativity and Commutativity," Acta Informatica, Vol. 3, Fasc. 3, (1974), pp. 203-216.

[6] R. Brent, and R. Towle, "On the Time Required to Parse an Arithmetic Expression for Parallel Processing," submitted for publication.

[7] Y. Muraoka, "Parallelism Exposure and Exploitation in Programs," Ph.D. Thesis, Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Rep. 424, Feb. 1971.

[8] P. W. Kraska, "Parallelism Exploitation and Scheduling," Ph.D. Thesis, Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Rep. 344, Aug. 1969.

[9] D. Kuck, and K. Maruyama, "Time Bounds on the Parallel Evaluation of Arithmetic Expressions," SIAM Journ. of Computing, Vol. 4, 2 (June, 1975), pp. 147-162.

[10] R. Brent, "The Parallel Evaluation of General Arithmetic Expressions," J. Assoc. Comput. Mach., Vol. 21, (1974), pp. 201-206.

[11] D. E. Muller, and F. P. Preparata, "Restructuring of Arithmetic Expressions for Parallel Evaluation," Coordinated Science Lab., Univ. of Ill. at Urbana-Champaign, Rep. R-676, April, 1975.

[12] S. C. Chen, "Speedup of Iterative Programs in Multiprocessor Systems," Ph.D. Thesis, Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Rep. 694, Jan. 1975. (NSF-OCA-GJ-36936-000004).

[13] S. C. Chen, and D. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems," IEEE Trans. on Computers, Vol. C-24, 7 (July, 1975), pp. 701-717.

[14] D. E. Knuth, "An Empirical Study of FORTRAN Programs," Dept. of Comput. Sci., Stanford Univ., Rep. CS-186, 1970.

[15] D. Kuck, Y. Muraoka, and S. C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speed-Up," IEEE Trans. on Comput., Vol. C-21, (Dec., 1972), pp. 1293-1310.

[16] R. Towle, "Control Structures Inside DO Loops," Ph.D. Thesis Proposal, Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Nov., 1974.

[17] S. C. Chen, and A. H. Sameh, "On Parallel Triangular System Solvers," submitted for publication.

[18] A. H. Sameh, S. C. Chen, and D. Kuck, "Parallel Direct Poisson and Biharmonic Solvers," submitted for publication, Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Rep. 684, July 1974. (NSF-)CA-GJ-36936-000006).

[19] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," Oper. Res., 9 (Nov.-Dec. 1961), pp. 841-848.

[20] E. W. Davis, Jr., "A Multiprocessor for Simulation Applications," Ph.D. Thesis, Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Rep. 527, June 1972.

[21] A. Bernstein, "Analysis of Programs for Parallel Processing," IEEE Trans. on Electronic Computers, Vol. EC-15, (Oct., 1966), pp. 757-763.

[22] D. Fisher, Program Analysis for Multiprocessing, Burroughs Corp., TR-67-2, May 1967.

[23] L. Lamport, "The Parallel Execution of DO Loops," Comm. of the ACM, Vol. 17, 2 (Feb., 1974)

[24] H. T. Kung, "New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions," Proc. of the Sixth Annual ACM Symposium on Theory of Computing, April 1974.

[25] D. Kuck, P. Budnik, S. C. Chen, E. Davis, Jr., J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towle, "Measurements of Parallelism in Ordinary FORTRAN Programs," IEEE Computer, (Jan., 1974), pp. 37-46.

[26] S. C. Chen, D. Kuck, and R. Towle, "Time and Parallel Processor Bounds for Fortran-like Loops," submitted for publication.

[27] S. C. Chen, D. Kuck, and R. Towle, "Control and Data Dependence in Ordinary Programs," submitted for publication.

[28] H. S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," J. Assoc. Comput. Mach., Vol. 20 (1973), pp. 27-38.

[29] A. H. Sameh, and D. Kuck, "Linear System Solvers for Parallel Computers," submitted for publication. Dept. of Comput. Sci., Rep. 701, Feb. 1975. (NSF-OCA-GJ-36936-000009).

[30] D. J. Kuck and A. H. Sameh, "Parallel Computation of Eigenvalues of Real Matrices," IFIP 71, Vol. II, pp. 1266-1272, North Holland Publishing Co., Amsterdam-London, 1972.

[31] A. H. Sameh, and D. Kuck, "A Parallel QR-Algorithm for Symmetric Tridiagonal Matrices," Proc. of Second Langley Conf. on Scientific Computing, Oct. 1974. Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Rep. 700, Feb. 1975. (NSF-OCA-GJ-36936-000006).

[32] B. L. Buzbee, "A Fast Poisson Solver Amenable to Parallel Computation," IEEE Trans. on Comput., Vol. C-22, 8 (1973), pp. 793-796.

[33] P. Morrison, and E. Morrison, Charles Babbage and His Calculating Engines, Dover, N.Y., 1961.

[34] F. L. Alt, "A Bell Telephone Laboratories' Computing Machine--II"" Mathematical Tables and Other Aids to Computation (The National Research Council), Vol. 3 21-28 (1948-1949).

[35] C. Leondes, and M. Rubinoff, "DINA, A Digital Analyzer for Laplace, Poisson, Diffusion, and Wave Equations," AIEE Trans. (Commun. Electron.), Vol. 71 (Nov., 1952), pp. 303-309.

[36] K. Zuse, "Die Feldrechenmaschine," Mathematik, Technik, Wirtschaft-Mitteilungen, Vol. 4 (1958), pp. 213-220.

[37] R. G. Hintz and D. P. Tate, "Control Data STAR-100 Processor Design," Compcon 72, IEEE Computer Society Conf. Proc., (Sept., 1972), pp. 1-4.

[38] W. J. Watson, and H. M. Carr, "Operational Experiences with the TI Advanced Scientific Computer," 1974 National Computer Conf., pp. 389-397.

[39] K. E. Batcher, "STARAN/RADCAP Hardware Architecture," Proc. of 1973 Sagamore Conf. on Parallel Processing, pp. 147-152.

[40] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," AFIPS Conf. Proc., 30 (1967), pp. 483-485.

[41] M. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Comput., Vol. C-21, 9 (Sept., 1972), pp. 948-960.

[42] M. Minsky, "Form and Content in Computer Science," ACM Turing Lecture, Journ. of the ACM, Vol. 17, 2 (1970), pp. 197-215.

[43] S. Winograd, "On the Time Required to Perform Addition," Journ. of the ACM, Vol. 12, 2 (April, 1965), pp. 277-285.

[44] S. Winograd, "On the Time Required to Perform Multiplication," Journ. of the ACM, Vol. 14, 4 (Oct., 1967), pp. 793-802.

[45] R. Brent, "On the Addition of Binary Numbers," IEEE Trans. on Comput., Vol. C-19 (1970), pp. 758-759.

[46] S. C. Chen, and D. Kuck, "Combinatorial Circuit Synthesis with Time and Component Bounds," submitted for publication.

[47] R. Brent, "The Parallel Evaluation of Arithmetic Expressions in Logarithmic Time," Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, N.Y. (1973), pp. 83-102

✓ [48] E. W. Davis, Jr., "Concurrent Processing of Conditional Jump Trees," Compcon 72, IEEE Computer Society Conf. Proc., (Sept., 1972), pp. 279-281.

[49] D. Kuck, "ILLIAC IV Software and Application Programming," IEEE Trans. on Comput., Vol. C-17, 8 (Aug., 1968), pp. 758-770.

✓ [50] P. Budnik, and D. Kuck, "The Organization and Use of Parallel Memories," IEEE Trans. on Comput., Vol. C-20 (Dec., 1971), pp. 1566-1569.

✓ [51] R. C. Swanson, "Interconnections for Parallel Memories to Unscramble p-Ordered Vectors," IEEE Trans. on Comput., Vol. C-23, 11 (Nov., 1974), pp. 1105-1115.

✓ [52] D. Lawrie, "Memory-Processor Connection Networks," Ph.D. Thesis, Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Rep. 557, Feb. 1973.

[53] H. Hellerman, Digital Computer System Principles, McGraw-Hill, N.Y., (1967).

[54] G. J. Burnett, and E. G. Coffman, Jr., "A Combinatorial Problem Related to Interleaved Memory Systems," Journ. of the ACM, Vol. 20, 1 (Jan., 1973), pp. 39-45.

[55] G. J. Burnett, and E. G. Coffman, Jr., "A Study of Interleaved Memory Systems," AFIPS Conf. Proc., 1970 Spring Joint Computer Conference, Vol. 36, (1970), pp. 467-474.

[56] E. G. Coffman, Jr., G. J. Burnett, and R. A. Snowdon, "On the performance of interleaved memories with multiple-word bandwidths," IEEE Trans. on Comput., Vol. C-20, pp. 1570-1572.

[57] C. V. Ravi, "On the Bandwidth and Interference in Interleaved Memory Systems," IEEE Trans. on Comput., Vol. C-21, (Aug., 1972), pp. 899-901.

✓ [58] D. Chang, D. Kuck, and D. Lawrie, "On the Effective Bandwidth of Parallel Memories," submitted for publication.

[59] R. L. Davis, "Uniform Shift Networks," IEEE Computer, Vol. 7, (Sept., 1974), pp. 60-71.

[60] D. Lawrie, "Access and Alignment of Data in an Array Processor," to appear in IEEE Trans. on Comput.

✓ [61] V. E. Beneš, Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, N.Y., (1965).

✓ [62] D. C. Opferman and N. T. Tsao-Wu, "On a Class of Rearrangeable Switching Networks," Bell Syst. Tech. Journ., Vol. 50, (May-June, 1971), pp. 1579-1618.

[63] K. E. Batcher, "Sorting Networks and Their Applications," 1968 Proc. Spring Joint Comput. Conf., pp. 307-314.

[64] M. C. Pease, "An Adaption of the Fast Fourier Transform for Parallel Processing," Journ. of the ACM, Vol. 15, (April, 1968), pp. 252-264.

[65] W. H. Stellhorn, "A Specialized Computer for Information Retrieval," Ph.D. Thesis, Dept. of Comput. Sci., Univ. of Ill. at Urbana-Champaign, Rep. 637, Oct. 1974) (NSF-OCA-GJ-36936-000003).

[66] G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes, "The ILLIAC IV Computer," IEEE Trans. on Comput., Vol. C-17, 8 (Aug., 1968), pp. 746-757.

[67] R. Moulder, "A Data Management System Utilizing the STARAN Associative Processor," 1973 Sagamore Conference Proc., p. 161.

[68] D. Kuck, "On the Speedup and Cost of Parallel Computation," to appear in Proc. on the Complexity of Computational Problem Solving, The Australian National Univ., Dec. 1974.

[69] W. Buchholz, Planning a Computer System, McGraw-Hill, N.Y., 1962.

Acknowledgment

I am indebted to S. C. Chen, D. H. Lawrie, and R. A. Towle for much discussion and help on the material discussed here.

PIPELINED PROCESSORS - A SURVEY[†]

C.V. Ramamoorthy and H.F. Li
Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California at Berkeley
Berkeley, California 94720

(Invited Paper)

I. Introduction

Processor architecture for computer systems has been a fast developing and fruitful area of investigation for many years. Through the impregnable efforts of past years, numerous distinguishing schemes for processor systems have evolved and secured their status in the art and science of computer design. Included in this category are:

(1) efficient sequential processors for sequential programs,

(2) pipelined processors [1] that enhance the amount of overlapped processing in each instruction stream,

(3) array processors [2] for executing some instructions on an array of data concurrently,

(4) associative processors [3-5] that possess some Content Addressable Memory (CAM) for certain decision making and information retrieval purposes, and,

(5) multi-processors [6] that may be necessary to meet some real-time constraints or for large computing systems with a heavy workload.

These five schemes are not necessarily exclusive; rather, some of them can be combined to produce a most effective processor system to satisfy some design objectives. For example, pipelined sequential processors are quite common, and associative array processors are used in certain large scale systems. But among these five schemes, pipelining stands out as a very versatile, popular, and effective technique to be applied to a wide range of systems, from mini, medi, to large scale systems, so as to improve their throughput rates or processing powers. In fact, pipelining can be employed as a basic complementary technique in sequential, array, associative or multi-processors as well. This clearly demonstrates the significance of pipelining in processor architecture.

The word 'pipeline' may be confusing to some laymen since a false image of oil pipes may be conceived. Actually, pipelining refers to a segmentation of a computational process into subprocesses so that the latter for successive instructions (computations) can be carried out in an overlapped fashion, analogous to an industrial assembly-line. So, very loosely, pipelining can be defined as the technique of decomposing a repeated sequential process (hardware or software) into subprocesses, each of which can be executed efficiently on a special dedicated, autonomous module that operates concurrently with the others. As a simple illustration, consider the process of executing an instruction. Normally, it involves fetching the instruction, decoding the operations involved, and fetching the operands before it is finally executed. If this process is being decomposed into the above four subprocesses executed on four modules as shown in Fig. 1b, four successive independent instructions may be executed in parallel. Specifically, while the EXEC module is executing the first instruction, the OPFETCH module may be fetching the operands needed for the second instruction, the DECODE module may be setting up for the different operations involved in the third instruction, and the INFETCH module fetching the fourth instruction. The overlapped execution among the four modules is best depicted using a space-time diagram. As drawn in Fig. 1c, the horizontal axis represents the time and vertical axis the space (modules). From it, one can observe how independent instructions can be executed in parallel in a pipelined processor.

[†]Research sponsored by National Science Foundation grants DCR72-03734-A01 and DCR74-21248 and U.S. Army Research Office Contract DA-ARO-D-31-124-73-G157.

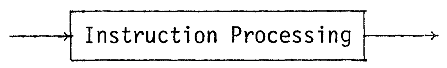


Fig. 1a Non-piped Processor

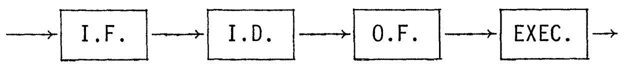


Fig. 1b Pipelined Processor

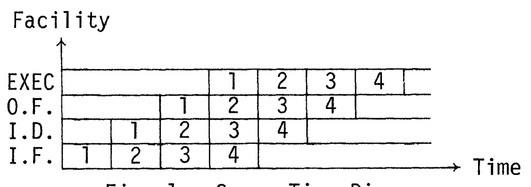


Fig. 1c Space-Time Diagram

With this macroscopic view of a pipelined processor, the advantages and requirements of pipelining will be reviewed here.

(1) Throughput Consideration: One of the most important performance measures of a system is its throughput rate, defined as the number of outputs, here the number of instructions processed, per period of time. Very obviously it directly reflects the processing power of a processor system -- the higher its throughput rate, the more powerful it is. Pipelining is a specific technique to enhance throughput, in addition to the possibility of using faster modules.

For this discussion, let us reconsider the example in Fig. 1. For a nonpipelined processor, the execution time of an instruction will be

$T_{np} = t_1 + t_2 + t_3 + t_4$. Therefore, for every T_{np} units of time, an instruction is completed, which corresponds to a throughput rate of $1/T_{np}$. In the pipelined case, suppose $t_b = \max\{t_1, t_2, t_3, t_4\}$ = speed of the slowest facility in the pipeline (defined as the bottleneck). Then, its throughput rate will be $1/t_b$ because for every $T_p = t_b$ units of time, an instruction can leave the pipeline after its execution. A direct comparison will reflect that $T_p < T_{np}$ with the result that the throughput rate of the pipelined processor can be much larger than that of the nonpipelined case. If $t_1 = t_2 = t_3 = t_4$, then the comparison can show four-time improvement.

Three characteristics have been hidden in this comparison. First, the decomposition of a process often prolongs the process by introducing some overhead. In this case, to guarantee non-interfering execution among the four modules, appropriate buffers have to be inserted between adjacent modules in the form of latches. These latches introduce additional propagation delay for an instruction. Consequently, the execution time of an individual instruction for the pipelined processor will be slightly larger than its nonpipelined counterpart. But one should not be mistaken that the resulting program execution time will be longer. In fact, usually it will be much shorter because more instructions can be executed per unit time. The execution time of an instruction is negligible compared to the execution time of a program or a large collection of instructions.

Second, it has been assumed that the execution time of a subprocess is not changed via pipelining; that is, the t_i's are the same in Figs. 1a and 1b. In practice, such an assumption may not hold. In some cases, because of the partitioning, some operations can be carried out more efficiently, and in some other cases, the converse may be true. However, in general, the assumption may be valid to at least a first order of magnitude.

Finally, no word has been said about the fineness of segmentation or pipelining. Since the comparison reveals that the throughput rate is $1/\max t_i$, it seems that decomposing the process into finer levels or modules may decrease $\max\{t_i\}$ and improve the throughput. But many practical considerations have to be made regarding the feasibility and overhead tradeoffs involved. They will form part of the later discussion in this paper. In short, throughput enhancement is one of the main advantages of pipelining. But the amount of improvement has to be evaluated carefully.

(2) Efficiency Consideration: Another important performance measure for a system is its efficiency, sometimes also called utilization factor. Efficiency also directly reflects how effective a processing scheme is and can be used to indicate how future improvements should progress, such as removal of bottlenecks. Similarly to most performance measures, it can be evaluated both analytically and experimentally by measurement. Here, an attempt will be made to illustrate the analytical efficiency of pipeline processing, based on the space-time relationship introduced earlier.

It is natural to view efficiency as the percentage of busy (productive) periods with respect to a certain time span of consideration. Here some slight complication arises because a pipelined processor consists of several modules some of which may be busy while the others are idle. To evaluate the efficiency of the processor system as an entity, [7] proposes a uniform space-time span index as:

$$\text{Efficiency of pipeline} = \frac{\text{total space-time span of tasks}}{\text{total space-time span of facilities}}$$

where the term task (process) is used to fit the loose definition of a pipeline. Sometimes, the modules in the pipeline are of different natures with different importance (or cost) factors. Then a refined index which also includes such considerations has been suggested in [8] as:

$$\text{Efficiency of pipeline} = \frac{\text{total weight space-time span of L tasks}}{\text{total weight space-time span of n facilities}}$$

For example, for a linear pipeline as the one in Fig. 1 (no looping inside the pipeline so that a task will flow through each facility once only), an analytical efficiency measure can be (assuming the execution time of each module is time invariant)

$$\eta = \text{Efficiency of linear pipe} = \frac{L\left(\sum^{n}\alpha_i t_i\right)}{\sum^{n}\alpha_i\left(\sum^{n} t_i + (L-1)t_j\right)} \qquad \text{(see Fig. 2)}$$

where

t_j = speed of the slowest facility (bottleneck)

t_i = speed of the ith facility in the pipeline

α_i = weight associated with the space-time span of the ith facility as determined as its importance, such as cost-speed factor

L = number of tasks (instructions) pumped into the pipeline in a certain period of time. For highest efficiency, it will be assumed that tasks are pumped in continuously.

η = total number of facilities in the pipeline

In the ideal situation when all modules have the same speed, the equation simplifies into

$$\eta = \frac{L}{n + (L-1)}$$

so that when L approaches infinity (in the steady state of processing), the efficiency may approach unity. In all other cases, as L approaches infinity, the efficiency approaches

$$\eta \rightarrow \frac{\sum^{n}\alpha_i t_i}{\left(\sum^{n}\alpha_i\right) t_j} < 1 .$$

Two observations should be noted at this point. First, this equation holds whether or not there are additional buffers inside the pipeline because of the linearity assumption. As will be demonstrated later, buffering is an important tool

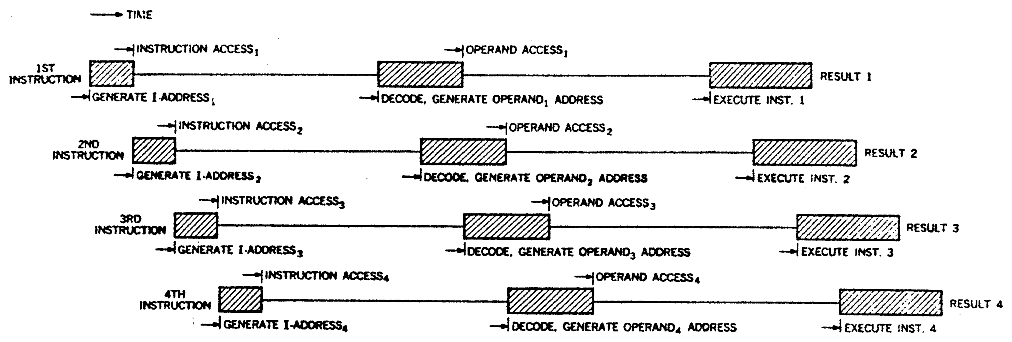


Fig. 2 IBM 360 Model 91 Instruction Sequencing Illustration

to increase throughput in many practical pipeline designs, for example, when more than one EXEC module is available if the latter is a bottleneck. Second, in deriving the equation, it has been assumed that a continuous supply of tasks (instructions) is available. In reality, execution may be discontinued because of various reasons such as precedence constraints, branching, interrupts, etc. which will be subject to closer scrutiny in the rest of this paper. This space-time span evaluation actually represents the efficiency limit of a pipelined processor if sufficient control and management have succeeded in justifying the assumptions to some approximate degree. The difficulties encountered here help to reveal many problems that underlie a pipeline design. These problems will be addressed individually in the subsequent sections.

(3) Cost-Effectiveness: Cost-effectiveness is one of the major advantages of pipeline processing. It has been demonstrated how pipelining can enhance the throughput rate. Yet one can achieve the same or even higher throughput rates using array or multi-processors. The tradeoff consideration here is cost and speed, or cost-effectiveness. Usually when a process is being decomposed into subprocesses executed on independent modules, the latter can be designed to execute those subprocesses very efficiently at a much lower cost than the nonpipelined counterpart. Cost here includes both the direct cost of the modules and the control cost required. Then if it is compared with an array or a multi-processor (nonpipelined) configuration, it may prove more 'cost-effective' simply because the latter may need more in both categories of cost. This is why pipelining is so important in smaller scale systems where the highest throughput for a certain cost bracket is desired. Of course, for large scale systems with array or multiprocessors, pipelining can be utilized as a complementary technique to enhance the throughput of individual processing units. In such a case, the cost-effectiveness of pipeline processing remains as its outstanding merit.

So far we have reviewed a few important characteristics of pipeline processing. Before proceeding further in discussing the problems and solutions in existing processors, the following design guidelines are included to generate a more complete picture.

(1) The repeated process can (most efficiently) be subdivided into subprocesses, each executed by an independent module in a compatible speed with respect to the others. When a certain facility in the pipe has a much slower speed than the rest, it will be the sole bottleneck and hence uniquely affects the throughput rate of the pipe.

(2) The submodules in the pipe and the associated control for sequencing them are cheaper than the nonpipelined counterpart in an array or multi-processor configuration. This is equivalent to the cost-effectiveness consideration just mentioned.

(3) Intermediate buffering is relatively cheap. Therefore the size of intermediate data packets or information transfers should be reasonably small, depending on the level of the pipelining action.

(4) Routing of intermediate information is easily accomplishable. If very complicated decisions or switching are involved, perhaps the overhead defeats the purpose of pipelining.

(5) Sharing of other system resources, including buses, memories, registers, etc., does not result in severe interference that degrades actual performance to a large extent. Sometimes, the inadequate supply of independent instructions or operands due to interference or other reasons will destroy the power of a pipelined processor and lower its efficiency drastically. However, this poses a number of design and operational problems not easily resolvable as we will observe later on.

(4) Pipeline Characterization: For the purpose of exposing the details of the problems to follow, a characterization of pipelining in processors will be provided here. Similar to many other techniques such as parallel processing, pipelining in processors exists in two levels and can take two forms. In the first level, pipelining can be seen in the entire instruction processing phase which is decomposed into autonomous subphases such as the one in Fig. 1b. Each subphase is represented by the execution of the corresponding module which possesses a certain amount of intelligence in controlling itself and communicating with the other modules. Tasks (instructions) are transferred from one module to the next for continued processing. But within each intelligent module, pipelining techniques can be further applied to speed up computation. Consider the EXEC module for example. Usually it is the slowest module in the pipe because many arithmetic operations require iterations or more levels of propagation delays in the logic circuitry. If

this module is pipelined by partitioning the operation into suboperations and inserting appropriate latches, the throughput of the EXEC module may be improved to a desirable extent. Embedded in this pipelining technique is the association of a local intelligent controller in each pipelined module for sequencing the pipelined operations to be executed and monitoring the corrections of the execution. The existence of such local monitors can be used to characterize the level of the pipelining action concerned. Here, the pipelined EXEC module may be regarded as the second level.

These two levels of pipelining can be seen in some computers. In the IBM System/360 Model 91 [9], the instruction processing unit is quite similar to the model in Fig. 1b. In addition, its execution unit includes a pipelined adder and a pipelined multiplier for providing a higher throughput of execution. Thus two levels of pipelining can be distinctly observed. Similar situations can be observed from other machines such as TIASC [10], CDC STAR-100 [11] systems. Hence, a top-down, level by level characterization of pipelining in processor systems can be conveniently established for the purpose of analyzing the system.

Besides the hierarchical nature of pipelining, different design and control strategies classify a pipelined module (whether it is level 1 or level 2) into two types: static and dynamic pipes. Sometimes, a pipelined module only serves a single dedicated function, for example, a pipelined adder or multiplier as in the IBM/360, model 91. Naturally, it can be termed as a unifunctional pipe with a static configuration. On the other hand, sometimes a pipelined module can serve a set of functions, each with a distinguishable configuration. For example, in the TIASC system, the arithmetic unit in the processor is a pipe that has different configurations (interconnection of modules) for performing different types of arithmetic operations. Then, a natural name for such a pipe is a multifunctional pipe. A multifunctional pipe can be either dynamic or static. In the static case, at any time instant, only one configuration is active. In other words, pipelining (overlapped processing) is permissible only if the tasks (instructions) involve the same configuration. Most, if not all, multifunctional pipes in arithmetic units of existing machines fall into this classification because static pipes are easier to control as will become clear later on. Dynamic multifunctional pipes permit overlapped processing among several active configurations simultaneously. Then throughput may be further enhanced. But the tradeoff here involves more elaborate control and sequencing so that in practice, its use has to be carefully justified. This classification of static and dynamic pipes will be very useful when we consider and evaluate pipelined processor architecture in the subsequent sections.

II. Structure of a Pipelined Processor

In this section, the basic structures of a pipelined processor will be examined, using the prominent IBM/360 model 91 central processor as the example. The throughput objective of a sequential pipe will be uncovered. Then from the analysis of its structure, the problems and requirements specific in pipelined processors will become noticeable. They will be discussed and some solutions in existing processors will also be illustrated and compared. However, attention regarding vector processing capabilities will be reserved for the next section.

2.1 An Example Sequential Pipelined Processor

To demonstrate the pipeline action in a sequential processor, the IBM/360 model 91 [9] will be used as the illustration. The central processor was designed to upgrade computational performance (throughput) by one or two orders of magnitude compared to the 7090 system via proper pipelining and circuit design. A typical instruction processing sequencing in the pipe can be as depicted in Fig. 2. Because of the highly overlapped operations among independent instructions, more instructions are completed per period of time, thus helping to achieve the desirable performance.

Let us look at each segment of the pipeline in more detail in order to observe the important problems and characteristics associated with a pipelined processor. Basically most segments of the pipe have a cycle time of 60 nsec, with the exception of the storage referencing and execution units. The different segments in the pipe are drawn in Fig. 3. The function of each segment is

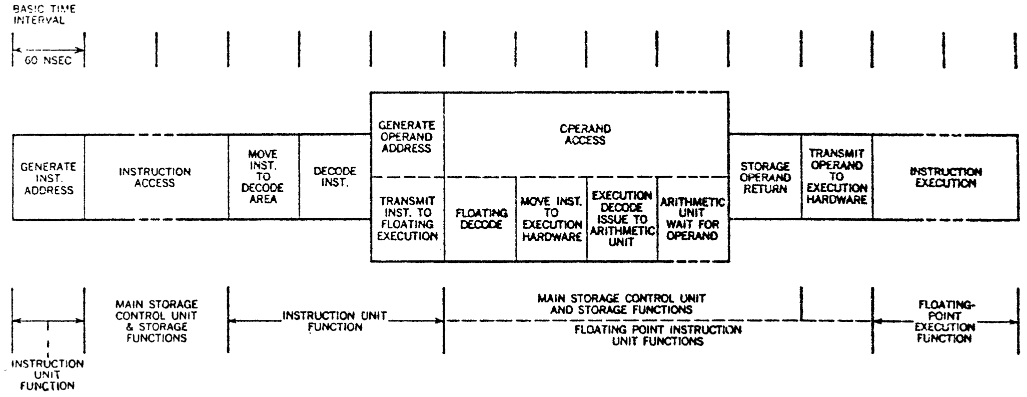

BASIC TIME INTERVAL
60 NSEC
GENERATE INST. ADDRESS
INSTRUCTION ACCESS
MOVE INST. TO DECODE AREA
DECODE INST.
GENERATE OPERAND ADDRESS
OPERAND ACCESS
TRANSMIT INST. TO FLOATING EXECUTION
FLOATING DECODE
MOVE INST. TO EXECUTION HARDWARE
EXECUTION DECODE ISSUE TO ARITHMETIC UNIT
ARITHMETIC UNIT WAIT FOR OPERAND
STORAGE OPERAND RETURN
TRANSMIT OPERAND TO EXECUTION HARDWARE
INSTRUCTION EXECUTION
INSTRUCTION UNIT FUNCTION
MAIN STORAGE CONTROL UNIT & STORAGE FUNCTIONS
INSTRUCTION UNIT FUNCTION
MAIN STORAGE CONTROL UNIT AND STORAGE FUNCTIONS
FLOATING POINT INSTRUCTION UNIT FUNCTIONS
FLOATING-POINT EXECUTION FUNCTION

Fig. 3 Functional segments involves in a floating storage-to-register instruction in model 91.

indicated in the figure. For a storage-to-register instruction, after the instruction has been fetched and moved to be processed, it will be decoded first. This decoding serves to decide the subsequent actions to be taken for this instruction. Since it is a storage-to-register instruction, two parallel sequences of operations will be initiated. The first sequence includes the effective address calculation and fetch for the operand from memory storage. To calculate this address, the delay time in the segment(s) involved is variable, depending on if it is indexed or not. The operand access segment again has a random delay, depending on the availability of the memory module to be referenced. The memory system in the 360 model 91 is interleaved to increase the bandwidth or memory supply rate. However, because of reference conflicts due to requests from other parts of the processor or system (such as instruction fetch or I/O), an operand fetch may have to be delayed for a complete memory cycle or more before it is acknowledged. This variable access time poses a constraint on the efficiency of the pipelined processor. A completely synchronous operation on the segments may be impossible because of these variable waiting times. And the need to be able to reduce the memory access time so as to match the speed of the other segments in the pipes remains to be one of the most critical issues in pipelined processor designs. With slow effective memory access time, the memory access segment may be a bottleneck of such a large magnitude that the throughput of the processor is not much improved via pipelining.

The second sequence of operations involves the setting up of operands to be submitted to an assigned execution station in the execution unit. If it is a floating point instruction, it will be mapped into a pseudo register-to-register (within the execution unit) instruction and transmitted to the execution unit. Here, the instruction is stored in the floating-point operand stack. In turn, it will be decoded and the operand registers (in the execution unit) concerned will be tagged. Then the execution unit will wait for the return of the operand from memory. When it happens, the two parallel sequences can merge (join) to initiate the next stage of processing, the actual execution. The ready operand pair will be transmitted to an available execution station to complete the processing. Of course, the segments in the above description operate independently of others in an overlapped mode, with suitable buffering in-between so as to achieve the pipeline objective.

It has been demonstrated how important it is to reduce memory access time, since most of the other segments have fast deterministic speeds. Even after the memory accessing problem has been solved, another bottleneck in the pipeline may emerge. This is the execution unit. Usually many arithmetic operations, especially floating point operations, require considerable delay because of their implicit internal circuit delay requirement or iterative characteristics. If there is only one execution station to serve the entire instruction stream coming in, the speed of the execution unit may not be compatible with the input rate, thereby unnecessarily slowing down the computation. One alternative is to provide multiple execution stations to perform different types of operations. In the model 91, there is a fixed point execution area and a floating point execution area. With this arrangement, floating and fixed point operations can be performed asynchronously but in parallel. But within each execution area, the multiplicity of execution stations can be increased, so that more floating or fixed point operation overlap can be achieved. This is equivalent to increasing the throughput of the execution unit as an entity. For example, the floating point area in the model 91 has two execution hardware: a pipelined adder and a multiply/divide pipe. The second level of pipelining in the execution station is an ingenious approach to speed up some slow arithmetic operations, though concurrency in execution among parallel stations already exists.

One notable characteristic of the model 91 execution unit is that it possesses multiple but unidentical execution hardware stations, for example, add, multiply/divide. This design decision was made by considering that a universal execution station might not be able to perform all functions as efficiently as specially designed stations, one for each type of operation. However, one should also note that this machine assumes sequential instruction processing. As hardware technology develops, some pipelined processors emerged with identical arithmetic unit pipes as in the TIASC system. Such a more general purpose execution hardware design is oriented towards vector processing which will be covered in detail in Section III. In that case, the execution hardware can assume a certain configuration during a vector instruction, such as adding or multiplying or two vectors. These two alternatives have their advantages and disadvantages. With only one type of execution hardware, more homogeneity is achieved and less cost incurred (in case there is only one unit). But to change its configuration for different operations may introduce too much switching overhead. Fortunately, for vector oriented applications, such an occurrence can be reduced. Moreover, as in the TIASC system, if the number of arithmetic unit pipes is more than one, they can be assigned dynamically certain configurations for certain applications, and hence, reconfiguration can be further reduced. Then it has the added advantage of being able to cope with an application requirement better.

Here we have shown the essential structures of a pipelined processor. Next, attention will be paid to studying some design and operational problems associated with a typical pipeline. Included are the following topics:

(1) Buffering: the concept and urgency of buffering in a pipeline and in what ways it can be accomplished.

(2) Busing Structure: For communication between segments and operand supply to allow processing to proceed or resume as quickly as possible.

(3) Parallelism Requirements and Handling: To secure correct execution and obey implicit precedence constraints in the instruction stream.

(4) Branching: Effect of branching in throughput and the ways to alleviate the inefficiency in existing systems.

(5) Interrupt Handling: How interrupts are handled in sequential and vector pipes.

(6) Sequencing Control: Need and usefulness for proper sequencing, and some mechanisms.

These six topic areas together will represent the major design constituents to be added to the basic structure just mentioned. Their importance and effects actually can decide the efficiency and performance of the resulting design.

2.2 Buffering

Buffering is essential in smoothing out the flow of a computation process when the exact timing for each processing module (segment) involved cannot be decided a priori. In the case of a pipelined processor, it is one of the most crucial but not very conspicuous components of the system. The impact of buffering here can be visualized in a common assembly line, say in the car industry. Occasionally a station (segment) of the pipe may be slowed down because of various reasons which prevent the continuous input of cars to the station. If there is sufficient storage space (gap) between this station and its predecessor (or successor) then the latter can continue its operation on other cars and ship them to the storage space available until it is full. When the station resumes service, it can try to clear up the cars in its input storage, perhaps at a faster speed. Concurrently, its predecessor (or successor) may take its break or continue providing useful service (perhaps to other stations as well). The advantages indicated here are that the waiting station can resume execution very quickly because inputs are already available, and that continuous flow may be achieved even though some occasional slowdown in a station happens (in a pipelined processor, the slowdown may be created by interference of resource requests and the nature of the varying times of some operations).

Therefore buffering is needed before or after any segment whose processing speed is not fixed. In a pipelined processor this means (1) memory storage access related stations including instruction fetch, operand fetch, and (2) execution unit stations. In a typical pipe as the model 91, the instruction buffer can hold 8 words of instructions to be followed in the sequence. In the execution unit, for the fixed point execution area, a buffer of 6 words of instructions (pseudo) and 6 words of operands is available, whereas in the floating point area, a buffer of 6 instructions and 6 operands (from storages) is also provided. These buffers serve the purpose of continuing the supply of instructions or operands to the appropriate units whenever a variable speed occurs. Similar buffers in other pipelined processors can be found. In the STAR-100 system, whose configuration is drawn in Fig. 4, a 64 quarterword (superword) buffer exists in the stream unit to buffer the data and to align the two operand vectors (in vector processing mode) for streaming in the operations involved. In addition, of course there is the instruction buffer holding 4 swords of instructions (each sword = 4 128-bit words). One sword in the instruction buffer will be filled by one memory fetch so that the buffer can supply a continuous stream of instructions to be executed even though memory conflicts may occur from time to time. Similarly in the TIASC system, whose schematic diagram is shown in Fig. 5, sufficient buffers are installed in the IPU and Memory Buffer Unit (MBU). The MBU specifically holds 8-word X, Y, Z (2 operands, 1 result) buffers to serve the arithmetic unit, and its instruction buffer consists of two 8-word fast register files. These are typical examples of the need and magnitude of buffering in a pipelined processor.

Sometimes, the concept of buffering may be applied to even a lower level of system consideration. As in the model 91 example, buffering can be installed also at each execution hardware station (such as adder and multiplier) to create virtual stations (the so-called reservation stations [12]). As explained before, the operations in the execution unit (for example, floating point) involve a sequence of inter-related segments. If during decoding the pseudo instruction, it is found that an unresolved dependency exists or the needed execution hardware is not available, further processing will be paused until the condition is removed. This will introduce the undesirable waiting time before execution can resume. One solution that could be considered is to add more execution stations. But it may not be a good alternative because the added stations may carry a lot of idle time due to their waiting for the

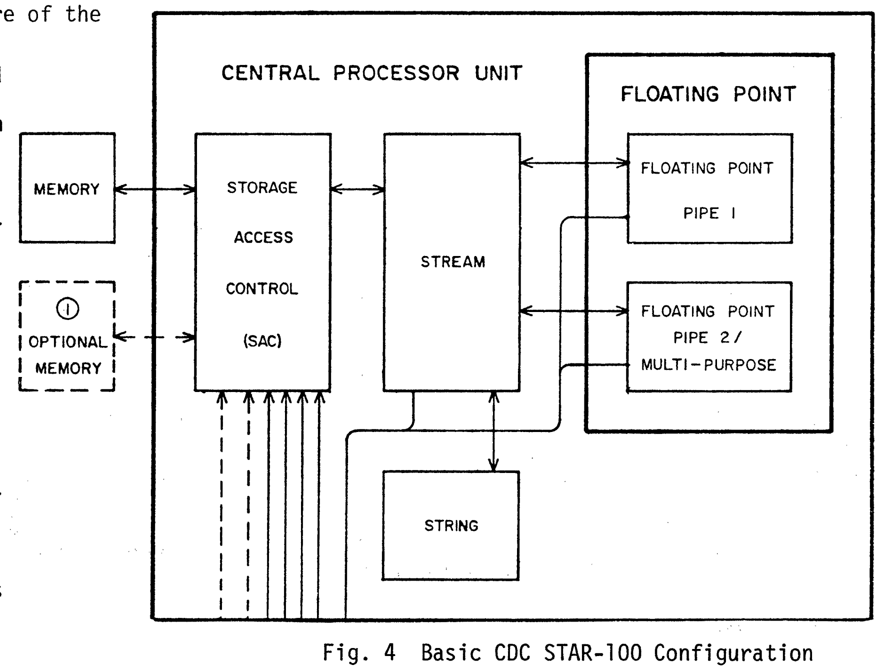


Fig. 4 Basic CDC STAR-100 Configuration

needed operands. So instead, the model 91 provides additional buffer pairs of each execution station. (There are three add and two multiply/divide reservation stations). Then while one buffer pair is being executed by the hardware station, the others can be established to receive operands for future execution. Thus virtual execution stations are formed.

2.3 Parallelism Requirement and Busing Structure

Pipelining requires the concurrent processing of independent instructions, though they can be in consecutive stages of execution. Once encountering an instruction that receives a source operand to be generated (or modified) by some previous but not yet completed instruction, further processing beyond the decoding and address calculation stages may become infeasible. The precedence constraint implicit in an instruction stream must be preserved. Otherwise, using a faulty operand will ultimately lead to a large number of errors that propagate throughout the entire computation.

Therefore, parallelism in an instruction stream has to be detected efficiently and correctly. It becomes more important since the efficiency or performance of the pipe almost is directly proportional to the parallelism factor in an instruction stream. With dependent instructions, their input and traversal through the pipe have to be paused until the dependency is resolved. This reduces overlapping. In turn, a further problem arises: how to resolve the dependency most effectively in order that computation can resume as early as possible? Usually this means how the new source operand should be transmitted to a convenient location for further processing. Thus, the design of an efficient internal busing structure is implicitly needed.

Parallelism in a sequential instruction stream can be detected by checking the source operand addresses of an instruction with the sink (result) operand addresses of instructions still inside the later parts of the pipe. If the source address matches with the result address of some earlier but uncompleted instruction, the former must be inhibited until its contents reflect the result of the most recent operation to use that address as its sink. For example,

```
LD  R1  LOC1  (Load)
MD  R1  LOC2  (Multiply)
```

Then the load must be completed before the multiplication can take place -- if not, the register R1 will be holding an erroneous operand

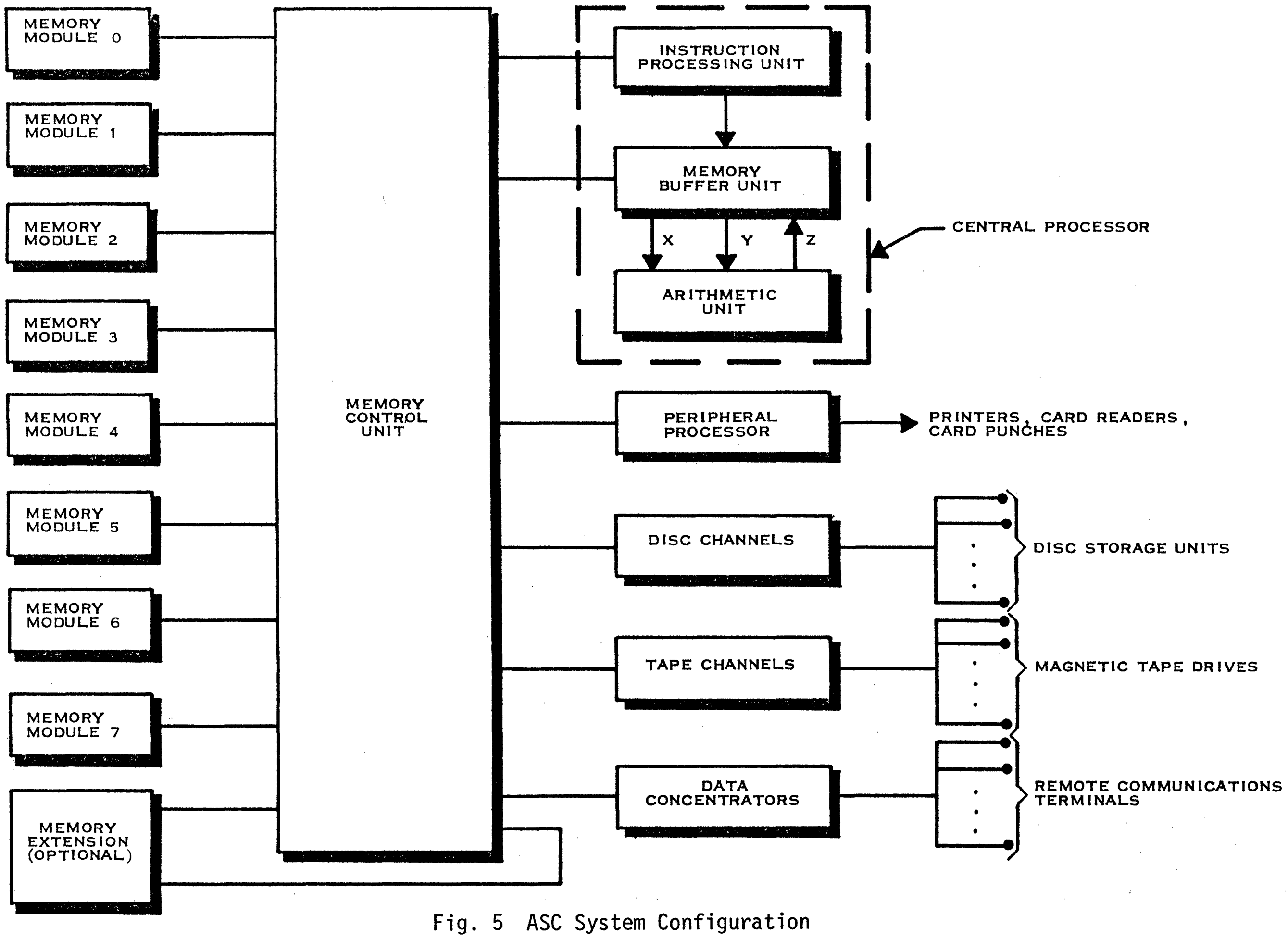


Fig. 5 ASC System Configuration

for the multiplication. This illustrates the basic precedence relationship that may exist between instructions. But the recognition hardware must not hold up independent instructions from entering the pipe.

One common technique in pipelined processors to accomplish this is to install a fast hardware scanner that compares the source addresses of the instruction (during or immediately after the decoding stage) with the sink addresses of previous uncompleted instructions. Once dependency is identified, two alternative actions can be taken.

In the IBM 360, model 91, since storage-to-register instructions are mapped into pseudo-register-to-register instructions, dependency is easy to check (notice that the mapping must be preserved in subsequent instructions). As an example, let us consider the floating-point execution unit again [12]. A busy bit is associated with each of the floating point registers. It will be set when it serves as the sink of some decoded instruction in the floating-point operation stack, and reset when the result is returned to the register. If a dependency is encountered and detected, the decode sets some control bits of the source register. Then when the result becomes available here, it will be transmitted immediately to some destination buffer where execution can proceed. (Therefore an execution unit station has been assigned to the dependent instruction while it is waiting for its source operand).

To allow processing to resume fastest, some needed results have to be transmitted to certain execution stations as quickly as possible. So the Common Data Bus (CDB) in the model 91 was invented. With the CDB, the entire floating-point unit is drawn in Fig. 6. The CDB can transfer data not only to the registers but also to the sink and source registers of all reservation stations (the virtual execution stations). It is fed by all units that can alter a register. To make this possible, tags (address) are assigned to the registers. Then the processing sequence can be described as follows. In decoding each instruction, the busy bit of each source register will be checked. If it is zero, the independent instruction can be transmitted to a certain execution station, say A1 (virtual adder 1). At the same time, the busy bit of its sink register will be set and the corresponding tag set to the destination of A1 (so that the sink register will receive the result from A1). If the busy bit is on, instead of waiting for the source operand to be generated and stored to the register, the dependent instruction will still be issued to an available execution station, say M1 (virtual multiplier 1). However, the tag of the register, rather than its content, will be transmitted to the reservation station M1 so that M1 will accept data whose tag matches with its own from the CDB. As an illustration:

$$\text{ADD}\quad \text{F1,FLB1}\quad ((\text{F1}) + (\text{FLB1}) \rightarrow (\text{F1}))$$
$$\text{MD}\quad \text{F1,FLB2}\quad ((\text{F1}) \times (\text{FLB2}) \rightarrow (\text{F1}))$$

In executing the ADD, A1 is used, and the tag of F1 is set to 1000 (that of A1) and its busy bit set to 1. In decoding the MD, the busy bit of F1 is 1. So rather than sending (F1) to M1, its tag (1000) is transmitted to M1. In addition, the tag of F1 is changed to 1010 (tag of M1). When CDB is broadcasting the data tagged with 1000, M1 will succeed in matching the tag and so ingate it to the buffer and resume execution (if FLB2 is available). Notice that the result of ADD is not stored in F1 in reality because that operation is redundant (the tag of F1 is 1010 and not 1000).

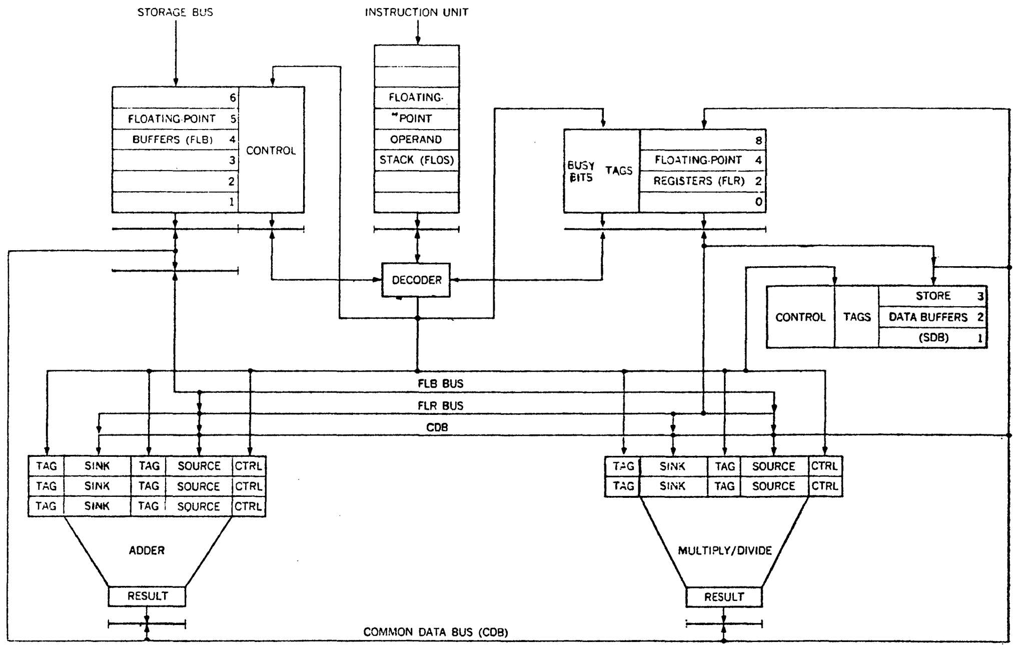


Fig. 6 Floating Point Unit of IBM 360 Model 91 with CDB and Reservation Stations

This CDB with tagging permits the intermediate operands generated from the pipe to be used most quickly by the following instructions, without having to go through many levels of actual storage (to actual registers or even to memory storages). However, its effects are rather local in the instruction stream (which is exactly what is desired in keeping the smooth flow of the instruction stream).

A similar alternative can be found in other pipelined processors such as the TIASC and CDC STAR-100. In the TIASC processor [13], an instruction dependency is recognized by hardware. It scans the instruction stream and distributes the independent instructions across MBU-AU pairs to insure proper, yet efficient execution sequences. Update capability is incorporated by allowing the contents of the Z-buffer to be transmitted to the X- or Y-buffer in MBU when the latter are being used as scratch pads in local computation. In the STAR-100 system [14], a more explicit busing structure is maintained because of its different units. In the floating point pipes (whose configurations are drawn in Fig. 7), a direct route called shortstop is established between the output (transmit segment) of each pipe to either of its inputs. This eliminates the time necessary to store the generated result in the register file and then to read it out again.

Although an efficient busing structure can reduce the adverse effect of instruction dependency, there is still a big burden on the programmers or the compilers to produce codes that expose sufficient parallelism to allow overlapped processing beneficial. If more independent instructions are intermixed appropriately with those dependent ones, more concurrent processing can take place while the dependency is resolved with little incurre time (that is, the resolving of dependency is hidden behind other useful processing). This is a very important factor in deciding how efficient a program or an implemented algorithm can be executed on a pipelined processor. Algorithm efficiency must also consider the architectural features of the processor on which it is executed.

2.4 Branching

Branching is another serious adverse effect that may arise because of program structures. It is more damaging to the pipeline performance than the previous instruction dependency. When a conditional branch is encountered, one cannot tell which sequence of instructions will follow until the deciding result is available at the output. Therefore, a conditional branch not only delays further execution but also affects the entire pipe starting from the instruction fetch segment. An incorrect branch of instructions and operands fetched may create a discontinuity of instruction supply.

To remedy the effect of branching, different techniques can be employed to provide mechanisms whereby processing can resume even if an unexpected branch occurs. In the IBM 360 model 91 [9], a loop node and back-eight test are designed with

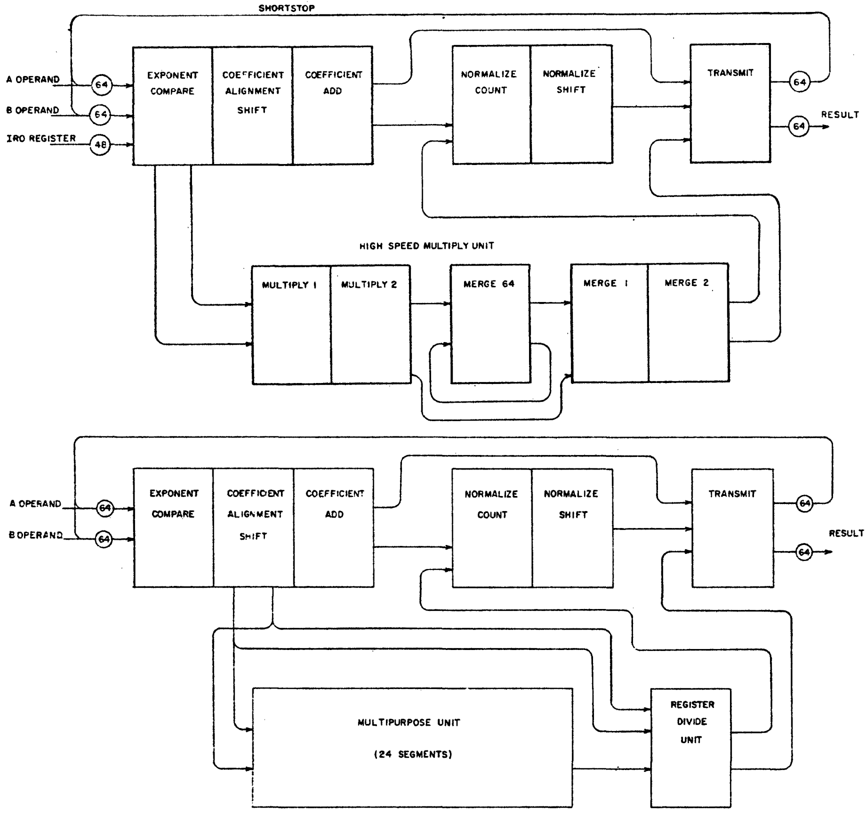


Fig. 7 Floating Point Pipe 1 and 2 of CDC STAR-100 system

the help of an additional branch target buffer. In TIASC, a load lookahead [15] mechanism (instruction) is explicitly provided, with appropriate hardware and buffer support. Likewise, in the CDC STAR-100 [14], the instruction stack has special branch back capability. We will try to explain these schemes in this section.

The branch-on-condition handling is best illustrated using the model 91 example (Fig. 8). In this processor, upon decoding a conditional branch instruction, this instruction will be tagged so that its outcome can be used to set a condition code (CC) (provided it is not inhibited). Also it will be assumed that no branch will be taken (CC not valid). However, to guard against an incorrect guess, two instruction double-words will be fetched from the branch and stored at the branch target buffer. Then conditional mode is entered where instructions are forwarded conditionally to later segments for processing. Operands are conditionally set up while actual execution is prohibited. Finally, if the CC becomes valid (branch should be taken), the conditional instructions must be deactivated and processing resumed using the branch target instructions. If the CC is invalid as guessed, then execution can continue almost instantaneously. This therefore reduces the waiting time on the average case (if the guess is more 'right' than 'wrong'). To further reduce instruction fetching time, short loops in programs can be fruitfully exploited. Very often, a short loop-backward in programs can be seen. If the instructions are already in the instruction buffer, it is wise not to erase any of them and assume the branch (loop) will be successful. Then no other memory access for instructions is needed and less memory interference to other parts of the processor will be created. The way to detect these short loops and reserve the instruction loop is by implementing a loop node and back-eight test.

A separation of eight instruction double words or less will be termed a short loop that can be completely stored in the instruction buffer. When a branch (backward) is obtained, the back-eight test will be used. If it is satisfied, the loop mode will be established. From that point on, the complete loop is fetched in the instruction buffer so that no further fetching is needed until the loop mode is removed by branching out. In conditional branches, the loop mode can be established to replace condition mode once a successful branch results and the back-eight test is satisfied. This method of back-eight test and loop mode is very useful in systems where available memory cycles are precious to the entire system.

The load lookahead mechanism in TIASC follows a similar philosophy. The instruction processing unit of the machine contains two instruction address registers (Present Address, PA and Lookahead Address, LA) and two instruction files of 8 words each (KA and KB). Each memory reference can fetch an octet (P) of instructions to one of the instruction files. Usually PA contains the starting address of the next octet to be fetched and LA supplies the address of the next octet to be fetched. To accommodate branching for a loop, a branch with lookahead can be set up by placing the branch instruction at the target location of a Load Look-Ahead (LLA) instruction. A LLA enters a count into a lookahead count register (LC) and enters the address of the LLA into a branch address register. The count corresponds to the difference of the instruction locations of the LLA and its target branch instruction. The count is decremented by one every time an instruction is executed, following the initiation of the LLA. When it has reached

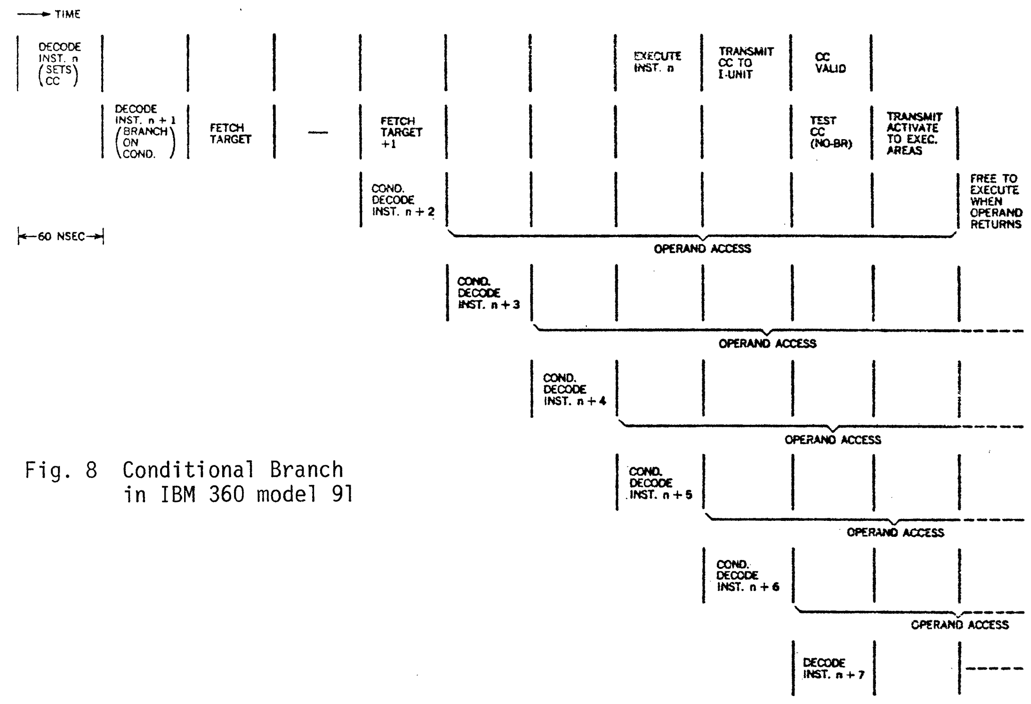
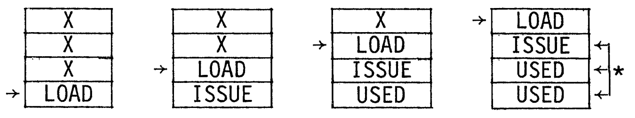


Fig. 8 Conditional Branch in IBM 360 model 91

a value designating that the branch has already been requested from memory, the control will transmit the contents of BA to LA. This causes the fetching of the octet containing the LLA and the loop control is reinitialized. In this way, a lookahead loading of instructions in a loop up to 256 instructions is allowed and instructions will be continuously available for execution before the branch instruction is completed.

The STAR-100 processor has a 16 128-bit word instruction stack. Each quartersword is loaded in one minor cycle (i.e. 4 words). Branching is allowed within the instruction stack. The loading and management can be as depicted in Fig. 9. After the stack is loaded, any branch within the stack can be honored easily. However, the stack will be cleared whenever a branch out of the stack occurs. The reason is because the stack can be completely filled by a request to memory (i.e. in one memory cycle). With a faster memory rate and a large instruction stack, local loops can be maintained easily without having to install testing strategies.

(4-sword Instruction Stack)



*Branch back on any previous part of the stack.

Fig. 9 STAR-100 instruction stack loading and issuing with branch tolerance

Therefore these methods are useful to help to supply instructions continually to the pipe segments even though branch instructions are inevitable. For fixed (targeted) branches, lookahead strategies can provide the means to continue the instruction sequence. But for conditional branches, more elaborate schemes to recover from unexpected branches have to be established (such as the conditional mode).

2.5 Interrupt Handling

Interrupts, as deemed inevitable, have the same adverse effect to pipelining as conditional branches. When an interrupt occurs, subsequent instructions (that follow the interrupt logically) have to be inhibited until the interrupt is served. Otherwise, a large overhead in recovery to the logically correct form may be needed. In the IBM 360 model 91, the notion of imprecise interrupts is used. These are instructions where interruption can be uncovered during decode time and hence involve interrupts that result from protection, addressing and execution functions. Once an interrupt is encountered, further decoding is prohibited. However, there may still exist instructions inside the pipe that are partially completed and which should be finished before switching the CPU to the interrupt routine. So the new status word (for the interrupt branch) is fetched to the branch target buffer in parallel with the execution completion. If the interrupt happens to be a precise one, the execution completion may cause an imprecise condition. Then the logically preceding imprecise signal should cancel all previous precise actions. Afterwards, processing can proceed down the interrupt instruction path.

For vector processing, execution of an instruction may take a long time. Therefore, as in the STAR-100 processor, special interrupt counters are available to hold addresses, delimiters, field lengths, etc. which are necessary to restart vector-type instructions after an interrupt (provided the interrupt does not affect the instruction processing sequence). This represents a recovery mechanism for processing to proceed afterwards when an unpredictable interrupt occurs. Since interrupts, unlike parallelism detection and branch instructions, are less predictable but fortunately less frequent, little distinguishable optimization techniques have been invented to reduce its affect to the pipeline continuity. Most adopted techniques are rather simple and crude.

2.6 Sequencing Control

For a second level pipeline (usually in the execution unit, such as a pipelined multiply or arithmetic unit pipe), the speed of a segment is fixed. Then rather than inserting buffers to interlock the operations among the segments, a sequencing control for routing operand pairs through the segments can be established. An exact schedule for traversing the segments can be followed after the execution is initiated. It has the additional advantage that with the removal of internal buffering (in the execution unit pipe), less delay and hardware in the pipe will be needed.

Since some pipelined execution unit requires internal looping (bi-flow pipe) in the segments, operand pairs admitted to this pipe must be routed properly so that two different active pairs will not try to access a same segment concurrently (recall that there is no buffer to resolve conflicts). This conflict avoidance is an important objective in the sequencing control. A general technique that can be applied to sequence operands properly can be developed as in [16]. A reservation table is used to represent the traversal path of an operand pair through the pipe. A typical example can be found in Fig. 10. In this example,

Time / Facility	1	2	3	4	5	6	7	8	9	10	11	12
1	+	×	0							+	×	0
2		+	×	0		+	×	0				
3			+	*	⊗	0						
4					+	×	0					
5							+	*	⊗	0		
6									+	×	0	

Static Collision Vector = 100100001
* = collision if + and × are initiated as indicated.

Fig. 10 Reservation Table for Sequencing

if the control waits for two minor cycles before initiating a second execution, no collision (resource conflict) will result. However, if initiation takes place at the next minor cycle, a collision will occur at facility 3 (or 5). From this reservation table, a static sequence control can be designed so that operand pairs are initiated at certain periodic intervals to attain a high throughput rate.

Specifically, a collision vector can be defined so that a '0' denotes that initiation at that interval (after the current initiation) will not cause any collision and a '1' the opposite. For the example, the collision vector is 100100001. Then, optimizing methods (usually mathematical programming techniques) can be employed to find periodic intervals to initiate the operand pairs to be admitted, assuming the latter exist. In fact, a state transition diagram as drawn in Fig. 11 can be constructed, a state being defined by the updated collision vector (previous updated vector or the static collision vector associated with each initiation) and a transition representing the number of cycles waited before initiating the next operand pair. From the transition diagram, the highest throughput cycle can be picked to initiate operand pairs to the pipe. In the example, it is the 1-1-1-5 cycle. This method of statically controlling the pipe is one way to guarantee highest throughput when a continuous stream of operand pairs to be executed is available. If the latter is not true, the collision vector approach can still be modified and applied to control the correct sequencing of operand pairs without incurring collisions.

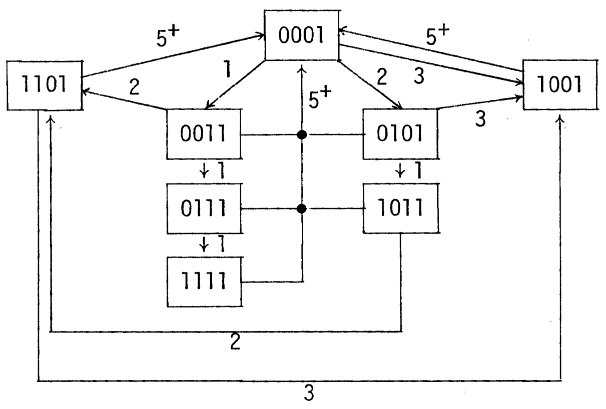


Fig. 11 Example State Transition Diagram

In the case of multifunctional pipes, the collision avoidance technique can be generalized. However, now two possible alternatives should be considered. In a static multifunctional pipe, only one configuration will be active at one time. So the previous unifunctional approach can be taken. But one must watch out for the overhead incurred in the reconfiguration process. Therefore, the sequencing control should try to scan and group instructions which require the same configuration to be executed together. In a dynamic multifunctional pipe, simultaneous active configurations are allowed. Then the generalization of the previous technique can be easily applied. For brevity, the details will be omitted here [17-18].

These techniques are useful not only to control operand routing correctly, but also to increase the throughput of a pipe with fixed-speed segments. With them, one can further upgrade the pipelined processor throughput.

2.7 Summary Discussion

To sum up the adverse effects of precedence constraints, branching and other unanticipated events to the performance of a pipelined processor, let us try to derive its analytical throughput with suitable parameters.

Consider a linear deterministic pipeline of L segments and suppose

p_0 = probability that a task (instruction) does not depend on anyone already in the pipe, that is, once initiatied, it can proceed without waiting or being cancelled.

p_i = probability that a task (instruction) depends on the ith previous instruction still in the pipe to validate its initiation, for $i = 1,\ldots,L$.

Thus

$$\sum_{i=0}^{L} p_i = 1$$

T_i = relative initiation time of the ith instruction.

For simplicity, let all facilities have the same speed T. Then

$$T_i = T_{i-1} + p_0T + p_1LT + \sum_{j=2}^{L} p_j[\max\{0,T_{i-j}+LT-T_{i-1}\}].$$

In the steady state, assume

$$T_i - T_{i-1} = T_j - T_{j-1} = d$$

(that is, expected delay in initiation between two consecutive tasks is d). Since

$$T_{i-j} - T_{i-1} = T_{i-j} - T_{i-j+1} + T_{i-j+1} - \cdots - T_{i-1}$$
$$= (j-1)d$$

$$d = p_0T + p_1LT + \sum_{j=2}^{L} p_j[\max\{0,LT-(j-1)d\}] .$$

More precisely, there exists an r such that

$$LT - (r-1)d \geq 0 \quad \text{but} \quad LT - rd \leq 0 . \qquad (1)$$

Then

$$d = p_0T + p_1LT + \sum_{j=2}^{r} p_j[LT - (j-1)d] . \qquad (2)$$

Equations (1) and (2) can be used to solve for r and d given p_i, L, T. But due to the nonlinear characteristics, a closed form solution is not available and an iterative algorithm for specific values of p_i, L, T has to be used. The index r arises because the present instruction may depend on only up to r previous instructions (on the average) still inside the pipe, instead of a maximum of L. This is because a cumulative delay may have resulted in these r previous instructions so that when considering the present instruction, the earlier ones (earlier than those r instructions) have already left the pipe.

Here, for the purpose of demonstrating the effects of sequencing, equation (2) is worth a second look. By proper sequencing, for every step, one tries to increase p_0 (no unresolved dependency) and other higher p_k $(k \geq r)$ (or dependency is far away) as much as possible. Several ways to achieve this exist. One is to produce optimized code via good programming or by a clever compiler, or a simple effective sequencing rule (perhaps implemented in hardware). Indeed, pipeline efficiency is highly dependent on both the design and operational methods used.

III. Vector Processing

One of the main requirements in justifying the pipelining of a process is that the same process will be invoked very frequently. Ideally if a continuous excitation of the pipeline is attained, then the maximum throughput will be within reach. For a pipelined processor, this is equivalent to the need of abundant parallelism in the instruction streams to permit the initiation of independent instructions almost continuously.

This ideal situation sometimes becomes true when the machine is processing some independent vectors such as adding two vectors, element by element, to form a result vector. If each element of a vector has to go through a transformation independent of the transformation of other elements of the vector, then they can be performed in an overlapped mode with the others, employing the pipelining characteristics. For machines with multifunctional pipelined execution units (second level), the latter can establish and retain a static configuration throughout until the entire vector is processed. Hence minimal control, decoding and reconfiguration overhead may be achieved while the memory operands are supplied to the execution unit in a most efficient way. This will become more apparent as our discussion proceeds.

In this section, vector processing in pipelined processors will be studied carefully. In subsection 3.1, the components of a vector instruction and the ultimate processing procedures will be demonstrated and a comparison of two prominent vector machines in this aspect will be included. This will then lead to the revelation and evaluation of the requirements, properties and tradeoffs in terms of time and space (control hardware) overhead in vector processing as contrasted with sequential pipeline processing. The analysis in subsection 3.2 will serve to expose the real crux behind vector processing. Hopefully, these discussions will also reveal the many facets, advantages and disadvantages and other special features associated with a vector pipe that may appear quite mysterious to some people.

3.1 Vector Instruction

A vector pipe can be characterized by the existence of one or more multifunctional pipes (second level) in the execution unit (arithmetic and logic unit) and the needed control and parameter specifiers in the processor. As mentioned in section 1, a multifunctional pipe can be either static or dynamic, depending on its reconfiguration control. In the static case, simpler control is required to establish and maintain a desired configuration for processing. There is a fixed route for each operand set to transverse throughout the computation, unless a new configuration is formed. While in the dynamic case, more complicated control and routing overhead will be involved, the throughput may be higher because of the simultaneous existence of several configurations. In reality, static vector pipes are more common, as will be illustrated in the TIASC and CDC STAR-100 examples to follow. Dyanmic vector pipes, though they may be superior in throughput, require too much control overhead and so their implementations still have to be studied more carefully.

For a vector that consists of the two levels of pipeline action, appropriate vector instructions have to be designed and implemented to denote the operations on some ordered data in vector or array form. Generally, in the first level, a vector instruction will be fetched, decoded, and the necessary control paths connected, before the needed elements of the vector are fetched from consecutive storage locations over a specified address range. The second level execution unit pipe carries out the specified operations on these elements, normally being supervised by a control ROM. Sometimes the results generated are stored back to certain consecutive addresses of a result field and sometimes other needed indicators will be generated and stored in the register file in the processor for future usage. The exact procedure and mechanism to accomplish all these functions vary from machine to machine. For the sake of later comparison and analysis, a description of an example of vector instruction execution will be provided here.

Before starting the execution of a vector instruction, certain additional information pertinent to the mode of processing has to be furnished to the system. Such information can be quite varied and detailed, such as the starting (base) address of each source vector and result vector involved (usually two source vectors and one result vector) and the control over what elements of the vectors should be operated upon. The method by which the CDC STAR-100 handles this will be demonstrated first. Then similar and different features in the TIASC system will be noted. Finally the vector processing power of the two systems can be compared.

The schematic diagram of the central processing unit for CDC STAR-100 system is drawn in Fig. 4. Basically it consists of four parts, operating in an overlapped, asynchronous mode: (1) Storage Access Control (SAC), (2) Stream, (3) String, (4) Floating Point units. The SAC is responsible for sharing the magnetic core storage among the 3 read and 2 write buses shared by the Stream and I/O units. To support virtual addressing (all user programs are run in virtual address space), it is also equipped with a small associative page table. The Stream unit provides the basic control for the entire processor. Internally, it may be regarded as a multi-segment pipeline (second level) as its carries out functions including

(i) memory references,
(ii) buffering and skewing of operand data,
(iii) buffering and decoding instructions,
(iv) setting up control signals for processing the instruction,
and (v) performing simple logical and arithmetic operations.

The String unit, as the name implies, is used to process strings of decimal or binary digits. It contains some fast half adders and full adders to carry out simple pencil and paper algorithms for binary arithmetics (divide and multiply). Finally the Floating Point unit consists of 2 pipes whose configurations are drawn in Fig. 7. Each pipe is (static) multifunctional as it has different configurations for performing different floating point operations. Pipe 1 performs arithmetic operations on operands in floating point format and address operations on nonfloating point numbers. Pipe 2 performs only two vector address type operations, in addition to other arithmetic operations. Pipe 1 and pipe 2 are quite similar in structure except that the latter has a high speed register divide unit and a multipurpose unit for some special arithmetic such as square root, vector divide, etc. The pipes can take on a certain configuration at any time. For example, to perform floating point addition, pipe 1 configures itself (under microcode control to be explained later) to activate the path: Exponent Compare - Coefficient Align - Coefficient Add - Normalize Count - Normalize Shift - Transmit. With this static configuration, operand pairs can be routed through the pipe at a steady and maximum rate. When the operand pairs can be supplied fast enough and the result stored suitably, an ideal throughput rate will be reached. Then if these pipe segments have the same speed, say one minor cycle, then one result element may be generated per minor cycle. The evaluation, other tradeoffs and overhead will be examined more closely later.

Let us now pause to examine a vector instruction before exploring the procedure of its execution. An ordinary vector instruction format in the STAR-100 computer is representable by 8 fields as indicated on Fig. 12; (1) F: function code, (2) G: subfunction code, (3) X, Y specify the registers that hold address offsets for the two corresponding source vectors (the offset operates as depicted in Fig. 13 and is useful for skewed vectors), (4) A, B specify the registers that hold the base addresses and field lengths of the two source vectors, (5) Z specifies the register holding the base address of the control vector, (6) C specifies the register holding the base address and field length of the result vector, and (7) C+1 then automatically specifies the register holding the offset for the control and result vectors. This automatic assignment is implied to maximize the utilization of each instruction word which has a limited length.

From these registers, the effective starting address and field length of each vector can be calculated. Then the rest of the vector can be referenced sequentially until a termination condition is reached. The control vector is a unique feature introducing the flexibility desired in vector processing. It performs prohibition responsibility, analogous to the control unit in an array processor such as the ILLIAC IV [2]. In the ILLIAC IV, the control unit broadcasts control signals to all the 64 processing elements so that the latter, except those inhibited by previous broadcast signals, will execute some operation on the appropriate array data. The control vector in the STAR-100 performs the analogous function, but in a time stretched fashion (compared to the simultaneous inhibition of array elements). Each bit of the control vector is used to specify whether or not the corresponding result element should be stored (for most vector instructions; however, in some modified cases like macros it has other duties as will be explained later). When a bit is set in the control vector, the corresponding element of the result vector will not be modified and stored. Thus, the nth bit read from the control vector will be used to control the storing of the nth element generated in processing the vector instruction.

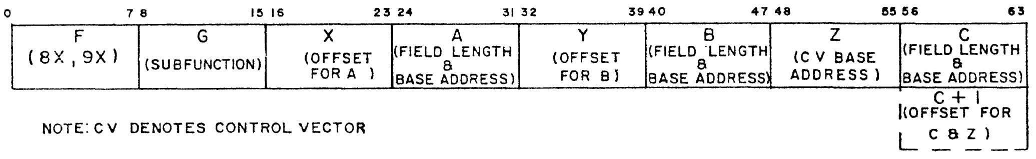


Fig. 12 Vector Instruction Format in CDC STAR-100

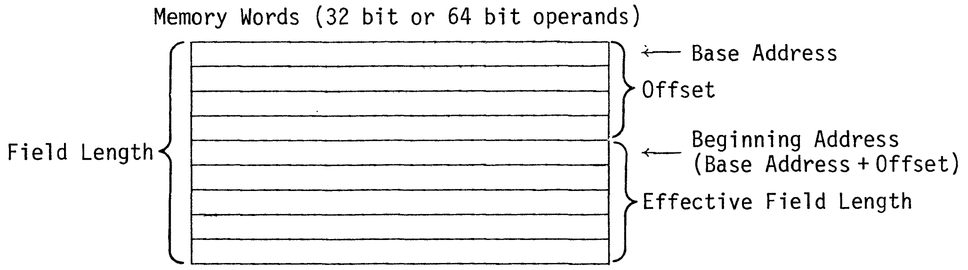

Memory Words (32 bit or 64 bit operands)
Base Address
Offset
Field Length
Beginning Address (Base Address + Offset)
Effective Field Length

Fig. 13 Addressing Offset for Vectors

As an illustration consider a vector add instruction:

VADD A,B,C (A+B→C)

Suppose the instruction format provides the following information:

(A) = content of A register:
field length of A vector
= 12 half-word (32 bits)
base address = 10000_{16} (bit addressing)
(B) = field length of B vector
= 4 half-word
base address = 20000_{16}
(X) = offset for A vector = 4 half-word
(Y) = offset for B vector = -4 half-word
(Z) = base address of control vector
= 40004_{16}
(C) = base address of result vector
= 30000_{16}
field length = 12 half-word
(C+1) = control vector and result vector offset
= 4 half-word

Then the starting address and effective field length of A vector can be calculated according to:

starting address = base address + offset
effective field length = length - offset

Hence, for the example, the effective addresses and field lengths are:

starting address of A vector
= 10000_{16} + offset = 10080_{16}
effective field length = 12 - 4 = 8 halfwords
starting address of B vector = 20000_{16} - 80
= $IFF80_{16}$
effective field length = 4 - (-4) = 8 halfwords
starting address of C vector
= 30000_{16} + 80_{16} = 30080_{16}
effective field length = 12 - 4 = 8 halfwords

The results of the VADD instruction can be summarized in Fig. 14. Notice that the addressing used is bit address and a '1' in the control vector will permit the storing of the corresponding element in the resulting vector. For example, 40005 stores a '1' so that C_5 is transformed into A_5+B_{-3}. The skewing effect is quite apparent in this example.

The mechanism to generate the desired output has to be explained further. After the instruction has been decoded at the stream unit, the appropriate microcode sequence in the Microcode Unit (MIC) will be initiated. This microcode unit resides in the stream unit and is responsible for vector type operations. The processor uses microcode to start up and shut down a vector instruction. The microcode is loaded in a read only memory (to users). When the CPU initiates an instruction requiring microcode control, it sends the F (function) code and a microcode pulse to the MIC. The latter then takes over control of the start up and termination of the instruction. In the case of interrupts, it also has to branch to save all the operands and parameters necessary to resume execution afterwards. Therefore it is the heart of the vector processing control. In fact, it is the central control once a vector-type instruction has been noticed via decoding. Typically it controls operations including:

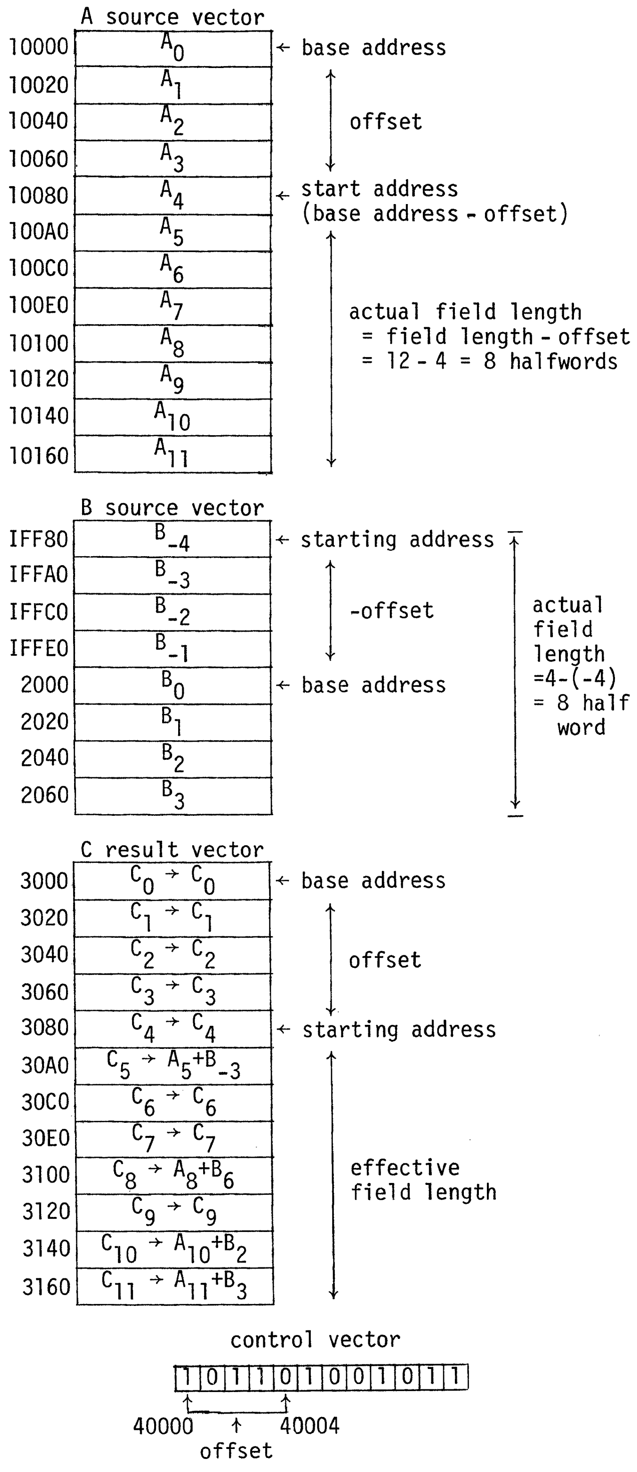

A source vector
10000 A_0 ← base address
10020 A_1
10040 A_2 offset
10060 A_3
10080 A_4 ← start address (base address - offset)
100A0 A_5
100C0 A_6
100E0 A_7 actual field length = field length - offset = 12 - 4 = 8 halfwords
10100 A_8
10120 A_9
10140 A_{10}
10160 A_{11}
B source vector
IFF80 B_{-4} ← starting address
IFFA0 B_{-3}
IFFC0 B_{-2} -offset
IFFE0 B_{-1}
2000 B_0 ← base address
2020 B_1
2040 B_2
2060 B_3
actual field length =4-(-4) = 8 half word
C result vector
3000 $C_0 \rightarrow C_0$ ← base address
3020 $C_1 \rightarrow C_1$
3040 $C_2 \rightarrow C_2$ offset
3060 $C_3 \rightarrow C_3$
3080 $C_4 \rightarrow C_4$ ← starting address
30A0 $C_5 \rightarrow A_5+B_{-3}$
30C0 $C_6 \rightarrow C_6$
30E0 $C_7 \rightarrow C_7$
3100 $C_8 \rightarrow A_8+B_6$ effective field length
3120 $C_9 \rightarrow C_9$
3140 $C_{10} \rightarrow A_{10}+B_2$
3160 $C_{11} \rightarrow A_{11}+B_3$
control vector
1 0 1 1 0 1 0 0 1 0 1 1
40000 40004
offset

Fig. 14 Example Vector ADD

(1) the reading of addresses from the register file (in the stream unit) for the vector parameters according to the designations specified in the instruction
(2) the calculation of the effective addresses, field lengths, etc. for monitoring the

starting the operations involved in the vector instruction
(3) the setting up of the usage of read-write buses as specified by the G (subfunction) field for the operands and results

and
(4) the transfer of addresses and other information to appropriate interrupt count registers wherever needed.

Once the effective addresses are computed, the operand elements will be fetched and paired for the operations involved, for example, going through the second level floating point pipe. The static configuration of the execution pipe will remain active until the vector instruction is terminated. A termination is marked by either of the following events:
(1) A vector is exhausted (e.g. when the effective field length is or has become zero, or the difference between the effective field length and the number of operand pairs encountered thus far is zero).
(2) Some other data fields or strings have been exhausted.

From the above description, one can grasp what a vector pipe really includes and how vectors can be processed in an overlapped manner. It is interesting to find out some other alternatives to achieve a vector pipe. So let us examine a similar vector machine, the TIASC system. The TIASC handles a vector instruction in a similar way, though some additional distinguishing features should be mentioned. To facilitate understanding, the central processor unit composition in TIASC has to be briefly explained. Its schematic diagram is provided in Fig. 5. It consists of 3 main components: (1) Instruction Processing Unit (IPU), (2) Memory Buffer Unit (MBU) and (3) Arithmetic Unit (AU). The IPU is analogous to the Stream unit in the STAR-100, MBU analogous to the load/store and AU plays the role of actual processing of data. In vector mode, the IPU fetches, decodes the instruction and calculates the effective addresses for the vector fields. After receiving the needed information from the IPU, the MBU starts fetching source operands and pairing them to be sent into the AU pipe (the AU can have one to four identical pipes). Each AU pipe has different configurations for performing different arithmetic operations (including integers) as in a typical static multifunctional pipeline. The two levels of pipeline action are quite apparent in this case.

A vector instruction in TIASC has some outstanding characteristics. Its instruction format can be as depicted in Fig. 15. However, rather

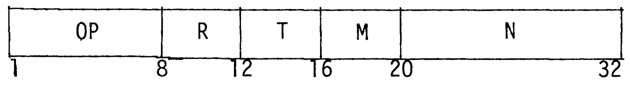


Fig. 15 Vector Instruction Format in TIASC

then specifying particular registers to fetch operand address and control information, some registers in the IPU have already been dedicated for vector processing, called the vector parameter file (VPF). It consists of 8 32-bit registers whose individual functions or interpretations have also been assigned permanently as drawn in Fig. 16. This fixed organization has the advantage that they can be hardwired to the input of the control ROM or other logic units for fast operation, without having to worry about access conflicts among them. The first register contains the operation code and the type and length of the vector considered (single or two-dimensional). Then the base address and the register containing the index (offset) are specified for each operand vector in the subsequent register in the VPF. The fifth and sixth registers are used to specify the increment for each vector and the number of iterations (field length) in this inner loop. For the outer-loop (two-dimensional vectors), similar information about the increments and number of iterations is included in registers seven and eight. The vector instruction, after having been decoded, will provide the information regarding whether the parameter file has to be loaded from main memory or retain some previous setting for immediate usage. If a load is needed, since the memory is

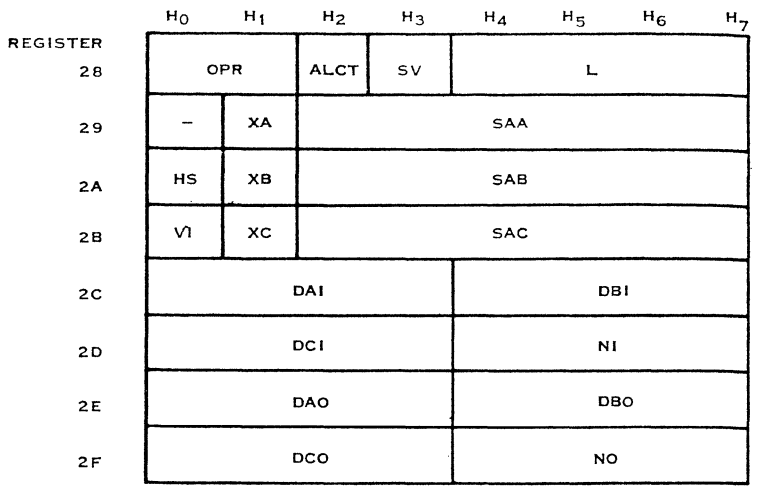


Fig. 16 Vector Parameter File Format in TIASC

inter-leaved, one memory cycle will be needed for VPF loading. The significance of this and the subquent additional activities will be examined more carefully in the next subsection. Afterwards, the sequence control in MBU takes over (as the MIC in STAR-100) the fetching of operands and the routing of operand pairs through the AU pipe.

So one observes that TIASC has at least two distinguishing features in vector processing:
(1) its dedicated use of the vector parameter file;
(2) the interpretation and usage of the VPF allow variable increments within different vectors concerned (contrary to the sequential mode in STAR-100) and two-dimensional vectors can be explicitly handled (inner and outer loops).

These features help to execute some vectors more efficiently and reduce the overhead that may have been incurred. Observe that once a vector instruction is initiated, the operand pairs are submitted to the AU continuously, in most cases, once per minor cycle (provided no severe memory interference results from other pipes or parts of the system or processor). Then the maximum throughput rate may be achieved (1 result per minor cycle = 60 nsec.). Also the sequence control for the AU is handled exclusively by the microcode stored in the ROM (read only to users) in MBU. Therefore the MBU serves as the unique interface between the IPU and the AU.

From the previous discussions, one can visualize the concept of vector processing and the two ways to achieve high throughput in two similar machines. The design of these processors really represents a complex and ingenious effort in pushing processor architecture, in both hardware and software aspects, to the very front of research and development. To bring out more interesting special features in these machines, the vector-type instruction set in the STAR-100 will be examined once again. From it, a final brief comparison between the two giants, STAR-100 and TIASC in this respect, will be derived.

Generally speaking, the CDC STAR-100 has a richer and more powerful vector instruction set. Two outstanding features are:
(1) Vector Macros
(2) Sparse Vector Instructions

In vector macro instructions, operations are similarly performed on the source vectors except that in some cases, no result vector is created. Then, instead, the result will be represented and stored in one or two registers as specified by the instruction.

For example, SELECT GE $A \geq B$, ITEM COUNT TO (C) involves: Comparing each element of vector field A with the corresponding one in B. The comparison will terminate if
(i) the condition $A_i \geq B_i$ is met for the current i, or
(ii) one of the vector fields is exhausted.
Then the number of operand pairs encountered thus far is stored in the register specified by C.

In this macro operation, control vectors can be used not only to prohibit the storage of result elements but also to disable the operation on some elements. In the example, even if $A_i \geq B_i$ is true for some i, if that comparison is disabled by the corresponding element in the control vector, execution will not be terminated. Thus using this kind of instruction, comparison of ordered vectors (e.g. lexicographic comparison) can be easily handled. The itemcount will be useful in some cases to indicate at which element the condition is satisfied. On the other hand, ordinary vector compare instructions also exist in the STAR-100 machine.

For example, COMPARE GE $A \geq B$, ORDER VECTOR $\rightarrow$ Z involves:
(1) Comparing the two vectors element by element.
(2) Storing 1 or 0 at the result vector elements depending on the satisfaction of the comparison condition.

Then the result of each pair-wise comparison will be recorded and available for later use, such as in sorting. Thus ordinary vector and vector macro instructions may form a powerful vector instruction set to be tailored to suit some application in mind as close as possible. With them, many quite complex sequential algorithms may turn out to be very effective which will be studied in another section.

The sparse vector instructions in the STAR-100 system further facilitate processing of large vectors with a lot of insignificant elements because then the latter can be packed easily into a sparse vector to be operated upon later. This can save both memory storage space and later effective processing time. A sparse vector can be formed using the following procedure as illustrated in Fig. 17.

Step 1: Generate an ordered vector using COMPARE instruction to indicate insigificant elements.
Step 2: Compress the vector into a sparse vector by storing the chosen elements from the former to memory, according to the ordered vector generated at step 1. The ordered vector has to be retained throughout the lifetime of the sparse vector to specify the positional significance of its elements.

After this, the sparse vector can be efficiently operated upon to generate desirable, interpretable results as in other vector instructions, with the help of the ordered vector. The advantages with sparse vectors should be emphasized: (1) the explicit hardware support for compaction of large vectors to reduce memory space needed, (2) if the sparse vector has to go through several operations or computation steps, effective processing time can be saved as well in that the operation on insignificant elements is no longer necessary, (3) if a variable increment for each vector (as in TIASC) is desired instead of in a sequential manner, one way to implement it is to use sparse vector instructions (though a more obvious way is to include the appropriate control vector) for the purpose of saving space and time.

While the TIASC does not include sparse vector instructions, its explicit two-dimensional

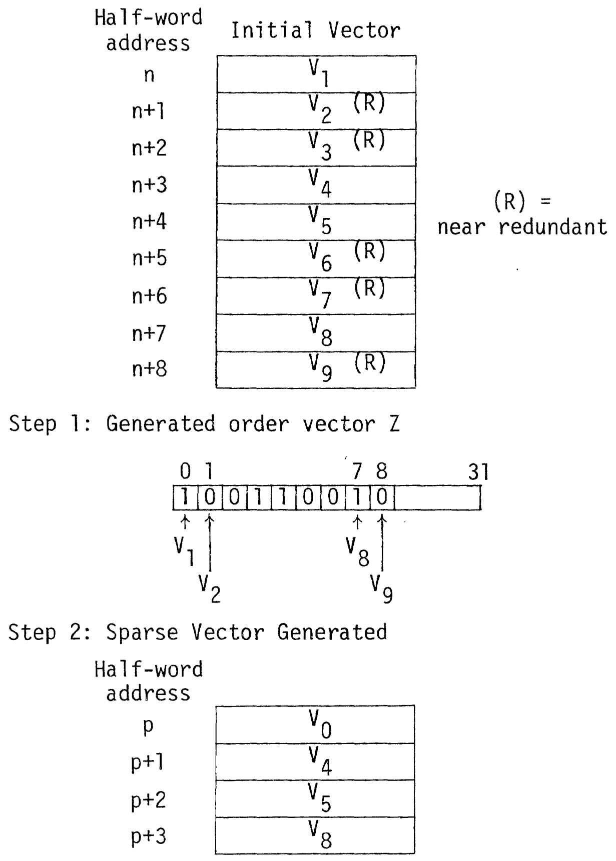


Fig. 17 Example Compression of Vector into Sparse Vector Field

vectors and variable vector increments are good features which promise high vector processing capability. Included in the vector instruction set of both machines are some very interesting and high level instructions such as Vector Search, Dot Product, Merge, Shift and Order instructions that allow programmers more power in developing their programs and the system to execute the algorithms implemented with the help of these advanced instructions more efficiently. The TIASC has also demonstrated how a 32-bit machine can cope with vector processing by efficiently making use of 8 bit opcode and the other relative fields, together with a dedicated vector parameter file. While the STAR-100 shows a stronger vector instruction set (a vector instruction is composed of 64 bits) because the F (function) and G (subfunction) codes can be used to specify more things, the vector parameters to be used can be assigned to any one of the registers (therefore not dedicated). It is hard to say which scheme is absolutely superior. And to summarize, the comparison between the vector processing powers of the TIASC and STAR-100 will be tabulated on the next page.

3.2 Implications, Requirements and Tradeoffs

How vectors can be processed has been demonstrated in the previous section. Now a closer look at some hidden or less conspicuous aspects in a vector machine is appropriate. From the previous description, one notices at least four things.

(1) There is some set-up time involved before executing a vector.
(2) Additional control in configuring the execution pipe and monitoring operand admission and traversal is needed.
(3) Richer instruction sets and clever compilers are pre-requisites to producing optimized code for vector machines.
(4) An intrinsic tradeoff between sequential and vector processing can be derived from the above considerations.

These four observations will be discussed and scrutinized here.

(1) Set-up Time and Flush Time. As demonstrated in the example TIASC and CDC STAR-100 systems, each vector instruction involves a set of vector parameter registers or control vectors to hold the information needed before the instruction can be initiated. The contents of these parameter registers are used to control the addressing operation and storage of result operands, as well as the final termination. In the example STAR-100 system, they will be used by the Microcode unit and later other buffers in the stream unit for initiation of operand fetches and execution continuously until a termination condition is detected by the microcode control. In the case of TIASC processor, they will be used by the IPU for address calculation, MBU for memory references and also by the microcode control (in MBU) for monitoring the subsequent execution activities. These parameter registers can be loaded from memory. In doing so, many additional memory fetches (register loading) have to be performed before the vector instruction can be started. This represents an overhead in time -- the set-up time. If the vector involved has a relatively short field length (therefore the number of iterations to be executed will be small), sometimes the set-up time may be comparable to the actual processing time of the vectors.

Besides the set-up time, there is another time measure of interest: the flushing time. The flushing time denotes the period of time between the initial operation (decode) of the instruction and the exit of the result (for vectors, the first result element) through the entire pipe. Therefore it directly measures the sum of the speeds of all the facilities that the instruction and an operand pair have to go through. Sometimes it is interesting to compare the flush times of a vector pipe and a sequential pipe. (Note, the flush time of a pipelined processor often is larger than that of its nonpipelined counterpart.) A vector pipe often has to perform more activities inside such as checking the termination condition, checking the control vector, etc. (though some of them can be overlapped with other operations). Therefore it will not be surprising to discover that a vector pipe may have a longer flush time than its sequential counterpart. However, this is insignificant if the vector field length is long, because then the execution time of the vector instruction will be dominated more likely by the field length as we will explore in the next paragraph. But for short vectors, this may be a disadvantage.

Compare and Contrast

STAR-100	TIASC
Vector parameter registers to be specified.	Vector parameter file fixed, therefore easy to reference and store.
Very strong vector instruction set.	Strong vector instruction set.
Sparse vector instruction included.	Sparse vector not included.
Vector increment is fixed.	Variable vector increment allowed.
Control vector introduces flexibility similar to the control unit in array processors. Can be used to implement variable vector increment.	No control vector used.
Explicitly speaking, vectors are only one-dimensional.	Two-dimensional vector explicitly accommodated. Computes 2 level loops effectively.
Use microcode control once a vector instruction is decoded.	Use microcode control to sequence each AU.
String unit and Floating Point unit (2 nonidentical pipes) will be responsible for most of the actual processing of data. Therefore, concurrency is among different execution units.	4 identical AU-MBU pairs can be installed to carry out all kinds of arithmetic operations (fixed or floating point). Concurrency of execution is among 4 identical pipes.
Floating point facility more powerful (e.g. Pipe 2 has fast divide, special multi-purpose segments).	AU has to be responsible for floating point operations (consists of 8 segments).
Requires set up time for vector processing.	Also requires set up time (though could be less because of the fixed VPF is easier to manage).

Here an endeavor will be made to compare sequential and vector pipeline processing in terms of time efficiency analytically. For a vector pipe, usually the memory operand supply rate is fast enough to meet the speed of the execution pipe(s). For example, in the TIASC system, the eight interleaved memory modules can maintain a total data transfer rate of 400M words per second, twice that required to support a central processor with four arithmetic unit pipes when processing vector instructions [13]. Therefore, analytically, for an effective vector field length of ℓ, the execution time of the vector instruction can be expressed as (assuming the bottleneck is in execution unit):

$$t_{vp} = t_s + t_{vf} + (\ell-1)t_e$$

where t_{vp} = vector instruction processing time

t_s = set-up time

t_{vf} = vector pipe flush time including decode, address calculation, operand fetch and paired, termination check and execution

t_e = the speed of the bottleneck of the execution unit pipe (in the case of TIASC, all 8 segments have the same speed, namely 1 minor cycle = 60 nsec)

Analogously, the same situation in a sequential pipe can be analyzed. Suppose the same instruction has to be executed on a vector in this case. Without vector processing power, this instruction has to be invoked ℓ times, that is, go through the entire pipe ℓ times. Even if the execution unit is fast enough here, most likely the fetching of operands can be less efficiently performed. (In vector machines, consecutive storage locations for operands will be fetched.) The processing time of the ℓ instructions may be expressed as:

$$t_{sp} = t_{sf} + (\ell-1)t_b$$

where t_{sp} = sequential (pipeline) processing time

t_{sf} = sequential pipe flush time

t_b = speed of bottleneck in the pipe, most likely in fetching operands if the execution unit is fast enough because more interference from unstructured memory references for instructions and operands results.

Comparing t_{vp} and t_{sp} yields:

$$t_s + t_{vf} + (\ell-1)t_e \leq t_{sf} + (\ell-1)t_b$$

iff

$$t_s \leq (\ell-1)(t_b-t_e) \quad \text{if} \quad t_{vf} \approx t_{sf} .$$

This reveals that if the vector length is reasonably large, vector processing is beneficial, considering the time advantage. If the set-up time is large compared to the difference of the speeds of the bottlenecks of the two pipes, then a large vector field length is needed to justify processing it in the vector form. Usually (t_b-t_e) will not be very much smaller than t_s (about 10 times), so that vector processing provides time efficiency in pipelined processors.

(2) Additional Control and Hardware. Vector pipes are designed to be cost-effective. They are implemented with sufficient flexibility and power to match the speed of an array processor (which usually is more expensive). For those vector machines with multifunctional pipes, additional control to establish the desirable configurations and routing the operands between pipe segments are needed. This is usually accomplished using microcoded control to allow flexibility and simpler circuitry. The hardware and firmware cost so introduced represents a portion of the cost of vector processing. These control functions sometimes are not very conspicuous but they do require a considerable amount of hardware support.

In addition, some other costs arise indirectly. The vector parameter file or registers represent part of the indirect hardware needed. Larger instruction sets to cope with vector processing also demand longer word lengths -- a result that affects the cost throughout the entire system. For smaller word length machines, one can try to get around the problem using techniques such as dedicated VPF in TIASC. Because of its cost effectiveness and speed advantages, vector processing power may prove adaptable to medium scale systems.

Buffering-wise, to keep up the execution speed, additional memory buffers (as the MBU) may be necessary to maintain an effective memory supply rate. Memory management problems, though out of the scope of this paper present a rich area to be explored for vector machines. All this direct and indirect control cost marks the space overhead incurred in vector processing and should be evaluated appropriately in tradeoff considerations.

(3) Richer Instruction Set and Clever Compilers.

Once the skeleton processor is designed, the instruction set has to be designed carefully. As in the case of STAR-100, suitable higher level vector macro and sparse vector instructions can be implemented (with proper hardware support) so that some application algorithms can be easily handled (fewer instruction and operand fetches and other conflicts). Without such well designed instruction sets, the power of the processor may depreciate many times because inefficient operations, redundant or excessive memory references and poorly utilized facilities may result.

Then the question arises: Since many of the rich instructions are by no means conventional, how to use them effectively in programs becomes a prime concern. For assembly language program writing, the user has to familiarize himself not only with the algorithm he is going to implement, but also with the details of these unconventional instructions first [19]. Because of the various architectural aspects involved, he has to choose a suitable algorithm carefully. Many a time, a fast (theoretical) algorithm will turn out to be inferior to some less effective serial algorithm because of the machine vector characteristic. As a simple example, consider sorting methods. In those vector machines, bubble sort will be quite inefficient because of the static multifunctional pipe involved. The bubbling of an item (compare and interchange) will incur too much reconfiguration cost, memory fetch overhead and set-up cost for the pipe. On the other hand, merge sort algorithms may be better because the machine can merge two ordered vectors in one pass without reconfiguration and additional set-up. As in the TIASC, the instruction vector ORDER A,B,C will try to compare element by element and store the smaller element in C until the entire ordering is accomplished. For example, if

$$A = 1,3,4,5,7,8,9$$
$$B = 2,3,5,8,10$$

then $C = 1,2,3,3,4,5,5,7,8,8,9,10$.

Therefore only a simple vector instruction is needed to merge sort two ordered vectors. Another good alternative is to find the peak value of an unsorted vector at every iteration, remove and store it at the appropriate place and repeat until the vector is completed sorted. It is easy to find the peak value of an unsorted vector by using instructions such as SEARCH and therefore this represents a better strategy (though quite similar) than the conventional bubble sort. This simple example discussion reveals how important it is to find the right algorithms to be implemented on these vector processors. The overlooking of architectural aspects may prove fatal in studying program efficiency.

Besides the direct program writing, each system also requires the installation of clever language processors to fully utilize its power. Additional optimization procedures should be incorporated to exploit its vector capability. For example, the optimized Fortran Compiler in TIASC was designed to produce highly optimized object code with complete diagnosis and messages. In general, the additional optimization included is accomplished by analyzing the source program logic and performing optimization on the object code instructions involved. Vector instructions will be used wherever feasible and scalar operations are reordered wherever possible to reduce pipeline reconfiguration and memory reference delays (8-way interleaved memory system). Therefore the compiler not only can recognize array (vector) oriented operations in DO-loops but also can reorder some scalar operations generated to meet the architectural characteristics of the machine. Of course the other more conventional optimization procedures are also included, such as elimination of redundant subexpressions, removal of constant assignment statements in a loop and proper register assignment, etc. This burden on compiler designers is quite heavy. Thus the software cost for vector processing is an important item not to be omitted.

(4) Tradeoff Summary. In this section, we have revealed both the time and the space overhead needed in vector processing as compared to a sequential pipelined processor (such as the IBM 360 model 91). The advantages of vector processing are its speed improvement for reasonably long vectors and the more orderly management to better utilize the memory system and other resources when dealing with vectors. The costs it incurs are the needed firmware control and additional software facilities to utilize its power. When the latter have been solved successfully at less cost, vector

processing may be generalized and applied to smaller scale processing systems as well.

IV. An Example Buffer Free Pipeline Design -- Multipliers

Multi-level pipelining action can be implemented in a processor, as revealed in some detail in the previous sections. Before ending the analysis, a closer study of some typical low level pipelines in processors may be of interest, just to understand the physical limits in applying pipeline discipline in computer systems.

As a first step for implementing a pipeline, an algorithm for the function (process) to be implemented has to be chosen. The algorithm should demonstrate sufficient parallelism to allow repeated iterations or new inputs to proceed as fast as possible. To back up this assertion, here pipelined multipliers will be studied carefully. Observe that a good algorithm will yield a good reservation table with the smallest initiation time intervals or shortest execution time for some specific sequences.

The most common method of multiplication is the pencil and paper algorithm in which the multiplicand will be shifted and, if the corresponding bit in the multiplier is 1, added to the partial sum until the multiplier is exhausted. Clearly this is not an effective pipeline algorithm because too much shifting and adding (complete additions) are needed. Even if the 0's in the multiplier are skipped, the speed of the multiplier is too slow to match the speed of the other parts of the system.

One could try to build a very fast adder such as the Wallace Tree [20] of carry-save adders (CSA). But such implementation requires too much hardware support. Obviously a speed-cost tradeoff exists here. The method favored in the IBM 360 model 91 was a hybrid method [21]. Intuitively, a carry-save adder tree is still used as shown in Fig. 18.

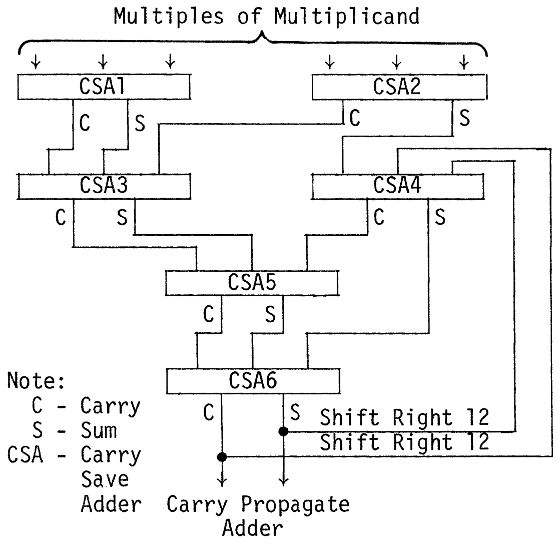
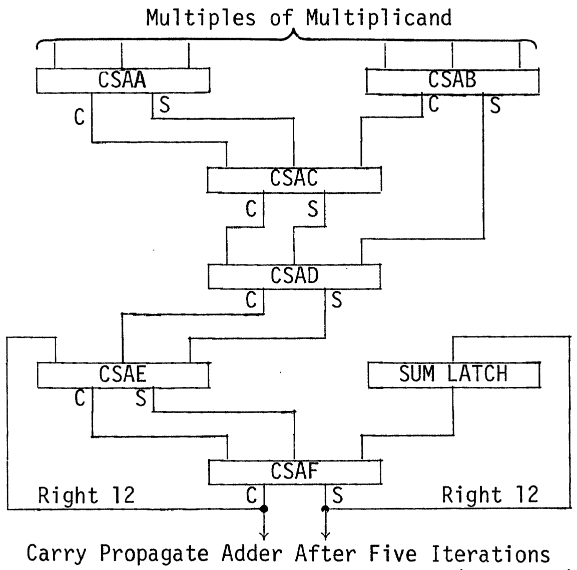


Fig. 18 CSA tree for multiplication using iterations

At each iteration, 12 bits of the multiplier are retired by generating the corresponding six multiples (each corresponds to 2 bits of the multiplier) of the multiplicand and admitting them to the CSA tree. Therefore five iterations are needed to exhaust the 56 bits of the multiplier. Here, however, the iterations cannot be effectively pipelined because in each pass, the six multiples have to go through CSA6 and loop back to CSA4 to synchronize with the next iteration. In this example, the next iteration must thus wait for 3 levels of CSA delays for synchronization (for the previous iteration to route through CSA3, CSA4, CSA5 and CSA6). To improve it, the adopted pipelined design is drawn in Fig. 19a and its schematic diagram in Fig. 19b. Four stages are



Carry Propagate Adder After Five Iterations
Fig. 19a Pipelined CSA Multiplier (Model 91)

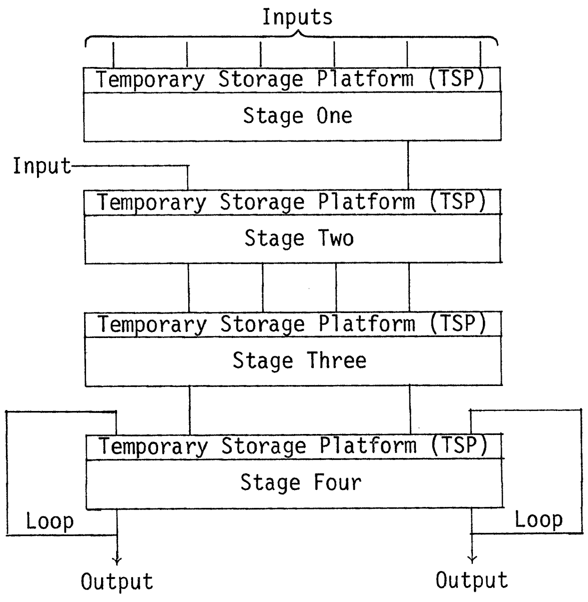

Inputs
Temporary Storage Platform (TSP)
Stage One
Input
Temporary Storage Platform (TSP)
Stage Two
Temporary Storage Platform (TSP)
Stage Three
Temporary Storage Platform (TSP)
Stage Four
Loop
Loop
Output
Output

Fig. 19b Schematic Diagram

identified with a loop at the last stage which consists of two CSA. Therefore the clock delay is two levels instead of three in the previous design. The loop in this case is used to add the result of the iteration (right shifted 12 bits) to that of the next iteration (for the next set of 12 bits) until the partial sum and carry finally emerge (at the end of five iterations) to be added together for the final output. A timing diagram is drawn in Fig. 20 to illustrate the sequence of actions involved. At time 0, the first input I_1 (for the first 12 bits) is gated into stage 1. At the next interval, I_1 is gated into stage 2 while I_2 into stage 1. This process repeats until I_5 has exited from stage 4 when the final product is accumulated. Observe that latches have to be inserted between stages to tolerate concurrent processing. The total iterative tree requires 8 clock periods = 16 levels of CSA delays. If the multiple generation process is fast enough (to allow the iterations to proceed), a multiplier with a speed equal to 3 cycles (60 nsec each) can be built.

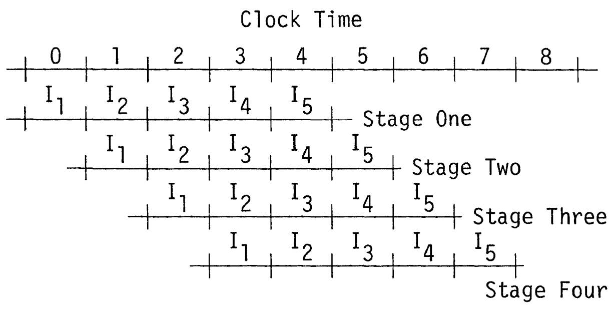


Fig. 20 Timing of the multiplier in Fig. 19

The intrinsic requirement here is that the multiples of the multiplicand can be generated fast enough. However, this need not be possible. Sometimes simple full adders may be needed, in which case, the propagation delay is considerable. So, in small machines, maybe some other alternatives can be considered as well. Here, an effort will be made to describe two of them.

The first observation is that the generation of multiples can be made simple if a simple shifting after table looking (decoding) is involved. For example, 1100 × M represents 16M - 4M, 0111 × M represents 8M - M. Then the generated (shifted) multiples can be added as before. The second observation is that if the clock period can be reduced from two levels of CSA to one level, the required iterations may be completed faster.

The first scheme to be described also uses CSA's to generate the partial sum and carry of the iterations involved (CSA has the advantage over full adders in that it need not carry propagation inside). The configuration is drawn in Fig. 21. Each multiplier will be grouped into $g_1,\dots,g_m$ each containing 4 bits. A decoded and shifting network is required in addition to the CSA's. For the current g_i, the corresponding multiples of the multiplicand will be generated. The three multiples (16M, ±8M⊕±4M, ±2M⊕±M) are admitted to the first CSA at the first clock (= 1 level CSA delay).

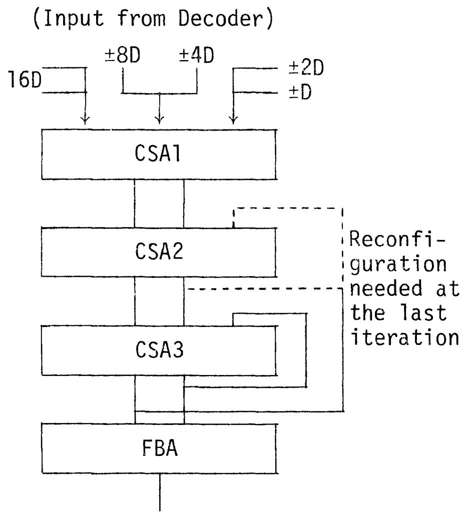


Fig. 21 Fast piped multiplier (Rate = 1 CSA delay)

Iterations repeat until m groups in the multiplier are exhausted. Then after the last iteration (g_m) has passed through CSA3, the loop from its partial sum output is reconfigured to feed to the input of CSA3 (instead of CSA2 in previous intervals). This serves to synchronize the flow to produce a final partial sum and carry to the full binary adder.

The CSA pipe has an ideal throughput rate because input is allowed at every clock interval. For a smaller machine, if m is equal to 6, then the total delay in the tree for 6 iterations is only 6+3 = 9 levels of CSA delays. Also the multiple generation at the beginning is much simplified -- since also a simple decode of g_i is needed to shift the operands to the corresponding positions. If the shifting hardware is fast enough, a very fast multiplier using the pipeline concept to exploit the parallelism among iteration has been created.

The necessity of reconfiguration at the last iteration in the previous example may be an undesirable feature. A modified scheme using three loops as shown in Fig. 22 will eliminate this requirement. Then the multiples are admitted into the pipe as before with a rate equal to one level of CSA delay. Using this arrangement, the total delay for 6 iterations is 6+3 = 9 levels of CSA, the same as the previous scheme.

These pipelined multiplier examples have demonstrated some important facts. In the first place, the choice of the algorithm is quite important in determining the efficiency of the design. Then, strategies must be developed in trying to reduce the clock-interval and hence improve the speed of the pipeline. In so doing, extra care must be taken in matching the speeds of the different segments of the pipe, for example, the loop back that may create 3 or 2 levels of delay. The physical limit of applying pipeline principles is also exposed. Since pipelining requires the insertion of latches among segments (to avoid circuit race conditions, etc.), the latter

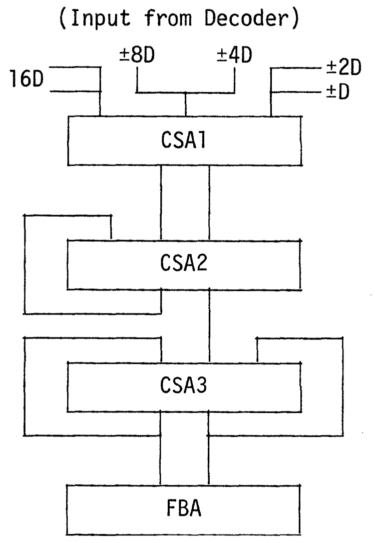


Fig. 22 Improved Pipelined CSA Multiplier (Rate = 1 CSA delay)

have finite delays and therefore mark the physical upper limit of throughput rate in any pipeline.

V. Conclusion

Pipelined processors represent a clever approach to speed up instruction processing when the memory access time has improved to a certain extent. Without having to duplicate the entire processors n times, a throughput rate of close to n times improvement over a nonpipelined case may be achieved. To make this possible, certain problems have to be solved including: parallelism and busing structure, handling of unexpected events and efficient sequence control with well-designed instruction set. Special vector processing capability is one way to specify parallelism in programs easily. These problems and solutions are discussed and solutions in existing machines illustrated. The multi-level application of pipeline discipline is promising in upgrading the performance of a processor, especially from a cost-effective point of view and certain deserves future investigation to generalize its application to even smaller scale systems.

References

[1] T.C. Chen, "Unconventional superspeed computer systems," AFIPS SJCC 1971, pp. 365-371.

✓ [2] D. McIntyre, "An introduction to the ILLIAC IV computer," Datamation (April 1970), pp. 60-67.

[3] A.J. Evensen and J.L. Troy, "Introduction to the architecture of a 288-element PEPE," Proc. 1973 Sagamore Conf. on Parallel Processing, pp. 162-169.

[4] J.A. Rudolph, "A production implementation of an associative array processor - STARAN," AFIPS FJCC 1972, pp. 229-241.

[5] O.E. Marvel, "HAPPE - Honeywell Associative Parallel Processing Ensemble," Proc. Symp. on Computer Architecture (Dec. 1973), pp.261-268.

✓ [6] D.C. Stanga, "Univac 1108 multiprocessor system," AFIPS SJCC 1967, pp. 67-74.

✓ [7] T.C. Chen, "Parallelism, pipelining and computer efficiency," Computer Design (Jan. 1971) pp. 69-74.

✓ [8] C.V. Ramamoorthy and H.F. Li, "Efficiency in generalized pipeline networks," AFIPS NCC 1974, pp. 625-635.

[9] D.W. Anderson, F.J. Sparacio and R.M. Tomasulo, "IBM System 360 Model 91, machine philosophy and instruction handling," IBM J. Res. and Develop. (Jan. 1967), pp. 8-24.

✓ [10] W.J. Watson, "The TIASC - a highly modular and flexible super computer architecture," AFIPS FJCC 1972.

[11] R.G. Hintz and D.P. Tate, "Control Data STAR-100 processor design," COMPCON 72.

[12] R.M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," IBM J. Res. and Develop. (Jan. 1967), pp. 25-33.

[13] Texas Instruments Inc., "A description of the Advanced Scientific computer system," (April 1973).

✓ [14] Control Data Corp., "Control Data STAR-100 computer hardware reference manual," (1974).

[15] Texas Instruments Inc., "The ASC system - Central Processor," Austin, Texas (Dec. 1971).

[16] E. Davidson, "The design and control of pipeline function generator," Stanford Report (1972).

[17] E.S. Davidson, L.E. Shar, A.T. Thomas and J.H. Patel, "Effective control for pipelined computers," COMPCON 75, pp. 181-184.

[18] C.V. Ramamoorthy and H.F. Li, "Sequencing control in multifunctional pipelined systems," Proc. 1975 Sagamore Conf. on Parallel Processing.

[19] T. Kishi and T. Rudy, "STAR TREK," COMPCON 75, pp. 185-188.

[20] C.S. Wallace, "A suggestion for a fast multiplier," IEEE Trans. Elec., EC-13 (1964), pp. 14-17.

[21] S.F. Anderson, J.G. Earle, R.E. Goldschmidt and D.M. Powers, "The IBM System/360 Model 91: floating-point execution unit," IBM J. Res. and Develop. (Jan. 1967), pp. 34-53.

MULTIPROCESSOR ARCHITECTURE - A SURVEY

Philip H. Enslow Jr.
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

(Invited Paper)

Abstract -- Multiprocessors are defined as a class of computer system in terms of both hardware and operating system organization. The hardware organization is characterized by the nature of the interconnection system used between the primary functional units, memory, processors, and input/output. The three basic interconnection systems are time-shared bus, crossbar switch, and multiport memory. The three basic organizations for operating systems are master-slave, separate executive for each processor, and symmetric treatment of all processors.

Introduction

Motivation

There are four general levels at which improvements in system performance can be made:

- Devices and circuits: the basic hardware speed.
- System architecture: the algorithms implemented in the functional units --- processor, control, memory, and input/output.
- System organization: the method of interconnecting the functional units.
- System software: the speed and efficiency of the operating system, translators, and other supporting software.

Multiprocessors are a special class of system organizations supported by appropriate system software.

Two methods that have been utilized to varying degrees to improve system performance are concurrency and simultaniety:[a]

- o *Concurrent* execution of several different programs --- Multiprogramming.
- o *Concurrent* memory operations --- interleaved memory.
- o Execution of I/O operations *simultaneously* with CPU operations --- overlapped I/O.
- o Multiple, *simultaneous* I/O operations.
- o Multiple, *simultaneous* operations in the processor unit(s)
  - oo Replicated processor units --- multiprocessing, etc.
  - oo Fragmented or segmented algorithms --- pipelining.
- o Multiple, *simultaneous* memory operations --- associative processing.

It is by the use of all of these techniques that parallelism is introduced into the system.

[a] To refresh your memory: "concurrent events" occur during the same *interval* of time; "simultaneous events" occur at the same *instant* of time.

Multi-Computer Systems

There are a number of multi-computer systems that are not multiprocessors. An obvious example is the system with a stand-alone peripheral or satellite processor. Perhaps less obvious are the various forms of coupled systems, both loosely and closely coupled, such as the IBM ASP (Attached Support Processor) System and others having direct electrical connections. Specific examples and details are given in Enslow [1].

Definition of a Multiprocessor

A multiprocessor is defined in the American National Standard "Vocabulary for Information Processing" as "A computer employing two or more processing units under integrated control". That definition is good as far as it goes, but that is not far enough! The comment about "integrated control" is extremely important, for a multiprocessor *must* have a single integrated operating system. What has not been covered by the ANSI definition are the concepts of sharing and interaction which are at the central core of the philosophies of multiprocessing.

With respect to the hardware, the system must have the capability for sharing of main memory by all processors (arithmetic/logic unit and control unit only) and the sharing of input/output devices by all memory and processor combinations (Figure 1). There are some qualifications on the requirement for sharing of *all* of the resources of any one type, but those will be covered below.

The important aspect of interaction is the level at which it occurs. In the multi-computer systems mentioned above, the level of interaction is the complete file or data set. Interaction is basically an I/O transfer. The operational level of interaction allowed must be more flexible. In a multiprocessor this level must be allowed to descend to the lowest level. Interaction must be possible at the physical levels of files, data sets, and even data elements. From the operational point of view, interaction must be possible at the level of complete jobs, tasks, and individual steps.

It is the combination of these expanded concepts of sharing and of interaction at all levels that completely characterizes the hardware and software required to provide a true multiprocessor.

- o A multiprocessor contains two or more processors of approximately comparable capabilities.
- o All processors share access to common memory.
- o All processors share access to input/output channels, control units, and devices.
- o The entire system is controlled by *one* operating system providing interaction between processors and their programs at the JOB, TASK, STEP, DATA SET, and DATA ELEMENT levels.

Multiprocessor System Organizations

There are only three basically different organizations used for multiprocessors. These are characterized by the nature of the interconnection sub-system:

- Time-Shared Bus.
- Crossbar Switch
- Multiport Memory

These are each discussed in turn. Other system organizations that have been utilized to achieve parallelism are

- Asymmetrical or Nonhomogeneous
- Array or Vector Processors
- Pipeline Processors
- Fault-Tolerant Systems
- Associative Processors

Several of these are discussed in other invited papers in this series and in other. References [1,2].

Time-Shared/Common-Bus Systems

The simplest organization for any system, multi- or not, is to establish a common communication path and connect all of the functional units to it. This has been done to assemble some simple multiprocessors (Figure 2). They are "simple", for the inter-connection sub-system can be merely a multi-conductor cable. It is often a totally passive unit, i.e., it has no active components such as switches or amplifiers. Transfer operations are controlled completely by the bus-interfaces of the sending and receiving units. The unit wishing to initiate a transfer, e.g., a processor or I/O unit, must determine the availability status of the bus, address the destination unit, determine its availability and capability to receiving the transfer, notify the destination what to do with the data being transferred, and then initiate the transfer. A receiving unit has only to recognize its address and respond to the control signals from the sender. It is not really that simple, but those are the basic concepts. (The single bus in the PDP-11, the UNIBUS, has 56 lines to provide the control lines and data paths necessary to transfer words of only 12 bits.)

To add or remove functional units, the hardware changes are quite minimal, in fact, almost nothing. The units in the system must know what other units are present and their unit and internal location addresses, but that is basically a software problem. The interconnection subsystem is quite reliable by its very nature, and it is low-cost.

However, all of these benefits do not occur without other costs. The most important of these is the serious limitation on overall system performance that results from having only one path for all transfers. Interconnection techniques that overcome this weakness add to the complexity of the system.

The first step might be to provide two, one-way paths (Figure 3). The complexity is not increased very much, nor is the reliability diminished substantially; however, a single transfer operation usually requires the use of both buses, so not very much is gained .

The next step is to provide multiple two-way buses (Figure 4). Now there can be multiple, simultaneous transfers; but the complexity has greatly increased. No longer is the interconnection sub-system a totally passive unit. Logic, switching, and other control functions must be associated with each point at which functional units are attached to the transfer buses.

Crossbar Switch System

If the number of buses in a shared-bus system is increased, the point is reached where there is a separate path available for each memory box (Figure 5). The interconnection subsystem is then a "non-blocking" crossbar. The adjective non-blocking is usually omitted since it is a characteristic of the crossbar switches used in multiprocessor systems that they are "complete" with respect to the memory units, i.e., there is a separate bus associated with each memory and the maximum number of transfers that can take place simultaneously is limited by the number of memory boxes and not by the capacity of the switch.

The important characteristics of a system utilizing a crossbar interconnection matrix are the extreme simplicity of the switch-to-functional unit interface and the ability to support simultaneous transfers for all memory units. To provide these features requires major hardware capabilities in the switch. Not only must each cross-point be capable of switching complete parellel transmissions, but it must also be capable of resolving multiple requests for access to the same memory module occurring during a single memory cycle. These conflicting requests are usually handled on a pre-determined priority basis, e.g., I/O first, P_2 has primary access priority to M_2, etc. The result of this is that the hardware required to implement the switch can become quite large and complex. An example that has been cited is a system with 24 each 32-bit processors and 32 memory units. The number of circuits required in the switch matrix would be two to three times the number required for an IBM S/360 Model 75.

A characteristic of somewhat lesser importance that can be significant in specific instances is the capability to expand the size of the system by merely increasing the capacity of the switch. There are no changes required in any of the functional units because of the very simple interfaces utilized, and often the switch is designed so that its capacity can be increased merely by adding additional modules of cross-points. Note, that this discussion of expansion has addressed only the hardware. The modification of the operating system to support the larger system may often prove to be extremely difficult; however, this is true for all multiprocessor system organizations.

In order to provide the flexibility required in access to the input/output devices, it is a natural extension of the crossbar switch concept to use a similar switch on the device side of the I/O processor or channel (Figure 6). The hardware required for the implementation is quite different and not nearly so complex for controllers and devices are normally designed to recognize their unique addresses. The effect is the same as if there were a primary bus associated with each I/O channel and cross buses for each controller/device.

Multiport Memory Systems

If the control and switching logic that is distributed throughout the crossbar switch matrix is concentrated in the memory units, a multiport memory system results (Figure 7). This system organization is well suited to both uni- and multiprocessors, and it is used in both. The method often utilized to resolve memory access conflicts is to permanently assign specific priorities to each memory port. The system can then be configured as necessary at each installation to provide the appropriate priority access to various memory boxes for each functional unit. Except for the priority associated with each, all of the ports are electrically and operationally identical. In fact, the ports are often merely a row of identical cable connectors, and electrically it makes no difference whether an I/O or central processor is attached. A system which utilizes 8-port memory units may have any mixture of processor and I/O units subject to the restrictions that there must be at least one of each and the total be eight or less.

The flexibility possible in configuring the system also makes it possible to designate portions of memory as "private" to certain processors, I/O units, or combinations thereof (Figure 8). This organization can have definite advantages in increasing security against unauthorized access. It may also permit the storage of recovery routines in memory areas that are not susceptible to modification by other processors. There are also serious disadvantages in other processors not being able to access control and status information in a memory block associated from a failed processor.

The multiport memory system organization can also support non-blocking access to the memory if a "full-connected" topology is utilized. It will also permit the exploitation of interleaved memory addresses for access by a single processor, However, for multiple processors, interleaving may actually degrade memory performance by increasing the number of conflicts. With multiple processors it is usually preferable to utilize the property of "locality of reference" and not attempt to increase the effective memory speed by interleaving.

Comparison of The Three Basic System Organizations

A number of factors can be considered in comparing the three basic organizations described above or evaluating their use in specific applications. The most obvious are cost, flexibility, growth potential, and system throughput capacity.

Time-Shared Bus:

- o Lowest overall system cost for hardware.
- o Least complex. The interconnection bus may be totally passive.
- o Very easy to physically modify the hardware system configuration by adding or removing functional units.
- o The overall system capacity is limited by the bus transfer rate. This may be a severe restriction on overall system performance.
- o The failure of the bus is catastrophic.
- o Expanding the system by the addition of functional units may degrade overall system performance (throughput).
- o The system efficiency attainable (based on the simultaneous use of all available units) is the lowest of all three basic interconnection systems.
- o This organization is usually appropriate only for smaller systems.

Crossbar:

- o This is the most complex interconnection system.
- o The functional units are the simplest and cheapest since all of the control and switching logic is in the switch.
- o Because a basic switching matrix is required to assemble any functional units into a working configuration, this organization is usually cost-effective only for multiprocessors.
- o There is the potential for the highest total transfer rate.
- o System expansion (addition of functional units) usually improves overall performance.

- o There is the highest potential for system efficiency.
- o There is the potential for system expansion without reprogramming of the operating system being required.
- o Basically, expansion of the system is limited only by the size of the switch matrix which can be modularly expanded within engineering limitations.
- o The reliability of the switch, and therefore the system, can be improved by segmentation and/or redundancy within the switch.

Multiport Memory:

- o Requires the most expensive memory units since most of the control and switching circuitry is included in the memory unit.
- o The characteristics of the functional units permit a relatively low-cost uniprocessor to be assembled from them.
- o There is a potential for a very high total transfer rate in the overall system.
- o The size and configuration options possible are determined (limited) by the number and type of memory ports available. That design decision is made quite early in the overall design process and is difficult to modify.
- o There is a large number of cables and connectors required.

System Software

It is difficult to determine how much should be said about system software for the types of machines being discussed here. There is conceptually little difference between the system software requirements of a multiprocessor and those for any other large system utilizing multiprogramming. When the functional capabilities required in the operating system are listed

- Resource allocation and management
- Table and data set protection
- Prevention of system deadlock
- Abnormal termination
- I/O load balancing
- Processor intercommunication
- Processor load balancing
- Reconfiguration

only the last three may be thought of as unique to multiprocessor systems. Many of the problems to be solved in providing the common capabilities may be more difficult to solve because of the additional processor(s) present in the system; however, the effective utilization of these additional resources makes it even more important that efficient solutions be found. Otherwise poor performance by the operating system will destroy any cost-performance advantages that the system might have. The efficiency of the operating system becomes much more important in a multiprocessor system.

There are a few special problems that appear in multiprocessor and other parallel systems. One of these is the importance of short-term scheduling. Anomalies may occur if there are only a few jobs to be scheduled and the order in which they are chosen is "incorrect" (Figure 9c). However, if there is a large amount of work waiting for the system, such short-term effects will not affect the total productivity of the system.

Organization of Multiprocessor Operating Systems:

There are three organizations that have been utilized in the design of operating systems for multiprocessors:

- Master-slave
- Separate executive for each processor
- Symmetric or anonymous treatment of all processors

For most multiprocessors, the first operating system available usually operates in the master-slave mode. This is certainly the easiest type to implement and may often be produced by making relatively simple extensions to a uni-processor operating system that includes full multiprogramming capabilities. The master-slave type of system is simple, but it is usually quite inefficient in its control and utilization of the total system resources. It is not clear which of the other two system organizations is the best from a performance point of view; however, there appears to be evidence that both are superior to master-slave.

An operating system operating in the master-slave mode has the following characteristics:

- The executive routine is always executed in the same processor. If the slave needs service that must be provided by the executive, then it must request that service and wait until the current program on the master processor is interrupted and the executive is dispatched. The executive and the routines that it uses do not have to be reentrant since there will be only the one processor using them.
- Having a single processor executing the executive also simplifies the table conflict and lock-out problem for control tables.
- The entire system is subject to catastrophic failures that require operator intervention to restart when the processor designated the Master has a failure or irrecoverable error.
- The overall system is comparatively inflexible.
- Idle time on the Slave system can build-up and become quite appreciable if the Master cannot execute the dispatching routines fast enough to keep the Slave busy.

- This type of system requires comparatively simple software and hardware.
- This type of operating system is most effective for special applications where the work load is well defined or for asymmetrical systems in which the Slaves have less capability than the Master processor.

When there is a separate executive (distinct copy) operating in each processor the characteristics are quite changed:

- Each processor services its own needs.
- It is necessary for some of the supervisory code to be reentrant or replicated to provide separate copies for each processor.
- Each processor (actually each executive) will have its own set of private tables, although there are some that must be common to the entire system and will create table access control problems.
- It is not as sensitive to a catastrophic failure as the Master-Slave system; however, the restart of an individual processor that has failed will probably be quite difficult.
- In effect; each processor (executive) has its own set of I/O equipment, files, etc.
- Because of the point immediately above, the reconfiguration of I/O usually requires manual intervention and possibly manual switching.

To treat all the processors as well as all other resources symmetrically or as an anonymous pool of resources is the most difficult mode of operation; however, the resulting benefits may be worth the trouble:

- The "master" floats from one processor to another, although several of the processors may be executing supervisory service routines at the same time.
- This type of system can attain better load balancing over all types of resources.
- Conflicts in service requests are resolved by priorities that can be set statically or under dynamic control.
- Most of the supervisory code must be reentrant since several processors can execute the same service routine at the same time.
- Table access conflicts and table lock-out delays can occur, but there is no way to avoid this with multiple executions. The important point is that they must be controlled so that system integrity is protected.
- The potential advantages that can accrue with this type of operating system operation are:
  - o It provides graceful degradation
  - o It can provide better availability of a reduced capacity system
  - o The system provides true redundancy
  - o It makes the most efficient use of the resources available.

It must be emphasized that most operating systems for multiprocessors are not "pure" examples of any one of the three classes discussed above. The only generalization that is possible is that the first system produced is usually of the Master-Slave type and the ultimate being sought is of the Symmetric type.

Today and Tomorrow

Current Situation

There is no question that multiprocessors and other forms of parallel computing systems are accepted types of organizations. A table prepared by the author in mid-1973 listed the characteristics over 50 such systems.[1] Since that date, almost every manufacturer has announced further systems that fall into this category. The major factors governing future expansion of the concepts are performance and cost-effectiveness.

System Performance

There have been very few careful studies of the productivity increase attained by adding another processor. These studies are very difficult to perform in an accurate manner, for there are too many variables present. One example cited in the author's book gives a factor of 1.8 for a two-processor system but only 2.1 for three. Much of this non-linearity is probably due to the operating system, and dramatic improvements have been attained by simple changes in routines like the dispatcher. One manufacturer developed formulae to predict system performance improvements; however, these are also quite suspect.

Cost-Effectiveness

Cost-effectiveness comparisons between comparably sized systems of different manufacturers are very unreliable since the system price often does not reflect its tru cost in any dimension. In fact, it is doubtful that such comparisons between systems produced by the same manufacturer are even valid. Figure 10 illustrates some data that was derived from real systems costs and performances a few years ago. The starting points for cost and performance of the Mod 1 system are taken as unity. A fully expanded Mod 1 will provide almost double the performance at about 1.7 times the basic cost. A large gap then exists between the Mod 1 and Mod 2 system which provides over 3.4 times the performance at 2.2 times the cost. What happens if the Mod 1 processors are used in a multiprocessor configuration? The cost-performance curves are always below the standard Mod 1 uniprocessor due to the extra cost of the hardware features that permit multi-processing; however, when the capabilities of the single Mod 1 processor are saturated, then another is added to the system along with other equipment such as additional memory, and the cost-performance curve continues upwards in a fairly smooth manner. When necessary, a third processor can be added to continue this trend. With the multiprocessor system there are distinct advantages:

- it is possible to provide a smooth growth in system performance capabilities avoiding the large jump from a Mod 1 to Mod 2 system;
- there is no requirement for a major system change-over; and
- there is a definite performance range in which the multiprocessor provides better cost-performance.

The disadvantages are equally obvious:

- the multiprocessor provides lower cost-performance for most workload levels (this is the price that is paid for the reliability/availability improvements of the multiprocessor); and
- the ability to continue to expand the multiprocessor system change-over will finally be necessary at a much higher workload level.

Figure 11 is not accurate in the specific numbers and ratios displayed; however, it does display a condition that has occurred with a recently introduced system. The Mod A multiprocessor is little bit more costly than the uniprocessor Mod A; however, it does have the ability to expand past the limits of the single Mod A. In fact as the Mod A multiprocessor expands, it provides a cost-performance factor that is better than the next larger model from the same manufacturer, the Mod B. The logical questions then is why stop at a two-processor multiprocessor. The answer is that that is the limit of the hardware software configurations supported for the Mod A, and the user is forced to change to the Mod B system family.

The Future

Some have said that the general-purpose uniprocessor is reaching the limits of its capabilities. This will probably happen, but it does not appear that the limit has been reached yet. What is true is that the use of multiple processors often provides an easier, and often cheaper, method to increase performance.

Several studies of the future for data processing hardware presents arguments that large future systems will be both multiprocessors, as defined here, as well as asymmetric systems with several levels of processors in the systems each devoted to a hierarchy of functions such as main processing, file handling, physical input/output control, etc. This appears to be a much more probable picture for the future. It may be necessary to produce a new descriptive term to differentiate these systems from the classic multiprocessors as described here. One such term that has been coined is "polyprocessor". There is no question that the future will find multiprocessing and the other concepts of parallel processing in much wider use than the present.

References

[1] P.H. Enslow, ed., Comtre Corp., Multiprocessors & Parrallel Processing, John Wiley, New York, (1974), 340+xii pp.

[2] P.H. Enslow, "Multiprocessors and other parallel systems: An Introduction and Overview," Multiprocessors --- State-of-the-Art Report Infotech, Maidenhead, England, (to be published).

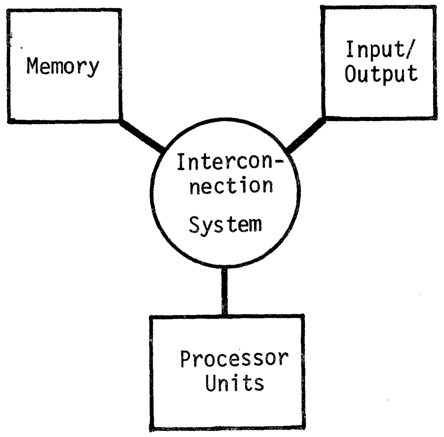


Fig.1: Basic Multiprocessor Organization.

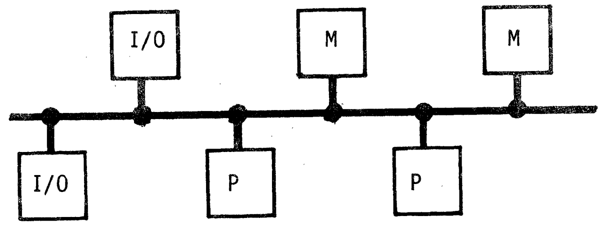


Fig.2: Time-Shared Bus System Organization --- Single Buss

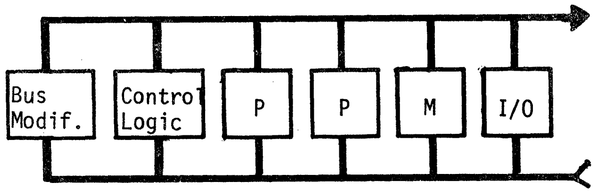


Fig.3: Time-Shared Bus System Organization --- Uni-Directional Buses

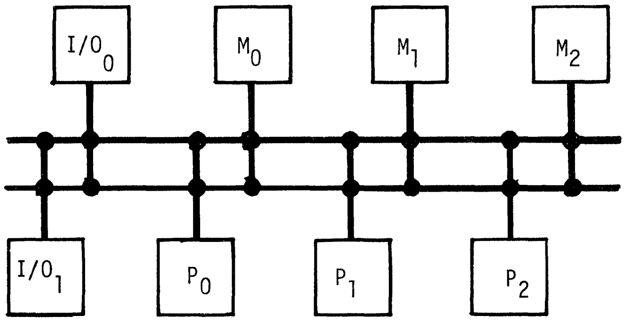


Fig.4: Time-Shared Bus System Organization --- Multiple Buses

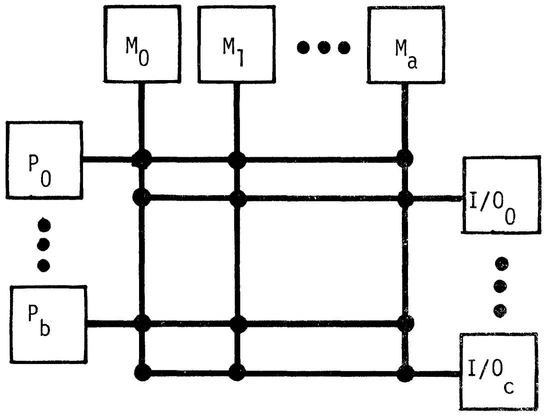


Fig.5: Crossbar Switch System Organization

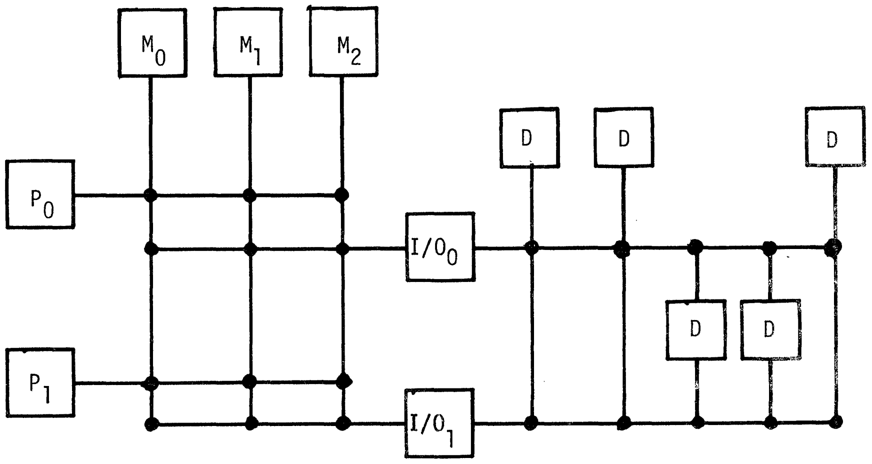


Fig.6: Crossbar Switch System Organization with I/O Crossbar Switch Matrix

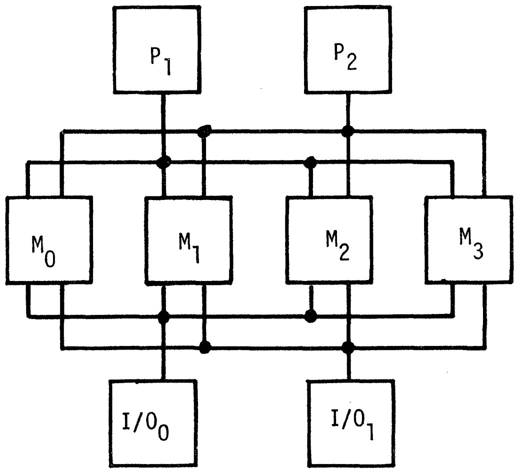


Fig.7: Multiport-Memory System Organization --- Basic Organization

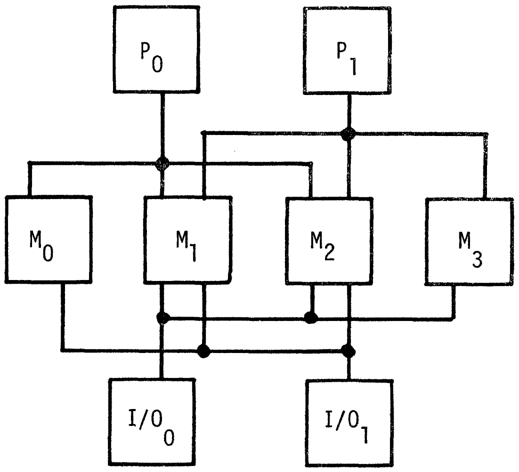


Fig.8: Multiport-Memory System Organization --- Including Private Memories

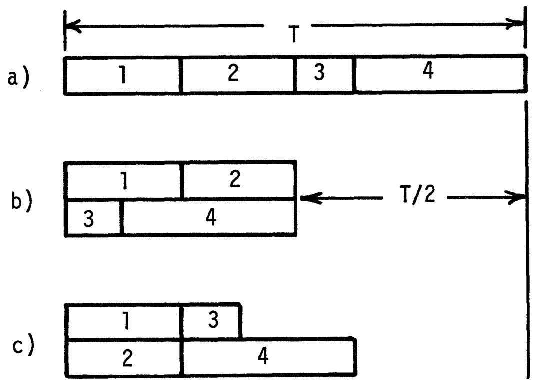


Fig.9: Multiprocessor Dispatching Anomaly

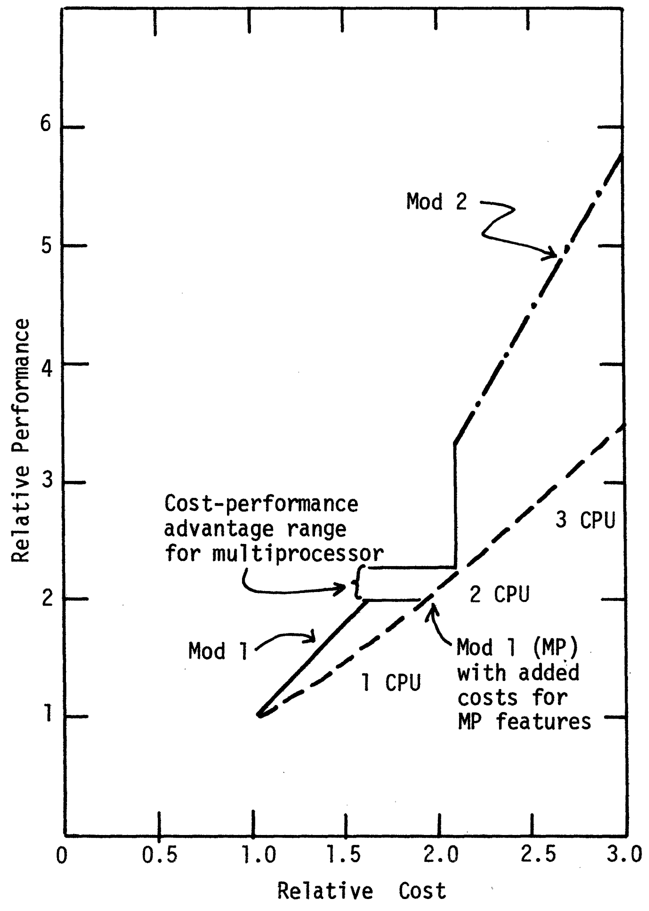


Fig.10: Cost-Performance Comparisons

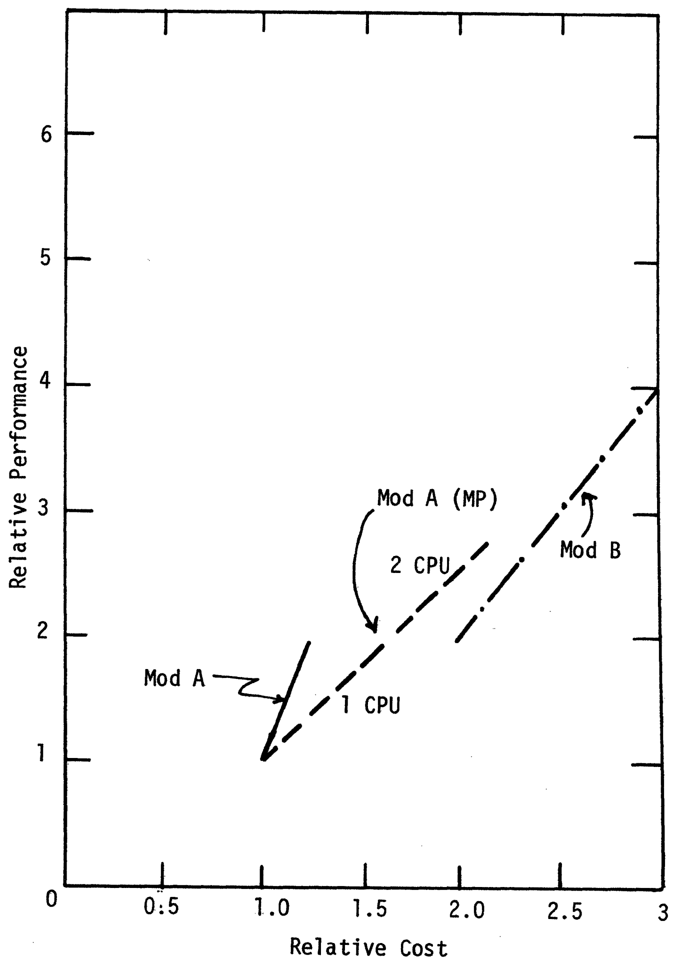


Fig.11: Cost-Performance Comparisons --- Inversion

PARALLEL PROCESSING IN SOFTWARE AND HARDWARE - THE MASCOT APPROACH

K Jackson
C I Moir
Royal Radar Establishment
Malvern
UK

Abstract -- Real-time computer based information systems contain a high degree of parallel processing outside the computer. Consequently, it is best to consider the tasks which the computer has to perform in support of these external parallel processes as being executed in parallel. A software structure is suggested which aligns modularity with parallel processing leading to flexibility during operational use. This software structure is capable of implementation on either single or multiple processor computers or a network of distributed computers. An experimental distributed computer system being built to study the problems of high integrity hardware configurations is described. An interesting feature of this system is the pseudo-random data transmission network it contains.

1 Introduction

All computer based systems contain three parts: an environment, men and software. The environment contains sensors and actuators some monitored and controlled by men, others by software. The men also inject data into the software and are given information by the software in return. Although the actual interactions between these three parts are particular to any given system, in general information can flow in both directions between the three possible pairs namely: environment to/from software, men to/from software and men to/from environment. Figure 1 represents this diagrammatically. The model applies equally to a wide range of systems from air traffic control to industrial process control and can even be applied to a computer bureau!

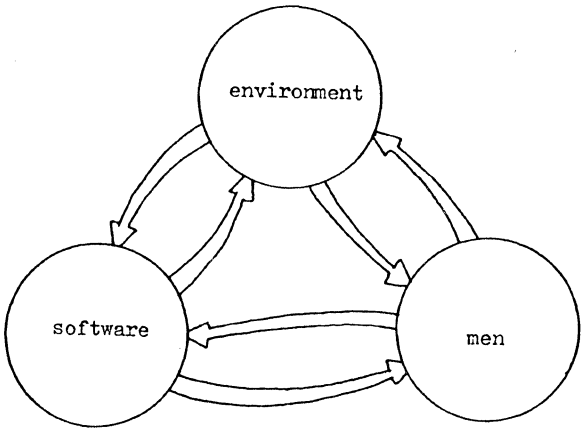


Figure 1. Components and interactions in real-time systems

Three pertinent points can be made from this model. Firstly, within the model there are many concurrent activities; each man works independently, each sensor and actuator works independently. Therefore the environment and the men can be considered to be sets of parallel processes. Secondly, the degree of interaction between these parallel processes depends upon the system. One extreme is represented by a multiple access computer where several people can use a single computing facility at once but the interaction between the users is nil. The other extreme is the dedicated command and control information system where several people must interact with each other and with the environment and software in order to perform a single joint task e.g. air defence. Thirdly there are many channels of interaction between the three components and the channels are open continuously.

It follows that the software in the general model must have a large number of tasks placed upon it. Each sensor and actuator in the environment under software control demands some attention and each of the men who interacts with the software has a (possibly unique) repertoire of tasks he can ask the software to perform. In the design of these systems it is therefore impractical to treat the software as a single sequential program.

The problem of dividing programs, and hence organising software, plays a central role in software engineering [1]. The main tool for determining divisions within a program is that of functional decomposition; the process of repeated decomposition continues until the units, or modules as they are usually known, are of a suitable size for an individual programmer to manage. When modules have been implemented and tested individually, the inverse process of composition - often called integration - begins. At this stage the interfaces between modules are tested and many problems arise due to poor or inadequate specification. In fact there is evidence [2] to show that testing (module and integration) accounts for up to 45 per cent of software production effort in large command and control systems. During integration a further form of modularity is introduced. The basic units are still the most elementary functional modules but it is convenient to group these together for presentation to either a higher order language compiler or an assembler. Finally, when the software commences execution, a third type of modularity is exposed. Here the unit is the sequential process which is separately scheduled by a despatcher algorithm or is executed in response to an external hardware interrupt.

Thus, software for command and control information systems has to perform a large variety of tasks which are not sequentially related. The software that is produced to satisfy this requirement tends to be modularised in three different ways:

during design - by function
during implementation - by some convenient grouping for compilation
during execution - by splitting into a set of interacting processes running apparently in parallel.

MASCOT (Modular Approach to Software Construction Operation and Test)[3] proposes a unified discipline of modularisation based upon the initial decomposition of a program into a set of parallel processes (with a hierarchial decomposition within the main constituents by function as necessary). This approach imposes a strong discipline on the module interfaces which significantly reduces the integration period and also gives a more flexible end product.

2 Software Structure

Software is responsible for taking in a conceptually continuous stream of data and transforming it into a continuous stream of output data. In practice data is not continuous in form but is quantised into units such as bytes, words or blocks etc. Diagrammatically a piece of software can be represented as in figure 2. The two

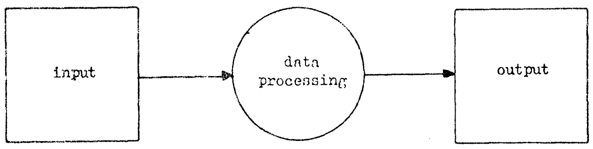


Figure 2. Elemental data processor

rectangular boxes represent the current quantum of input data and the current quantum of output data while the circle represents the processing operations performed by the software in making the necessary transformation. This diagram is over simplified for the type of applications being considered. Firstly we have said that there are several parallel streams of input and output data and therefore we need to have several input and output data boxes. Secondly we have argued that the data processing actions for the many input streams of data are not serially related and thus instead of a single data processing action we need several. These many processes must intercommunicate and the only means available for communication between parallel processes is through common data boxes. Therefore the software structure for command and control information systems (and most other real-time computer based systems) consists of a network of intercommunicating parallel processes as indicated in figure 3. The modular approach of MASCOT is primarily concerned with establishing such a network from a set of processes and intercommunication data areas.

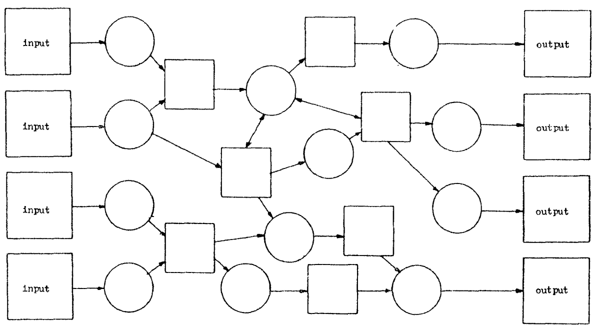


Figure 3. Network of data processors

A network of this nature can be constructed using conventional techniques without the MASCOT approach. However, the resulting program may contain hidden interactions (e.g. by two processes communicating via global data) and be consequently more difficult to debug and integrate. It also embodies a fixed network so that if a change is required one or more modules must be changed and then all the constituent modules must be link-edited to produce the total program. Consequently the unit of replacement is the complete program and this leads to difficulties in situations where 24 hours/day service is required.

In the MASCOT approach the concept of global data is eliminated. Instead, each process must explicitly state its total process external data intercommunication requirements. It follows from this approach that the result of the compilation and link-edit phases of software construction need not be a program but can be a kit of parts which can be used to construct networks of intercommunicating parallel processes. The method of constructing the network consists of satisfying the data requirements of each process by pointers to data areas of appropriate types. Because this network construction can take place after compilation and link-editing, the network can be changed dynamically.

3 MASCOT networks

At this stage it is worthwhile describing the software structure created by the MASCOT approach in more detail. Processes in MASCOT are called activities and the data processing actions of an activity are defined by a root procedure. It is the root procedure which has a set of formal parameters defining the number and type (see below) of intercommunication data areas it will require access to when it supports an activity. The network structure is strengthened by allocating each intercommunication data area a particular type which defines its internal structure. The act of creating an activity necessitates the specification of the particular root procedure required together with an appropriate set of intercommunicating data areas. Thus the kit of parts for generating a network of intercommunicating parallel processes contains root procedures and intercommunication data areas. These items are known as System Elements and the set of these items available for use in the construction stage is called the System Element File. This file includes information about the types of intercommunication data areas and the intercommunication requirements of each root procedure.

Before going on to describe the construction process in more detail, it is necessary to digress briefly into the philosophy of activity-activity intercommunication. The aim here was to be as unrestricting (and therefore as general) as possible. Thus we have suggested that each type of intercommunication data area can be interfaced to the activities which may use it by a set of access procedures. These access procedures can use the minimal set of process synchronisation primitives within the kernel (see section 4) to hide both the detailed data structure of the intercommunication data areas and the use of the synchronisation primitives. This leads to a clean design and further lessens the risk of interaction problems during the integration phase of software production because the access procedures can be exhaustively tested in advance and are usually sufficiently small to be guaranteed correct. Using this approach a large number of intercommunication mechanisms can be expressed within the same very simple framework. Two such mechanisms which have been found useful in a variety of different applications are described below.

Each mechanism introduces a particular category of intercommunication data area; these are the channel and the pool. The channel category of intercommunication data area is used for a message passing mechanism. This has two uni-directional interfaces implemented by a pair of access procedures: one for sending (called by the producer activity), the other for receiving (called by the consumer activity). It is useful to represent the channel diagrammatically by the symbol $\mathbf{I}$ (see figure 4). Many different types of channel can be used in a network but each channel type passes messages in a particular format and has a buffer with capacity to hold one or more messages in transit. The message format and buffer capacity are defined within the overall conventions for channel data structures. An example of a channel data structure and its access procedures is given in section 4.

Intercommunication data areas of category pool are complementary to channels. They are used primarily as a repository for non-transient data which is remembered and kept up to date as time passes. Pools are used for data bases and models of the environment etc. and are represented diagrammatically by the symbol $\sqcup$. No conventions have been laid down for the structure of pools nor for pool access procedures, but, as with the channel, each different pool structure is allocated an explicit type. It is expected that each designer will decide on his pool structure and use access procedures as appropriate. Typical use of pool access procedures would be to use a set as a data base management facility or to use them for resource control.

The unit of construction in MASCOT is not the activity but the subsystem. This is merely a network of one or more activities grouped together for convenience. The way subsystems are formed from system elements is best understood by example. Suppose we have two root procedures. The first one reads text as a sequence of characters from a channel of type 'CHARCHAN' and can recognise macro definitions; the definitions are remembered in a dictionary pool of type 'DICTPOOL'; subsequently any calls of the remembered macros in the text being read are recognised and expanded; the expanded text is output as a sequence of characters into another channel of type 'CHARCHAN'. The root procedure header might look like:

```
PROC expand = (REF CHARCHAN in, out,
          REF DICTPOOL dictionary):
```

using Algol 68 [4] notation for the parameters. The second root procedure called 'duplicate' takes a stream of characters from one 'CHARCHAN' channel and copies it into two further 'CHARCHAN' channels:

```
PROC duplicate = (REF CHARCHAN in, out1, out2):
```

If we now wish to create the subsystem "triplicated expansion" shown in figure 4 we could express it in the following way:

```
FORM triplicated expansion =

  (expand (input, inter1, dictionary),
   duplicate (trans1, out1, trans2)
   duplicate (trans2, out2, out3)
```

The FORM command (assumed to be issued via a command interpreter facility) first checks that each item mentioned exists as a system element and then that the parameters which have been given for the root procedure match the requirements as specified in the root procedure headers. Finally a new subsystem is created containing the (as yet inert) three activities. Once created the activities of the sub-system can be started by a subsequent command to start the subsystem. The subsystem is also the unit of stopping and removal.

The MASCOT approach leads to a network of intercommunicating activities organised for convenience into the set of subsystems which constitutes the system. The system can be varied whilst it is operational by adding, starting, stopping or deleting subsystems. This software structure offers many advantages for software production and it can be mapped on to many possible hardware configurations.

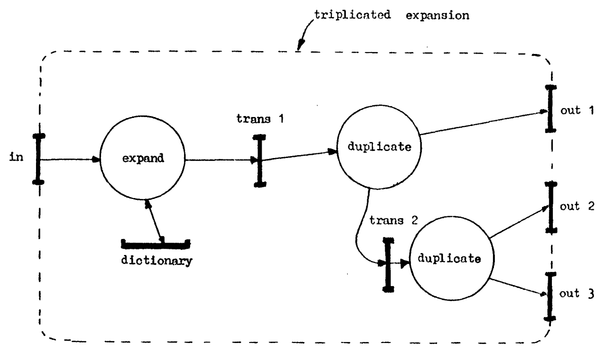


Figure 4. A subsystem

4 Mapping on to hardware

The software structure which has been described so far makes no assumption about hardware other than that it will supply a means of executing the activities in the network. This requirement can be satisfied at two extremes by either a single processor computer or by a network of inter-connected computers having one computer per activity. The latter solution may not be as far fetched nowadays as it once was but for the majority of applications the number of activities will usually be greater than the number of processors. Consequently there must be some means of running several activities on a single processor computer.

The main contention which arises when attempting to run a set of activities on a single processor is obviously the competition for processor time. This has been solved by the MASCOT kernel which includes a minimal set of synchronising primitive operations and two executive routines. One executive routine is responsible for allocating processor time on demand from external hardware interrupts; the other is responsible for allocating processor time to the base level (i.e. non interrupt driven) activities. The kernel also contains the procedures implied by the FORM command to create subsystems (and their activities), and to start, stop and remove them. The kernel has been implemented in Coral 66 [5] for a Marconi Myriad computer as a very small monolithic monitor [1] and the object code generated occupies less than 2000 words (24 bit). This includes a significant proportion devoted to a monitor facility which enables the creation of a time ordered record of calls of primitive operations by activities and executive decisions.

The objective in the specification of MASCOT kernel primitives was to identify a minimal set which was not only necessary and sufficient but also convenient for the handling of all problems of process synchronisation at the basic level. Thus all concepts of data passing via the kernel have been stripped out. Two basic mechanisms were identified as necessary [3,6]:

a) Mutual exclusion: it must be possible for each of a set of activities sharing data to gain exclusive access to the data (or some recognised part of it).

b) Cross Stimulation: An activity must be able to defer further execution until it receives an explicit software stimulus from another activity.

Although both mechanisms can be implemented using the conventional semaphore with P and V operations on it [1], it is considered that this practice leads to confusion. Whether a particular semaphore is being used for mutual exclusion or for cross stimulation in a given situation is not at all obvious. Therefore, on the grounds of improving understandability and convenience, it was decided to have a semaphore (called a controlqueue because it acts as the focal point for sequence control operations between activities) with four primitive operations available upon it. One pair of primitives - TEST/CLEAR - deals with the mutual exclusion mechanism; the other pair - WAIT/STIM handle the cross-stimulation mechanism with the condition that the WAIT operation can only be used by the activity which has secured (i.e TESTed) the control queue. To exemplify the use of these primitives we consider a channel and associated access procedures. Using Algol 68 notation again, we can define a channel data structure by the following MODE declaration:

```
MODE SIMPLECHAN =
    (STRUCT (CONTROLQ inputaccess, outputaccess,
     INT inpointer, outpointer, maximum
     [0:31] MESSAGE data);
```

This channel can be used for passing messages of mode MESSAGE (assumed to be previously defined). The two pointers are used to indicate the position of the next message slot to be used for input and output respectively. They are incremented by one after each insertion/extraction and used modulo the maximum value for the pointer, chosen to be $2^n - 1$ for efficiency (31 in this example).

An access procedure which could be used to put a message into such a channel might be written:

```
    PROC send = (REF SIMPLECHAN chan,MESSAGE n):
    BEGIN TEST (inputaccess OF chan);
          IF full(chan) THEN
          WAIT (inputaccess OF chan) FI;
          (data OF chan)[ inpointer OF chan
                    MODULO maximum OF chan] := m;
          inpointer OF chan PLUS 1;
          STIM (outputaccess OF chan);
          CLEAR (inputaccess OF chan)
    END;
```

where the procedure "full" delivers the value 't 'true' when all slots are full otherwise 'false' and the operators MODULO and PLUS are self explanatory.

The corresponding access procedure to remove a message might be written:

```
  PROC receive = (REF SIMPLECHAN chan,
                  REF MESSAGE m):
  BEGIN TEST (outputaccess OF chan);
        IF empty(chan) THEN
        WAIT (outputaccess OF chan) FI;
        m := (data OF chan)[ outpointer OF chan
                    MODULO maximum OF chan];
        outpointer OF chan PLUS 1;
        STIM (inputaccess OF chan);
        CLEAR (outputaccess OF chan)
  END;
```

where the procedure "empty" delivers the value 'true' when all slots are empty otherwise 'false'.

When called each of these procedures attempts to secure the appropriate interface for use by the calling activity. At this point the activity may be held awaiting its turn to use the interface. When the calling activity has possession of the interface it may yet be held up due to the state of the data area. If held at this point (WAIT), flow of data across the interface ceases until a stimulus to restart comes from the other side of the channel; such a stimulus is only transmitted when the change of state is likely to be of interest to activities on the opposite side of the channel.

The control queue has also been included in the control data structure for hardware interrupts which is known as a virtual interrupt. The primitives TEST INTERRUPT and CLEAR INTERRUPT are similar in operation to TEST and CLEAR (the only difference being that they operate on a virtual interrupt instead of directly on a control queue). The WAIT INTERRUPT primitive allows an activity to defer further processing until a hardware interrupt stimulus is received. Further details of the kernel facilities and the Myriad implementation can be found in references [3,6].

The kernel can be easily adapted to the co-equal multi-processor type of computer. The chief question to be answered is the extent of mutual exclusion necessary between the processors. The simplest if rather crude solution is to allow only a single processor to execute within the kernel at a time. Since the kernel has been designed with the utmost efficiency in mind, this solution may be perfectly adequate. More sophisticated solutions involve selective lockouts in crucial areas such as list manipulations and the need to prohibit more than one processor from simultaneously executing a primitive operation on the same control queue.

For the multi-computer system, in addition to the allocation of processor time to activities, there is a further problem of allocation of activities to computers. The simplest way of achieving this is to make the subsystem the allocatable unit so that each unit of computing is responsible for the execution of a set of subsystems. (This brings in a further level in the system - subsystem - activity hierarchy). It is useful to note however that with this organisation of software into subsystems it is possible to design and implement the software first on a large single computer. This allows the investigation of software problems of distributed computer systems in the absence of any of the hardware problems associated with the distributed computer hardware. Subsequently the software subsystems can be allocated to individual computers with a hardware link between two sides of any channel which straddles a computer-computer boundary. An experimental distributed computer system, which we call FRIMP, is being built to study the problems of high integrity hardware configurations and the problems associated with transferring software from a single processor computer on to a distributed computer system.

5 FRIMP - Flexible Reconfigurable Interconnected Multi-Processor System

There are many applications which require a high level of system availability and integrity. System here includes both hardware and software. The hardware must be capable of detecting and isolating faulty units (computer, store, peripheral) and it must be possible to introduce new hardware into the system whilst it is operational. This includes reinstatement of faulty units after repair and adding new units to upgrade performance. The conventional approach to improving hardware availability is to duplicate, triplicate, quadruplicate ... hardware units until the desired cumulative mean time between failures has been achieved. The problem introduced by this approach is that there must be voting logic which tends to become the vulnerable point of the hardware. Another approach is the multi-processor system having spare processor capacity that can be utilised in the event of faults. Two problems arise here: firstly a fault can cause untold damage to data and programs before it is discovered, thus reducing integrity; secondly, there is a strong possibility of store contention reducing the gain of adding extra processors. A third approach is to have a distributed computer system. This allows extra power to be added without such a dramatic application of the law of diminishing returns. Faults within a single computer are much less likely to cause chaos in others especially if defensive programming techniques are employed on the computer to computer data transfers. Online reconfigurability is possible provided that a very reliable intercomputer communication medium can be provided and a suitable control strategy can be found.

FRIMP is a distributed computer system which is being built in order to study the problems outlined above. The feasibility of the entire project depends upon a reliable data transmission medium. This must allow the necessary connectivity (including multiple paths) to be made between communicating devices and provide sufficient capacity. Also the total network must not depend on a single arbitrator at any one point. Current work on this medium is discussed below. The other major requirement for the distributed computer system is a control strategy. This will be based initially on the MASCOT approach with subsystems being allocated to individual computers. The subsystem allocation will attempt to keep all computers as busy as possible. Thus changes may be made to meet peaks in demand on a particular subsystem or to redistribute the subsystems of a failed computer. This control strategy must be secure, yet since it involves management of the system's resources as a whole it is very difficult to disperse the responsibility effectively amongst the individual computers. Fortunately the MASCOT concept reduces the need for global system management operations to such an extent that, when such operations are unavoidable, they can be done relatively slowly. Provided that any activities which are not affected by the global operation can continue normally, the system will have time to perform a self-test routine on every computer, "elect" a leader from amongst those which pass the test, wait for the leader to perform the operation, and then perhaps appoint a deputy to repeat the operation as a check. This mechanism avoids the cost of a majority voting system applied to all operations, but retains its security advantages in the most critical operations, namely those which involve the system as a whole. It also avoids the concentration of logic at a single, vulnerable point which is found in some majority voting systems.

FRIMP is being constructed using microprocessors which have a micro programming facility. This facility suggests several potential advantages over a software only version of MASCOT. Firstly a store protection mechanism using base, limit and access registers will be built. This will confine activities to their connected intercommunication areas and, since each area can have its own register, access time to the data will be quicker than using indirection or indexing. Secondly it is planned to build a micro programmed version of the most commonly used parts of the MASCOT kernel, whose very simple structure lends itself to this type of implementation. Finally where an intercommunication area is used by activities in different computers the access procedure mechanism could be used to trigger a virtual storage system into either fetching the data (write or read access) or fetching a copy of it (read only access).

6 Data Communications Medium

The need for a reliable data communications medium in FRIMP has been stated above. The problem was to find a means of achieving the desired level of interconnectivity and capacity between the many intercommunicating devices within FRIMP. The need for reliability and flexibility has excluded any method involving a central vulnerable point. This excludes single highway systems and multiple cross-bar highway systems. Connexion strategies where n devices of one type are connected to all m devices of another are also excluded on cost, due to the n * m expansion factor, in addition to having a single crossing point in some cases.

Current work at RRE is examining a communications network consisting of interconnected nodes with pseudo-random data routing. Each device connected to the network has its own unique code. Messages transmitted between devices carry a header which contains both the source and destination device numbers and the length of the message (0-256 bytes). Each node is a store and forward switching centre having up to eight input ports and up to 8 output ports, and storage for at least one maximum length message, though in practice more store than this would be provided. When a message arrives at an input port, the destination device is used to access a local look-up table indicating the set of output ports which would be suitable for onward transmission. The information extracted from the table can be 'here' (meaning the destination device is connected to this node, 'not connected' (a fault condition - the device being sent to is not connected to the network), or 'port selection data' (if the destination device is not local). The exact nature of the 'port selection data' depends upon the particular routing algorithm being used (see below). Once a port has been selected and found to be not busy the data is transmitted serially through it to the next node. Figure 5 gives an example of a network showing multiple paths between all devices not connected to the same node.

A computer simulation of the network has been performed using Algol 68 RT. (A variant of Algol 68-R [2] which allows parallel processing). The program has been used to investigate the properties of the network and to study the effects of routing strategies and node design on performance. The simulation consists of four types of process: node, transmitter, receiver and statistical sampler. The node process is launched as an independent task once for each node in the network being simulated. The transmitter or receiver process is launched once for each device of the appropriate type. The statistical sampler is also the control process and it reads in a file to define the network connectivity and define the pattern of data packet transmission which each transmitter will attempt to make. Data output by the sampler, when it runs each simulated millisecond, includes the total number of packets sent and received since the start, the number of new packets introduced during that millisecond and the maximum transmission and access delays. At the end of the run a histogram is printed which indicates the distribution of transmission and access delays during the simulation run.

Using this model two different node processes have been tested. The first one imposes a fixed 2 byte packet size on transmissions through the network, and the routing data in the look-up table contains in addition to 'here' and 'not connected' either 'clockwise' or 'anti-clockwise'. Ports are marked 'clockwise' or 'anti-clockwise' and so the route essentially gives a choice of one of two sets of output ports. The algorithm chooses the first one it encounters which is not busy. In the second node process the look-up table gives a first choice port and a second choice port. The ports are examined in that order. Also in this simulation the packet data length is variable in the range 0-256 bytes. One surprising fact which has emerged from these simulations is that the characteristics of the network are not significantly changed by altering the node algorithm, but in both cases a deadlock situation can arise if all nodes are allowed to accept all the data they are offered and can accommodate. Therefore a limit has been set which stops a node accepting 'new' data (i.e. data from a device connected to that node) if the proportion of the node storage occupied exceeds a given limit. Data passed on from other nodes is still accepted and further new data is also accepted as soon as the peak is passed. Each node acts entirely autonomously in this way, monitoring only its own traffic, and is therefore not vulnerable to failures elsewhere in the network. The other surprising fact is that independently of connectivity and transmission pattern, imposing a limit on the rate at which new traffic is accepted actually improves the overall throughput of the network and reduces the maximum transmission delays. A limit which blocks inputs when 75% of the available storage in a node is filled is sufficient to prevent deadlock even under saturation conditions (when every transmitter generates data as fast as the network will allow), but a lower limit gives the best throughput. The graphs in figure 6, for a 16 port/node simulation, show the rate of acceptance of new packets (roughly equivalent to throughput), access delays and transmission delay as traffic increases plotted against the number of busy ports limit. This phenomenon is being vigorously investigated and the simulation work programme includes an investigation of techniques for adaptive route changing which can be applied independently at each node.

Conclusion

The MASCOT approach allows design of real time computer based command and control information systems in terms of a network of interconnected parallel processes. This design can proceed independently of the eventual computer configuration. The network of processes can be implemented and run on a large machine by providing the small MASCOT kernel. Then it can be implemented on the target machines, if necessary changing the programming language in the process but keeping the same modular real time parallel processing structure. This form of modularity, it has been proved, significantly reduces software integration time and promises the possibility of re-usable program modules because of the tight discipline which forces explicit specification of each activity's external data interface.

The MASCOT modular structure is also very suitable for applying a distributed computer system to a single overall task (e.g. air defence, air traffic control). A distributed computer system is being built, using micro-processors, as a vehicle for validation and experimentation. Initial simulation studies of the proposed communications network for the distributed computer system indicate that it will meet its objective and also indicate some interesting properties which will be investigated together with the possibility of adaptive route control at each node.

References

[1] R C Holt, "Structure of Computer Programs: A Survey", Proc IEEE, Vol 63, No 6, June 1975.

[2] D W Kosy, "Air Force Command and Control Information Processing in the 1980's: Trends in Software Technology", Rand Corporation, Santa Monica, 1974.

[3] K Jackson, H R Simpson, "MASCOT - A modular approach to software construction operation and test", RRE Technical Note 778, Ministry of Defence (Procurement Executive), London, UK.

[4] P M Woodward, S G Bond, "Algol 68-R Users Guide", Her Majesty's Stationery Office, London, UK, 1972.

[5] P M Woodward, P R Wetherall, B Gorman, "Official Definition of Coral 66", Her Majesty's Stationery Office, London, UK, 1970.

[6] K Jackson, H R Simpson, "MASCOT - A Modular Approach to Software Construction Operation and Test", Agard Conference Proceedings No 149 on Real Time Computer Based Systems, NATO, May 1974.

Acknowledgements

The authors are indebted to many colleagues for useful comment and discussion and in particular to Dr R J W Kershaw who undertook the network simulation work.

This paper is published by permission of the Director, Royal Radar Establishment, Copyright HBMSO, London, UK.

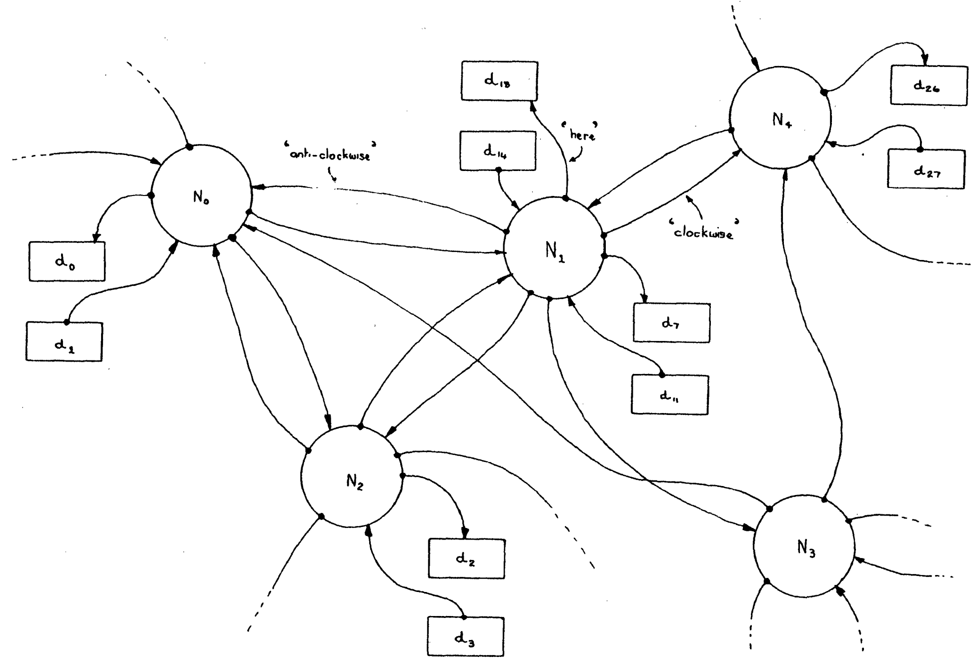


Figure 5. Random network example

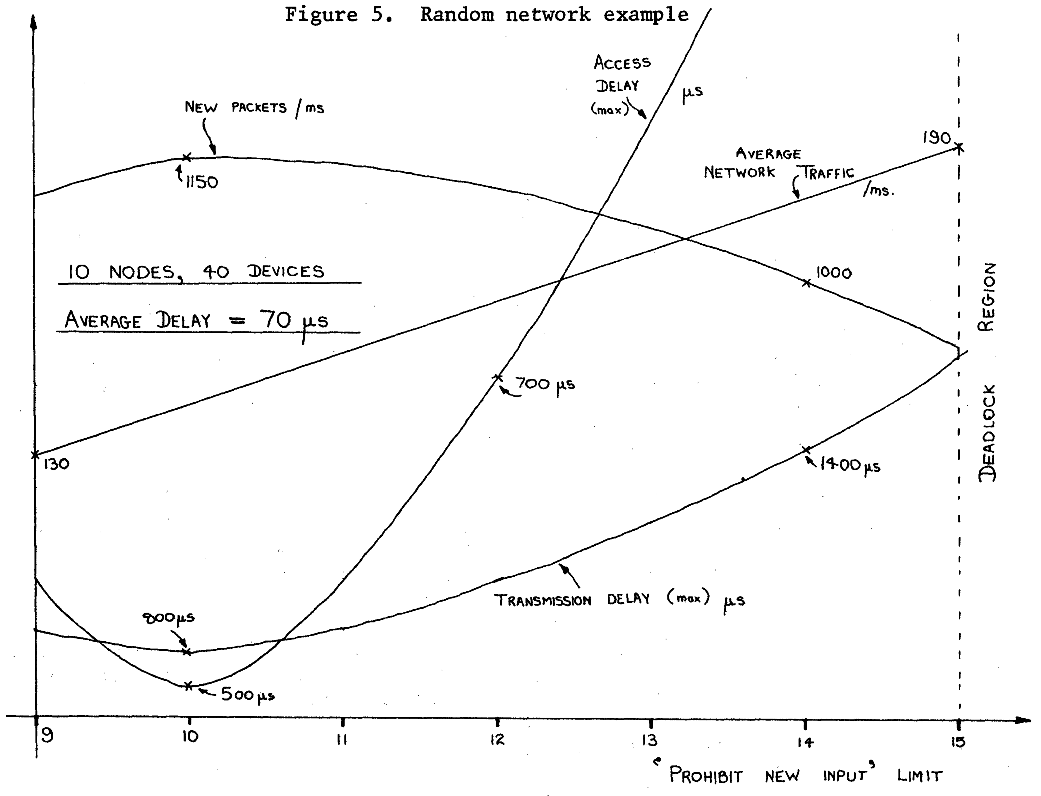


Figure 6. Graphs describing transmission characteristics of random network

SEQUENCING CONTROL IN MULTIFUNCTIONAL PIPELINE SYSTEMS[†]

C.V. Ramamoorthy and H.F. Li

Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California at Berkeley
Berkeley, California 94720

Abstract -- A parallel and pipeline processing implementation scheme termed RSRP is considered. It consists of parallel functional pipes sharing resources at strategic locations. The sequencing problem for RSRP systems is considered first from a theoretical point of view, by characterizing its intrinsic complexity using a convenient classification termed "inherently difficult". From this result, the semi-exhaustive nature of an optimal strategy is justified. However, for low level of implementation, faster heuristics are necessary. Some simple efficient heuristics were compared using experimental simulation on systems whose models were based on some existing machines.

1. Introduction

The study of computer architecture coupled with technological advancement has blossomed in many ways. In developing new processing techniques and organizations for amplifying the computing power available from a computer system, both parallel and pipeline processing techniques are favorable candidates. They actually revolve around the same concept: overlapped operation among the facilities or modules in the computer system. By overlapping, more tasks could be executed at one time. Consequently, higher utilization and throughput rate are within reach. As a co-product, the turnaround times of computing jobs could be improved which may be an important requirement in some real time applications such as weather forecasting, air traffic control and other real time processes.

In a parallel processing architecture, usually there are numerous processing elements which could execute independent instructions or task(s) concurrently. In practice, due to the limitation on the memory-processing element relationship, most parallel processing exists in a form of array processing where identical processing elements are executing the same instruction for some array type of computation. So a SIMD characterization [1] fits array processing very well. Well-known array processing machines include ILLIAC IV, STARAN and PEPE [2-4]. On the other hand, a more general MIMD structure where different instruction streams are being executed can also be seen in the PEPE machine since there are three identifiable different control-PE paths so that three different instruction streams can feed into the 288 processing elements with parallel instructions to be executed. However, a general parallel processing system with powerful autonomous identical processors to execute MIMD streams is still an uncommon super-giant, though parallel processing at a lower level, namely among smaller logical elements, is a common practice.

[†]Research sponsored by National Science Foundation grants DCR72-03734-A01 and DCR74-21248 and U.S. Army Research Office Contract DA-ARO-D-31-124-73-G157.

Pipeline processing can be characterized by the SISD classification because a pipeline may commence operation on one set of data for some instruction per minor cycle. Since a processing phase is being decomposed to several sub-phases executed on autonomous functional modules in a pipeline processor, the overlapping among the processing of consecutive instructions provide the amplification of throughput desired. As the submodules are usually cheaper than a complete module, therefore pipelining is useful to speed up operations in processors in a most cost-effective manner wherever possible. It is applicable to many levels of a processor design. In a higher level, the instruction processing is pipelined into many phases such as instruction fetch, decode, operand fetch and execute in the IBM/360 model 91, 195, etc. [5]. To speed up a CPU further, execution units can be pipelined for performing the arithmetic operations as in TIASC arithmetic units and the CDC STAR-100 machine [6-7].

Parallel and pipeline processing techniques are complementary in achieving higher throughput, and it is by no means surprising if a person discovers the presence of both in a computer system such as in the STAR-100 and PEPE machines. In this paper, a modeling of parallel-pipeline processing is established for the discussion of throughput and sequencing control in a system with parallel functional paths (pipes) sharing some strategic resources, termed a Reconfigurable Shared Resource Pipeline (RSRP) system. Sequencing in a parallel-pipeline system is a vital activity in order to fully utilize the overlapping modules in the system because a continuous stream of instructions should be provided and executed with as little disturbance as possible to the system configuration. In this way, the system resources can be kept in a busy state to produce useful outputs.

The importance of scheduling or sequencing in a parallel or pipeline machine can be reflected by the enormous research efforts devoted to the study of optimal algorithms for them. There are roughly two lines of research to be followed. First is the development of optimal sequencing algorithms for a deterministic task system where task precedence relationships and execution requirements are known [8]. The other is a more realistic model where a stochastic precedence relationship or execution time is allowed in the task system [9]. The optimal algorithms should be able to derive an optimal schedule for executing a given task system under specified conditions in a very efficient manner. Their qualities are judged mainly by their average speed, and sometimes their worst case performance [10]. Since 'average speed' is difficult to define and compare both qualitatively and quantitatively, the latter figure of reference is adopted by many people. For instance, essentially enumerative methods are considered poorer than simple systematic procedures that require a 'well-bounded' number of iterations or steps before its termination even in the worst situation.

Unfortunately, due to inherent characteristics, even under the most simple deterministic model, scheduling in general is quite difficult. Over the years of research, only three special cases were found to possess simple optimal algorithms. They include (1) a tree type of precedence relationship for a task system with unit execution time per task [11], (2) a 2-processor system executing a task system with unit execution time per task [12], (3) a two-facility pipeline system with variable speeds [13].

In this paper, the sequencing problem in a mixed parallel-pipeline architecture, the static and dynamic RSRP systems to be explained later, will be explored. The design and control of these two types of RSRP systems will also be discussed. Although a RSRP system does not necessarily fall into the framework of a general parallel processing system with identical processors, the inherent difficulty behind optimal sequencing in both static and dynamic RSRP systems will be characterized in order to justify the use of a semi-exhaustive approach to optimal sequencing and simple near-optimal heuristics in some practical situations. Finally, the heuristics for static and dynamic RSRP systems whose models are derived from existing systems will be compared with some experimental simulation.

2. Modeling

The processing phase within a computer system can be described by many possible models, based on the objective of modeling. For some purposes, a very detailed modeling is necessary. But for others, a simplified model helps the analysis and reveals the most important characteristics of the actual system. In most cases, system modeling revolves around a graph structure. Sometimes, additional semantics of tokens provide the additional information desirable. For example, marked graphs or Petri Nets [14-15] can be used to describe the exact operation and synchronization of a modular system.

For the purposes of this paper, we are concerned with the throughput of a complex processing system which has a structure describable by the various functional paths (pipes) within the system, sometimes with some strategic resources being shared among the functional paths. Under this processing organization, both parallel and pipeline processing characteristics emerge. Pipelining is recognized because a functional path is composed of a sequence of modules each performing some phase of processing in an overlapped mode with the others. Simultaneously, parallel processing may be achieved because concurrency of execution may take place among the various functional paths (pipes), analogous to the Multiple Instruction Multiple Data (MIMD) stream type of computer systems [1]. As a result, independent tasks or instructions are guided through the different required functional paths in a pipelined manner with the objective of getting the most utilization from the system resources and hence the highest throughput rate possible. For obvious reasons, such a processing system will be named Reconfigurable Shared Resource Pipeline system (RSRP).

There are two kinds of RSRP systems which will be considered here -- static and dynamic RSRP systems. A static RSRP system is less flexible and less intelligent in the sense that at any time instant, only one configuration or functional pipe may be active. Therefore pure pipeline characteristics exist, though over a time period different pipes may be excited. This design has the advantage of less control circuitry and overhead needed in monitoring the routing of operands and gating activities in the pipeline segments. This also has the obvious disadvantage of less overlapping in other inactive paths and hence reducing the opportunity to achieve maximum throughput. Some simple static RSRP systems can be observed in the arithmetic unit pipes in computers such as TIASC and CDC STAR-100 [16-17]. In the example of TIASC systems, the machine has fourteen different groups of instructions. For a group of instructions involving the same pipeline configuration, if the needed operands are fetched fast enough, the arithmetic unit can produce a fastest throughput rate of one result per minor cycle. However, to avoid excessive switching, only one active configuration or pipe is allowed at any time. On the other hand, a dynamic RSRP system permits concurrent processing in the various functional paths (pipes) with some additional control to route operands to correct transitions. Therefore, simultaneously several functional pipes may be active, although collisions at a shared resource have to be either avoided or resolved by proper buffering and sequencing control. There are certainly some tradeoffs between a static and dynamic RSRP system. Here their performance under some sequencing rules will be studied.

Hence, formally a RSRP system will be represented by a modified digraph consisting of a 3-tuple $G = (N,A,P)$ where N denotes the set of facility modules or nodes, A the set of transition arcs among the facilities, and P the set of legal functional paths (pipes) in the system. Sometimes when used in a deterministic model, it can be extended to a quadruple $G' = (N,A,P,T)$ where T provides the additional information about the execution speeds of the facilities in N. With this, the analytical throughput rates of the pipes under no interference conditions are computable. Notice that not all possible paths in the digraph are legal paths because there may exist configurations that do not have logical meaning and their activation will produce erroneous outputs.

3. Collision Avoidance

Given a dynamic RSRP system, some deterministic analysis will be useful for controlling the operation of the system for optimization purposes under different operating assumptions [18]. Because of the presence of shared resources and multiple tasks currently being executed by the system, care must be taken to accommodate the occurrence of collision. A collision occurs when two or more tasks try to access the same facility at the same time. When a collision has occurred, the system control must have built-in (hardware or software) collisions resolvers and/or buffers of some kind in order that proper execution can continue at its normal pace.

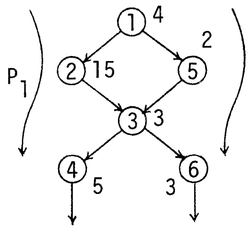
Similar to other undesirable events, a collision can be either prevented or resolved. If prevention is the goal, some global sequencer may be designed so that a task, once initiated, will not cause any collision with other tasks still resident in the pipeline system. This further implies that a task will 'flow' through the RSRP system without any waiting inside after its admission. This goal has the advantage that implicit requirements on intermediate buffering capabilities between facilities are not imposed. But then it loses some chance in enhancing more overlapped operation provided by sufficient buffering.

An almost exactly analogous situation between a dynamic RSRP system and a traffic network can be drawn up easily. A shared resource corresponds to a traffic junction. Under a deterministic assumption, the exact speeds of vehicles and the lengths of blocks of roads are assumed known. Cars may then be admitted under a global controller which will allow entrance at some pre-determined sequence of the synchronization signals. On the other hand, internal buffering may be used to avoid collision at a junction in a similar way as the use of traffic signals. Of course, excessive traffic congestion on one route will result in the overload of 'buffers' -- an expected phenomenon of an ill-balanced dynamic RSRP system. Sometimes, remedy may be sought by dynamically changing the periodic ratio of traffic signals for the junctions involved. In particular, the heavily loaded direction may be favored to relieve the unbalance -- similar to a dynamic priority assignment to shared resources among the different related processing paths.

For the immediate discussion, the collision avoidance technique in a RSRP system will be tackled. This is especially important when pipelining is implemented in a very low level (in order to achieve the ultimate speed). Then the speed of a typical facility node may be of the order of 50 nsec. and therefore intermediate buffering demands comparatively excessive static and dynamic overhead (since the cost of intermediate buffers will be almost the same as other component costs and the total delay of the pipeline may be doubled). Consequently, except for simple operand routing, additional buffering between facilities may be undesirable when pipelining is performed at a very low level.

When sufficient buffering is absent, collision inside the pipeline system has to be avoided by a global control mechanism. In [19], a reservation table approach is suggested for sequence control of a linear pipeline with a single configuration. From a static reservation table, the initiation procedure (of a certain periodic length) is chosen such that highest throughput rate is attainable with complete collision avoidance. For a multifunctional RSRP system, a similar approach utilizing a two-dimensional collision matrix is possible. As the name implies, a collision matrix is a generalization of a one dimensional collision vector characterizing the relationships among the functional paths.

Each entry in the collision matrix contains information regarding the collision relationship between the two pipes concerned. Specifically, the (i,j)th entry represents the time intervals after the initiation of pipe i so that the excitation of pipe j will not cause a collision inside. For example, $\{(2,6),(10,17),(20,\infty)\}$ in the (i,j)th entry means the excitation of pipe j after pipe i can take place between the 2nd and 6th cycles, or 10th and 17th cycles, or after the 20th cycle. Each entry in a collision matrix may contain several time intervals instead of a single one because the two pipes involved may share more than one resource, thus introducing more sites where collision may occur. As an example, the dynamic RSRP system in Fig. 1 has a collision matrix as shown.



P_1: 1-2-3-4

P_2: 1-5-3-6

(Speed of each facility is as labeled.)

Collision matrix:

$t_{11} = (15,\infty)$

$t_{12} = ((4,10),(16,\infty))$

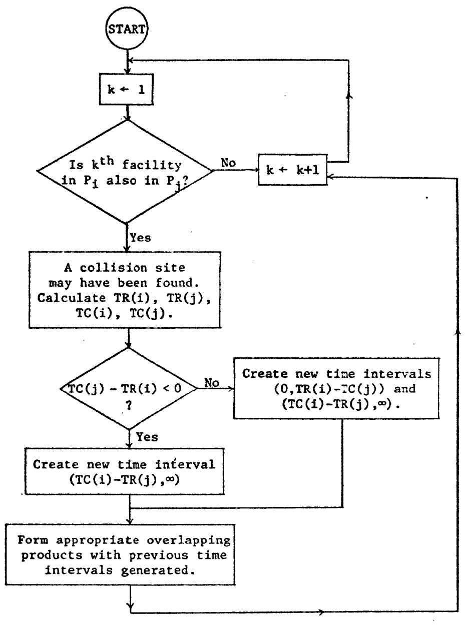
$t_{21} = (4,\infty)$

$t_{22} = (4,\infty)$

Fig. 1 Example Collision Matrix

The (1,1) entry is $(15,\infty)$ indicating that pipe 1 can be excited at regular intervals of 15 cycles or more because the slowest facility in pipe 1 generates an output in every 15 cycles and so forms the bottleneck of the pipe. The (1,2) entry is $\{(6,10),(16,\infty)\}$ because pipe 2 may collide with pipe 1 at facility 1 as well as facility 3. The (0,4) interval characterizes collision at facility 1 and (10,16) at facility 3. Notice that the (2,1) entry is single-valued despite the fact that pipes 1 and 2 share two resources. This is so because once pipe 2 is excited and has passed through facility 1, there is no way for the task in pipe 1 to catch up. The flow-chart of the algorithm which can be used to construct the collision matrix given (N,A,P,T) is illustrated in Fig. 2. It represents the procedure in generating the (i,j) entry. For simplicity, if pipe i and pipe j share a sequence of consecutive facilities, the latter are grouped together with a composite throughput rate equal to that of the slowest facility in this group. Also the time to reach and leave the composite facility will correspond to that for the slowest facility in the group.

With this collision matrix, an external global sequencer may sequence instructions or tasks according to some sequencing rule or algorithm and initiate them so that no collision will occur inside the RSRP system. Naturally one wonders what sequencing rules should be used given a task system. Should the sequencer try to minimize the


START
k ← 1
Is k^th facility in P_i also in P_j?
No
k ← k+1
Yes
A collision site may have been found. Calculate TR(i), TR(j), TC(i), TC(j).
TC(j) - TR(i) < 0 ?
No
Create new time intervals (0,TR(i)-TC(j)) and (TC(i)-TR(j),∞).
Yes
Create new time interval (TC(i)-TR(j),∞)
Form appropriate overlapping products with previous time intervals generated.

Notation: $TC(i)$ = time to leave the collision site via pipe i

$TR(i)$ = time to reach the collision site via pipe i

Overlapping product:

Illustration: Suppose previous time interval is $(4,\infty)$ and the newly generated are $(0,7)$ and $(10,\infty)$. The resulting intervals will be $(4,7)$, $(10,\infty)$.

Fig. 2 Flow-chart for Constructing Collision Matrix

execution time of the task system? What is the gain-overhead tradeoff? Is optimal sequencing intrinsically difficult? These problems will be the subject of the next section.

4. Theoretical Basis of Sequencing

To justify the use of a semi-exhaustive technique to perform optimal sequencing, a study of its intrinsic difficulty will be included. The classification of 'difficult' problems using 'polynomial completeness' will be used.

A big class of combinatorial problems requires the determination of certain properties in graphs, integer arrays, boolean functions and finite sets. Through the use of suitable encoding, these problems can be transformed into language recognition problems over a finite alphabet. Then one could test its intrinsic complexity by developing a conclusion as to whether there exists a fast recognizer for the language. Based on the 'satisfiability problem', the class of polynomial complete (PC) recognition problems are characterized so that if any possesses a 'fast' algorithm, it can be modified to become a 'fast' algorithm for any other in the class ('fast' means the algorithm terminates in a polynomial bounded time) [20-21]. Falling into the PC class of problems include well-known combinatorial problems such as 0-1 integer programming, set packing, node covering, set covering, Hamiltonian circuit, knapsack, etc. It has also been conjectured, with strong circumstantial evidence, that no PC problem has a fast algorithm.

It is nice to be able to classify languages in the previous way. But optimization problems in general do not restrict themselves to a Yes or No type of answer supplied by a recognizer. More generally, a minimization or maximization of some objective function subject to constraints is involved. So an extension of the PC classification is useful.

Definition. A PC problem L is reducible to an optimization problem P ($L \leq P$) if and only if there exists a polynomial bounded time transformation F from S_L to S_P and a simple recognition function G such that $X \varepsilon L \Leftrightarrow G(Z(F(X))) = 1$ where S_L and S_P are spaces of problem specification for L and P, $X \varepsilon L$ denotes X being recognized (Yes output), and $Z(F(X))$ is the output left in the optimization problem P with specification $F(X)$, corresponding to its optimal objective function value. The optimization problem P is said to be inherently difficult if there exists a PC language $L \leq P$.

Therefore, if an inherently difficult problem has a fast optimal algorithm, then $Z(F(X))$ can be generated in polynomial bounded time. Further, it implies X can be recognized in polynomial bounded time by simply concatenating output to the fast recognizer G. Consequently, L will have a fast algorithm. But because of the conjecture, no PC problem exhibits this property. So inherently difficult problems do not seem to possess any fast algorithm.

As an illustration of this notion of inherent difficulty and an aid to later proofs, the following example assertion is provided.

Lemma. The traveling salesman problem (TSP) is inherently difficult.

Proof. The traveling saleman problem is to find a shortest tour (through each city once and only once) given a (directed) graph indicating all the routes between them. For our discussion in this paper, let us assume that there exists an arc between any pair of nodes in the TSP. It will be shown that the (directed) Hamiltonian Circuit Problem (HCP) which is known in PC is reducible to it. The procedure is as follows. Given the HCP specification, attach a cost of 0 to all existing arcs and a cost of 1 to arcs that have to be added (see Fig. 3 as an example). This completes the F-transformation.

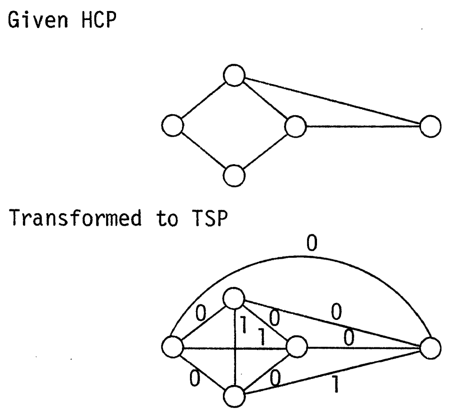


Fig. 3 Reduction of HCP to TSP

Define

$$G(Z(F(X))) = \begin{cases} 1 & \text{if } Z = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, both F and G are polynomial bounded in time, and hence HCP $\leq$ TSP. In fact, the resulting TSP has a shortest tour of zero cost if and only if the original HCP has a tour or circuit. Q.E.D.

Now one should realize that if sequencing problems are polynomial complete or inherently difficult, perhaps he should not be too ambitious to create a polynomial bounded time optimal algorithm. Sometimes, simple near-optimal techniques should be used. The 'inherently difficult' classification is quite useful because general optimization problems such as minimization of execution time or critical resource of a task system by a multiprocessor system can be found to belong to it [22]. Here, let's focus on the sequencing problems for pipelines. Observe that multiple pipeline processing is a type of multiprocessing, so that some results obtained for multiprocessors are readily extendable to it.

Let us deviate a little bit for the time being from a deterministic RSRP system and consider a linear pipeline executing a task system whose tasks have distinct execution time vectors on the facilities concerned. This problem will also form the basis of the sequencing discussion in RSRP systems. Here, the inherent difficulty of the sequencing problem of a linear pipeline with different tasks will be proved first.

Theorem 1. Under the assumption that a task should not be initiated if it would cause a collision inside the system, the minimum execution time of a given task system (with different execution time vectors) is given by

$$\min_S \left[\sum_{(i,j) \in S} d_{ij} + \sum_{j=1}^{N} t_{rj} \right]$$

where S is an ordering of the tasks to be executed, t_{ij} is the execution time of task i on facility j, d_{ij} is the delay caused by initiating task j after task i, and r is the index of the last task in the sequence S.

Proof. (See Fig. 4 for illustration.) $\sum_{(i,j) \in S} d_{ij}$ provides the total delay in initiating all of the tasks. The last term $\sum_{i=1}^{N} t_{ri}$ will therefore complete the total execution time of the tasks

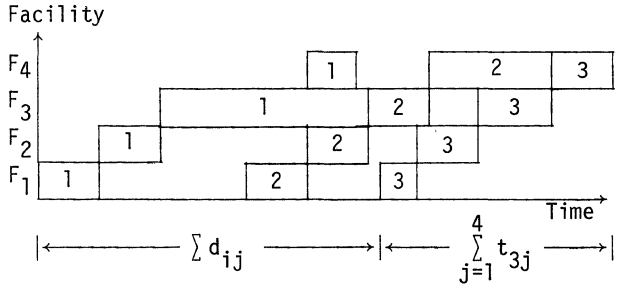


Fig. 4 Execution times of a linear pipe with variable execution times

by the pipeline. (Observe that the collision avoidance assumption here guarantees the same ordering of tasks as they leave the pipe.) Therefore the minimum execution time of the task system corresponds to the minimum of the expression for a certain sequence S. Q.E.D.

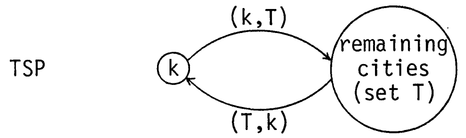
In exploring the intrinsic difficulty of this problem, let us be more general and assume that we want to minimize the execution time of a task system on a 'perturbed' pipeline. The perturbation is used to describe that there is still some previous task executed on the pipe. It therefore fits very well in a local optimization scheme of task systems in a stochastic environment where a continuous stream of task systems is available for sequencing at some time intervals. Situations where there is no perturbation can be taken care of by ignoring this parameter. Under this assumption, the following theorem is derived.

Theorem 2. The aforementioned sequencing problem is inherently difficult.

Proof. The traveling salesman problem (TSP) proved to be inherently difficult (Lemma 1) and can be shown reducible to it. Optimize

$$\min_S \left[\sum_{(i,j) \in S} d_{ij} + \sum_{j=2}^{N} t_{rj} + d_0(S) \right]$$

where $d_0(S)$ is the perturbation (measured in delay) due to previous task system to the sequence S. Now the similarity between this problem and the TSP suddenly reveals itself. Observe that the sequencing problem is actually equivalent to finding a cheapest trip through m+1 cities once and only once starting at some city (perturbed state) and then leaving the last one to a sink with a cost of $\sum_{j=1}^{N} t_{rj}$. Therefore a given TSP can be reduced easily by adding a fictitious node t and arc (i,t) for all nodes i in the original TSP with costs d_{it} by choosing an arbitrary starting node p. The resulting specification is solved as a sequencing problem whose optimal solution is obviously an optimal solution for the TSP (with node p as the perturbation state) because any shortest trip through the m+1 cities to the sink t will correspond to a shortest tour through the m+1 cities (see Fig. 5). To complete the proof, we have to



Transformed into a sequencing problem as: (starting at node k) cheapest trip from k, through T to sink s

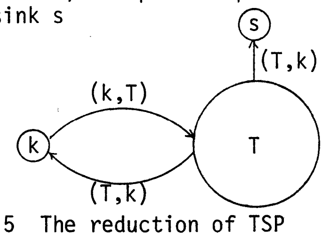


Fig. 5 The reduction of TSP

show how to derive the specification for the sequencing problem given the specification of a TSP of the reduced form. The procedure is done inductively. Assume it has completed k cities. For the (k+1)th city, extend the k-task system table to a (k+1)-task system as follows:

Task \ Facility	$1 \cdots\cdots n$	$n+1 \cdots\cdots n+2k$
1		
2	$[s_i^j]$	$\vdots$
$\vdots$		
k		
k+1	.....	.....

For the $(k+1)^{th}$ city, assign

t_i^j = execution time of the i^{th} task on facilities 1 through j, j = 1,2,...,N

s_i^j = (slack) execution time of task i on facility j.

Then let

$$s_{k+1}^{n+1} = t_1^n$$

$$\left.\begin{aligned} s_{k+1}^{n+2i-1} &= \max\{0, t_i^{n+2i-3} - t_{k+1}^{n+2i-2}\} \\ s_i^{n+2i-1} &= \max\{0, t_{k+1}^{n+2i-2} - t_i^{n+2i-3}\} \end{aligned}\right\} \begin{matrix}\text{for}\\ i = 2,3,\ldots,k\end{matrix}$$

$$\left.\begin{aligned} s_i^{n+2i} &= d_{i,k+1} \\ s_{k+1}^{n+2i} &= d_{k+1,i} \end{aligned}\right\} \text{for } i = 1,2,\ldots,k$$

(d_{ij} are city distances in TSP).

All other unspecified $s_i^j = 0$.

Then the complete task system for (k+1) tasks will be specified, and inductively, their delays after one another are precisely the respective distances. By construction, the starting city t represents the perturbed state (task 1) of the pipe which is initially present and causes delay to any sequence denoted by $d_0(S)$. To complete the picture, the arc distances to the sink d_{it} have to be modeled into the tasks system. Suppose the number of facilities so far is q. Compute

$$t_0 = \max_{i=1,\ldots,m+1} \{t_i^q\}$$

and let

$$s_i^{q-i} = t_0 - t_i^q$$

$$s_i^{q+2i} = d_{it} \ .$$

With this, the return distances from any node to the starting node p is modeled into s_i^{q+2i} while all tasks will have the same remaining execution time from facility 2 to facility q+m+1. The return distances are included as additional execution time on some later facilities. If the sequencing problem has a fast algorithm, so does the TSP. Q.E.D.

This theorem therefore asserts that even for a linear pipeline, if the task execution time on a facility is variable, then optimal sequencing under collision avoidance assumption is inherently difficult. Consequently, the optimal sequencing for a nondeterministic RSRP system under similar situations will also be inherently difficult.

If, however, the facilities have fixed speeds, will the optimal sequencing problem be simpler? Two different cases will be studied, and in both cases, static and dynamic RSRP systems, optimal sequencing is inherently difficult in general.

Theorem 3. The sequencing problem for a static RSRP system with reconfiguration cost is inherently difficult.

Proof. Observe that if more than one pipe can process some task, then trivially from D(3), the problem will be inherently difficult. So let us assume this is not so. Again, a reduction from the TSP will be used. Recall a static RSRP system permits one active pipe or configuration at one time. If a different configuration is needed, an extra amount of waiting for flushing the system and establishing the desired configuration will be necessary. Let

O_{ij} = overhead of i^{th} configuration to the j^{th} configuration.

Then given a task system with task i going through a pipe, say u(i), the total execution time will be minimized if and only if

$$\sum_{(u^{-1}(i),u^{-1}(j)) \in S} O_{ij} + t_r$$

is minimized where

t_r = execution time of the last task in the sequence S.

By a similar argument to Theorem 3, obviously

$$\text{TSP} = \text{static RSRP sequencing} \ .$$

The variable t_r corresponds to the distance from the last city visited to the original city. (Whether the perturbed state exists or not is irrelevant here. Also observe that no assumption has been made on the precedence relationship of the task system. The theorem holds whether or not this is empty.) Q.E.D.

So general optimal sequencing algorithms for static RSRP systems are complicated by prediction. Apparently, for the more flexible dynamic RSRP systems, where more overlapping among parallel pipes is allowed, the problem will be at least as difficult. Indeed it is so and can be cited as a theorem.

Theorem 4. Optimal sequencing in a dynamic RSRP system is inherently difficult.

Proof. A similar reduction procedure from the TSP can be constructed. First, given the specification of a TSP of m cities, add a fictitious sink node t and arc (t,i) with cost 0 and arc (i,t) with cost t_0 for all i = 1,...,m and some t_0 to be determined. Trivially, the solution of the resulting TSP will also yield the solution of the original TSP. Next try to reduce the resulting TSP to a sequencing problem in a dynamic RSRP system. The TSP is to minimize $\sum_{(i,j) \in S} d_{ij} + t_0$ where S is a sequence of traversals of the cities. The transformation is similar to that in Theorem 3. The procedure is done inductively, assuming having completed the task specification for k cities. Then for the (k+1)th city, expand the task table:

t_i^j = execution time of i^{th} task on facilities i through j,

s_i^j = (slack) execution time of task i on facility j.

Let

$$\left.\begin{aligned} s_i^{n+2i-1} &= \max\{0, t_{k+1}^{n+2i-2} - t_i^{n+2i-3}\} \\ s_{k+1}^{n+2i-1} &= \max\{0, t_i^{n+2i-3} - t_{k+1}^{n+2i-2}\} \end{aligned}\right\} \text{for } i = 2,3,\ldots,k$$

$$\left.\begin{aligned} s_i^{n+2i} &= d_{i,k+1} \\ s_{k+1}^{n+2i} &= d_{k+1,i} \end{aligned}\right\} \text{for } i = 1,2,\ldots,k$$

$$s_{k+1}^{n+1} = t_1^n$$

After the RSRP system is completed for the m+1 cities, let us assume the number of facilities so far is q. Compute

$$t_0 = \max_{i=1,\ldots,m+1} \{t_i^q\}$$

and let

$$s_i^{q+i} = t_0 - t_i^q \qquad i = 1,\ldots,m+1$$

and all other unspecified $s_i^j = 0$. This completes the RSRP specification whose optimal sequencing solution turns out to be precisely $\sum d_{ij} + t_0$ because by construction, the delay of executing task j after task i is precisely d_{ij} and also each facility has the same speed if $d_{ij} = d_{ji}$ (which holds for a TSP in an undirected graph). Hence if the sequencing problem in a dynamic RSRP has a fast algorithm, so does the TSP. Q.E.D.

These results indicate the necessity of simple heuristics (near-optimal) to be used in sequencing under the different conditions discussed. Some simple heuristics will be discussed in the next section. Meanwhile a semi-exhaustive approach to generate an optimal sequence for a dynamic RSRP system will be included to complete the discussion. Its application may be justified when the RSRP system is implemented at a high level so that each facility is actually a large computing module for performing specific computations. Also, in some cases, static local optimization for RSRP systems may be used to increase the throughput. Then an optimal sequencing algorithm for statically sequencing the pipes will be needed. So the following optimal algorithm is included. First, a theorem has to be developed.

Theorem 5. When maximum overlap in execution among all functional pipes is desired, the execution time of a given task system S_f, S_r is bounded by

$$LB(S_f) = \max_i\{T_i(S_f) + \sum_{j\varepsilon S_r} t_{ij} + \min_{j\varepsilon S_r}[\sum_{k \text{ following } i} t_{kj}]\}$$

where

$T_i(S_f)$ = completion time of the partial schedule on facility i containing the set of tasks S_f,

t_{ij} = execution time of task j on facility i,

S_f = a partial schedule for the task in S_f,

S_r = remaining tasks to be scheduled.

Proof. $T_i(S_f)$ yields the time facility i becomes available for any task in S_r, and $\sum_{j\varepsilon S_r} t_{ij}$ corresponds to the minimum additional time to finish the remaining tasks on facility i. The term $\min_{j\varepsilon S_r} \sum_{k \text{ following } i} t_{kj}$ gives the time needed for the fastest task to leave the pipe after leaving the ith facility. Then their sum will naturally form a lower bound on the execution time of $\{S_f, S_r\}$. Q.E.D.

With the above lower bound, one could devise a branch and bound algorithm to locate the optimal sequence as follows. For simplicity, we will consider only a list sequencing method, that is, the tasks will be ordered in a list to be executed according to the priority indicated in the list. The extension to an exact initiation schedule can be easily established.

Algorithm Search. Let S = task system, $T_0 = \phi$, i = 1, $T_c = T_0$ and Mark T_0.

Step 1: Among the ready tasks in S not yet in T_c, say this set is $S_c = \{u_1,\ldots,u_p\}$. Create $T_i, T_{i+1},\ldots,T_{i+p-1}$ such that $T_{i+k} = (T_c, u_{k+1})$ for $k = 0,1,\ldots,p-1$. Obtain $LB(T_{i+k})$. Let i = i+p-1. For all T_{j_0} ($j_0 < i$) such that $|T_{j_0}| = |S|$, a feasible solution has been found. Fathom (discard) all T_j ($j < i$) such that $LB(T_j) \geq LB(T_{j_0})$.

Step 2: Among all T_j ($j < i$) unfathomed and unmarked, choose one with smallest $LB(T_{j_1})$ and let $T_c = T_{j_1}$. Mark T_{j_1} and repeat step 1. If no other is available, the only feasible solution unfathomed will be the optimal solution. So halt. This procedure obviously will halt since there are only N! possible sequences and there always remains one feasible solution unfathomed.

The inherently difficult characterization propels one to believe that optimal sequencing in the dynamic situation may involve enormous overhead which causes a degradation in performance instead. Even after a task system (in a deterministic or partial stochastic sense such as in a lookahead type of sequencing) is identified to be sequenced, any optimal sequencing strategy developed for the general case, as the characterization is conjectured, will incur some decision discipline that takes a long time (if implemented by software means) or a large additional cost of hardware (if implemented by hardware and firmware mechanisms) or both. Also, what is optimal in a local task system may not be optimal in a more 'global' or larger task system that the former belongs. Under these circumstances, naturally a simple and near-optimal heuristic is often more advantageous. In view of this, the next section will be devoted to the comparison of some heuristics.

5. Sequencing Heuristics

In this section, sequencing of ready instructions in a semi-stochastic environment will be discussed. The term semi-stochastic is used to mark the fact that tasks or instructions are sequenced in a fixed burst under some lookahead scheme. So complete deterministic knowledge of the task systems (instructions) will be unavailable. The realism of this modeling assumption is easily conceivable because in the continuous behavior of the real world, a deterministic and finite model often is insufficient.

Three heuristics will be of particular interest here. They will be named First Come First Served (FCFS), Clustering, and RSRP Clustering. Their special features will be described and performance under memory conflict free situations compared using some experimental simulation. Their implementation using hardware and firmware control will also be included.

1. First Come First Served. As the name implies, FCFS discipline will allow the tasks or instructions to enter the RSRP system in the same ordering as they have arrived. So it is the simplest heuristic possible and its implementation schemata can be sketched as in Fig. 6. The initiation control is responsible for allowing the task or instruction at the end of the queue to enter the system at the correct moment to avoid collision inside or to allow proper reconfiguration to take place in the static RSRP system. The task queue will be monitored by the initiation control and there is little additional hardware or firmware to perform any reordering. Its performance can then be referred to as one achievable with the cheapest cost and legitimately it may be chosen to filter out other heuristics that are more costly but not much superior in performance to FCFS.

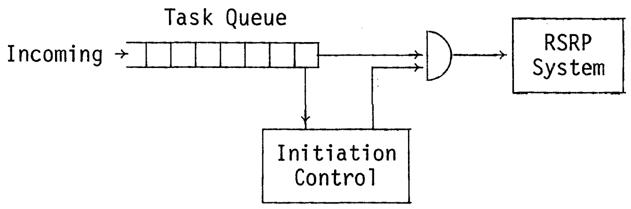


Fig. 6 FCFS Sequencing

2. Clustering. In a RSRP system, reconfiguration due to different types of instructions or tasks incur extra overhead and delay to the normal stream of execution. Specifically in a static RSRP system, if a task (instruction) has to flow through one pipe different from the current one in the system, it has to wait for some latency period until the latter has emerged, as in the arithmetic unit pipe of the TIASC system. So a sensible approach to remedy the situation is to reduce the occurrence of reconfiguration as much as possible. This stems the reasoning behind the 'clustering' heuristic where ready instructions or tasks that involve the same configuration or path are grouped together to be executed one after the other. So clustering really involves a scanning and grouping mechanism and its implementation can be as depicted in Fig. 7.

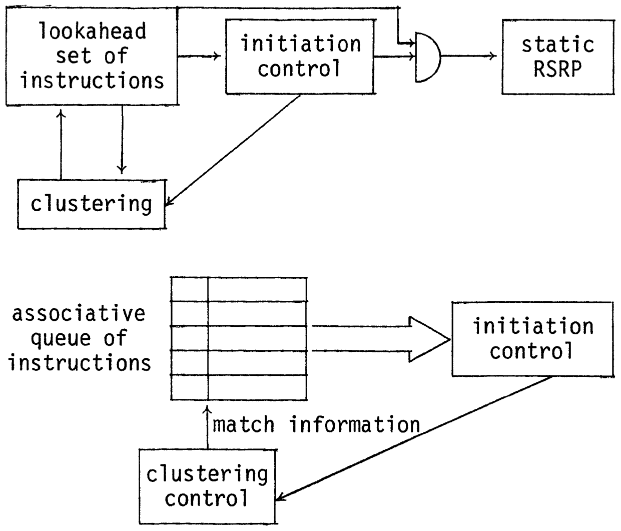


Fig. 7 Clustering

The additional hardware and control circuitry needed in this implementation include an associative queue rather than an ordinary queue for the set of lookahead instructions so that independent instructions are searched in parallel during execution in such a way that instructions of a same type are detected almost instantaneously and hence are available for the initiation control for controlling their entrance to the static RSRP system. For the other parts of the sequencing modules, no significant deviation from the previous scheme is necessary (except the synchronization clock pulses in the initiation control and the additional clustering unit which will change its associative match word from time to time based on signals from the initiation control). With the aid of the associative queue, prolonged delay due to retrieving or detecting of clustered instructions is avoided. Hence, this sequencer can function almost as quickly as the FCFS discipline. In addition, observe the static control overhead of clustering is primarily a linear function of the size of the task system in the lookahead set since it merely involves some additional associative registers.

3. RSRP Clustering. The same clustering heuristics may be applied to a dynamic RSRP system where concurrent processing among the various functional pipes are allowed. In many cases, grouping of tasks of the same type in a dynamic RSRP system still is advantageous when tasks of the same type usually incurs less latency. The routing of operands in a dynamic RSRP system is a bit more complicated than that in a similar but static RSRP system because a correct transition at a shared resource has to be chosen dynamically rather than statically. A localized monitor scheme for this routing is exemplified in Fig. 8. Each data packet will contain some redundancy holding encoded information about the path desirable. This encoded path information will be used by the second part, the decoding control at each shared facility (one with multiple exit arcs), to enable the correct transitions. Since this decoding activity can be performed in parallel with the actual processing,

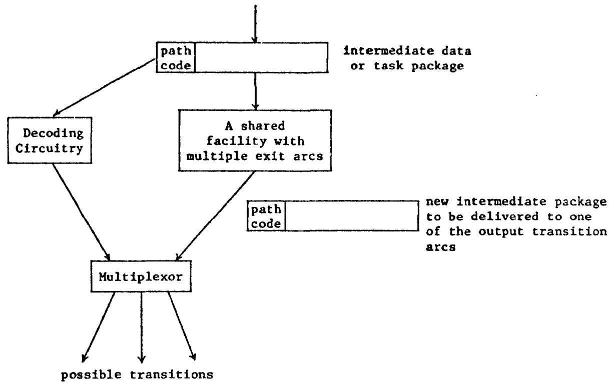

path code
intermediate data or task package
Decoding Circuitry
A shared facility with multiple exit arcs
path code
new intermediate package to be delivered to one of the output transition arcs
Multiplexor
possible transitions

Fig. 8 Localized monitor scheme in dynamic RSRP system

there is no apparent dynamic runtime overhead involved which may delay the availability of an output. Also since multiplexors are used to choose correct transitions at a static RSRP system in any case, the overhead discussed above is really quite tiny. The schematic diagram of RSRP clustering is exactly the same as that of the clustering method except in the initiation control, a two-dimensional collision matrix constructed by the algorithm in Fig. 2 is also provided. The matrix can be stored in shift registers or counters whose contents are constantly updated to control the initiation of tasks (instructions) already re-ordered.

6. Experimental Demonstration

The three heuristics (2 for static and 1 for dynamic RSRP systems) were tested on RSRP systems whose configurations are taken directly from the arithmetic unit pipes of TIASC and the floating point pipes of the CDC STAR-100 systems. The environments are parameterized in three aspects. First, the different types of tasks, in this case the instructions, are given some relative frequency of excitation. For instance, (0.1,0.2,0.4,0.1,0.2) implies the percentage of instructions executed are 0.1, 0.2, 0.4, 0.1, 0.2 for the five types (configurations) respectively. Second, the size of the lookahead set of instructions is variable. This marks a variable structure in the semi-stochastic sequencing discipline explained in the previous section. Third, the nature and amount of interdependency or precedence relationships of the instructions (mainly in operands) as they are generated are parameterized such that the amount of interaction and levels of dependency within and between lookahead sets of instructions are encompassed. Therefore, a stochastic precedence relationship is also allowed in the simulation model.

With these three types of parameters, the heuristics can be compared under different RSRP systems. The simulator built mainly consists of three parts.

(1) Instruction generator which generates the instructions according to the parameters specified. (A random number generator is used particularly to create instructions according to the mix ratio, dependency parameters, etc.)

(2) Collision matrix constructor which constructs the two-dimensional collision matrix given a RSRP system specification (including paths, speeds) according to the algorithm described in Fig. 2.

(3) Heuristic sequencers which simulate the hardware sequencers in Figs. 6 and 7 according to the sequencing discipline chosen and monitor the execution of the instructions. The output of the simulator consists of a (time-driven) execution profile of the instructions as they are generated and executed under the three heuristics adopted so that they can be compared easily.

A typical comparison is shown in Fig. 9. The horizontal axis gives the number of iterations (one iteration corresponding to the execution of the ready instructions in a lookahead set of instructions) and the vertical axis the corresponding execution time profile. This particular output illustrates that indeed the clustering philosophy is very useful compared to FCFS since it brings a reduction in execution time by 30%. But the dynamic RSRP system under the same clustering rule is

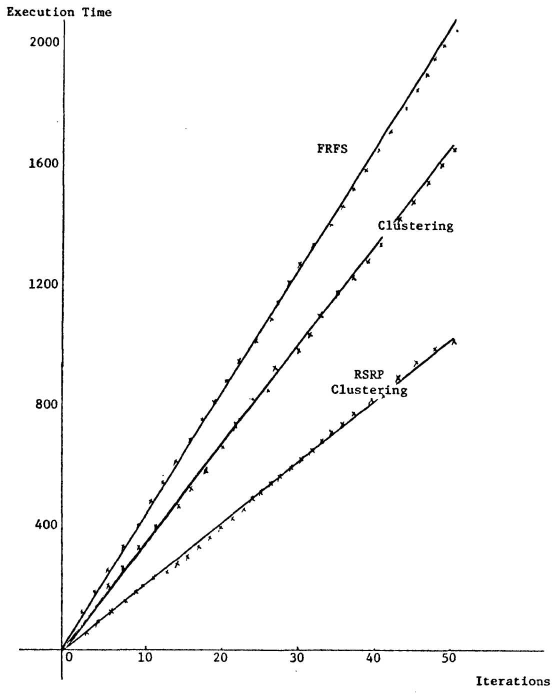


Fig. 9 A Typical Comparison (STAR-100 Pipe 1)

even more attractive as it further reduces the execution time by as much as 40%. To compare the three cases, a relative efficiency index is set up. Let

α_{ij} = relative efficiency of heuristic j with respect to heuristic i

$\triangleq T_i/T_j$

where T_i = execution time of the instructions under heuristic i (observe $\alpha_{ij} = \alpha_{ik}\alpha_{kj}$). The results of the comparisons under different parameters for the three cases are tabulated in Figs. 10a, b, c. From it, several observations are to be discussed.

STAR-100 Pipe 1

L	T_1 FRFS	T_2 Clustering	T_3 RSRP	T_2/T_1	T_3/T_2	θ	ϕ
Mix = (0.2,0.2,0.1,0.2,0.1,0.2)							
8	2056	1672	1004	0.815	0.605	0.5	0.4
16	3936	2797	1252	0.71	0.45	0.5	0.4
32	7195	4040	1780	0.562	0.44	0.5	0.4
8	2095	1661	973	0.795	0.575	0.3	0.4
16	3820	2794	1135	0.73	0.407	0.3	0.4
32	7509	4227	1729	0.564	0.409	0.3	0.4
8	1709	1340	869	0.785	0.658	0.3	0.6
16	3499	2578	1147	0.739	0.445	0.3	0.6
32	6672	3920	1605	0.59	0.41	0.3	0.6
Mix = (0.3,0.2,0.1,0.2,0.1,0.1)							
8	1608	1330	888	0.83	0.552	0.5	0.4
16	3227	2433	1187	0.745	0.49	0.5	0.4
32	6027	3571	1724	0.593	0.483	0.5	0.4
8	1793	1546	923	0.865	0.595	0.3	0.4
16	3734	2771	1251	0.74	0.46	0.3	0.4
32	6979	3951	1962	0.566	0.498	0.3	0.4
8	1816	1427	943	0.785	0.65	0.3	0.6
16	3628	2611	1316	0.71	0.501	0.3	0.6
32	6933	3906	1883	0.572	0.475	0.3	0.6

Fig. 10a

STAR-100 Pipe 2

L	T_1 FRFS	T_2 Clustering	T_3 RSRP	T_2/T_1	T_3/T_2	θ	ϕ
Mix = (0.2,0.2,0.2,0.1,0.2,0.1)							
8	4143	3377	2997	0.82	0.89	0.5	0.4
16	7795	5191	3318	0.67	0.64	0.5	0.4
32	13369	7291	3850	0.545	0.53	0.5	0.4
8	4277	3493	2859	0.815	0.82	0.3	0.4
16	7170	5148	3112	0.716	0.61	0.3	0.4
32	14180	7462	3826	0.53	0.515	0.3	0.4
8	3119	2524	2472	0.81	0.97	0.3	0.6
16	6582	4826	3094	0.734	0.645	0.3	0.6
32	12386	6960	3572	0.574	0.508	0.3	0.6
Mix = (0.3,0.1,0.3,0.1,0.1,0.1)							
8	3900	3044	2851	0.78	0.935	0.5	0.4
16	6144	4251	3237	0.676	0.76	0.5	0.4
32	11436	6797	3990	0.594	0.59	0.5	0.4
8	3142	2636	2430	0.805	0.85	0.3	0.4
16	6287	4594	3107	0.73	0.665	0.3	0.4
32	11894	6808	3905	0.575	0.575	0.3	0.4
8	2457	1977	1950	0.805	0.98	0.3	0.6
16	5515	4072	3141	0.74	0.77	0.3	0.6
32	10160	6274	3657	0.62	0.58	0.3	0.6

Fig. 10b

(1) α_{ij} is usually sensitive to the size of the lookahead set and the individual system tested. This is readily explainable because with more instructions clustered (which depends on the size of lookahead) at one, fewer reconfigurations may be necessary. Since the amount of concurrent processing possible is limited by the system structure, the latter dependency is also reasonable.

(2) α_{ij} is quite insensitive to other parameters such as instruction mix ratio, and dependency

TIASC Results

L	T_1 FRFS	T_2 Clustering	T_3 RSRP	T_2/T_1	T_3/T_2	θ	ϕ
8	1190	981	811	0.824	0.825	0.5	0.4
16	2050	1373	1112	0.67	0.81	0.5	0.4
32	4312	2101	1830	0.488	0.87	0.5	0.4
8	1297	993	801	0.766	0.806	0.3	0.4
16	2210	1405	1060	0.64	0.73	0.3	0.4
32	4180	2084	1767	0.5	0.845	0.3	0.4
8	1010	852	737	0.84	0.868	0.3	0.6
16	1885	1276	975	0.677	0.767	0.3	0.6
32	3861	1934	1575	0.52	0.815	0.3	0.6
8	1034	850	762	0.824	0.895	0.3	0.6
16	1975	1322	998	0.78	0.805	0.3	0.6
32	3766	1906	1501	0.506	0.79	0.3	0.6
8	1267	986	791	0.78	0.805	0.3	0.4
16	2189	1411	1049	0.642	0.742	0.3	0.4
32	3981	2021	1590	0.51	0.786	0.3	0.4
8	1079	945	866	0.865	0.91	0.5	0.4
16	2038	1344	1072	0.66	0.798	0.5	0.4
32	4092	2034	1134	0.499	0.81	0.5	0.4

Fig. 10c

structure. These two parameters have the same common characteristics; they tend to limit the amount of independent instructions of each type to be executed. With a reasonable lookahead set size, they influence the three heuristics to a relatively similar extent.

(3) In particular, $0.6 \le \alpha_{21} \le 0.8$ for 90% of the cases, hinting the clustering discipline is really beneficial compared to FCFS in a static RSRP system. But $\alpha_{32} \le 0.7$ for most cases in the STAR models and $\alpha_{32} \le 0.8$ for most cases in the TIASC model further implies the advantages of a dynamic RSRP system over a static one under the same clustering discipline.

Conclusion

RSRP design represents a powerful organization that embeds both parallel and pipeline processing characteristics. However, optimal sequencing in either a static or a dynamic RSRP system has been proven to fall into the 'inherently difficult' characterization and it is unlikely to possess a fast algorithm. An optimal sequencing algorithm in general is too complicated to be implemented by hardware or software. So in practice, simple heuristics may be more advantageous. These heuristics can be implemented with sufficient hardware support such that speed-up of execution is attainable. In particular, the clustering discipline has been demonstrated to be valuable to reduce reconfiguration overhead while a dynamic RSRP scheme introduces additional advantages over a static scheme because full concurrent processing among parallel paths is permissible.

References

[1] M. Flynn, "Some computer organization and their effectiveness," IEEE Trans. on Computers (Sept. 1972), pp. 948-960.

[2] G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick and R. Stokes, "The ILLIAC IV computer," IEEE Trans. on Computers (Aug. 1968), pp. 746-757.

[3] J.A. Rudolph, "A production implementation of an associative array processor -- STARAN," Proc. FJCC 1972, pp. 229-242.

[4] J.L. Troy, "Computer simulation of PEPE and its host at the instruction level," 1973 Sagamore Computer Conference on Parallel Processing.

[5] D.W. Anderson, F.J. Sparacio and R.M. Tomasulo, "The IBM system/360 model 91: machine philosophy and instruction handling," IBM J. Res. and Develop. (Jan. 1967), pp. 8-24.

[6] W.J. Watson, "The TI-ASC - a highly modular and flexible super computer architecture," AFIPS FJCC 1972, pp. 221-230.

[7] R.G. Hintz and D.P. Tate, "Control Data STAR-100 processor design," COMPCON 1972.

[8] C.V. Ramamoorthy, K.M. Chandy and M.J. Gonzalez, "Optimal scheduling strategies in a multiprocessor system," IEEE Trans. on Computers (Feb. 1972), pp. 137-146.

[9] K.M. Chandy and J.R. Dickson, "Scheduling unidentical processors in a stochastic environment," Proc. COMPCON 73, pp. 171-174.

[10] R.L. Graham, "Bounds on multiprocessing and timing anomoly," SIAM J. Appl. Math. (March 1969).

[11] T.C. Hu, "Parallel sequencing and assembly line problems," Oper. Res. (Nov. 1961), pp. 841-848.

[12] R.R. Muntz and E.G. Coffman, "Optimal preemptive scheduling on two-processor systems," IEEE Trans. on Computers (Nov. 1969), pp. 1014-1020.

[13] S.M. Johnson, "Optimal two- and three-stage production schedules with set-up time included," Naval Res. Log. Qtly (1954).

[14] A. Holt and P. Commer, "Events and conditions -- a. an approach to the description and analysis of dynamic systems, b. marked graph mathematics."

[15] C.A. Petri, "Kommunication mit automaten," trans. in Project MAC M-212 Report, originally published in 1962.

[16] Texas Instrument Inc., "A description of the Advanced Scientific computer system" (April 1973).

[17] Control Data Corp., "Control Data STAR-100 computer hardware reference manual" (1974).

[18] C.V. Ramamoorthy and H.F. Li, "Efficiency in generalized pipeline networks," Proc. NCC 1974, pp. 625-635.

[19] E. Davidson, "The design and control of pipeline function generators," Stanford Report.

[20] S. Cook, "The complexity of theorem proving procedures," Conf. Rec. of 3rd ACM Symp. on Theory of Computing (1970), pp. 151-158.

[21] R.M. Karp, "Reducibility among combinatorial problems," TR-3, Dept. of Comp. Sci., U.C. Berkeley (April 1972).

[22] H.F. Li, "A structured theory of parallel pipelined systems," Ph.D. dissertation, U.C. Berkeley (Dec. 1975).

TIME-SHARED MEMORY-PROCESSOR INTERFACE

Per-Erik Danielsson
Björn Gudmundsson
Department of Electrical Engineering
Linköping University
Linköping, Sweden

Abstract -- In multiprocessors the memory interface can be designed either as a two-level structure (multiports) or as a single time-shared bus. A nonexpensive reasonably fast synchronous bus system is proposed and has been tested up to a minimum of 60 ns bus time slice. The dynamic behaviour for different configurations of processors and memory modules has been subjected to simulation. The results show that if the bus time slice is short enough the single bus property contributes very little to performance degradation. Queues tend to develop at the memory modules when the number of processors increases. Diagrams illustrate how this can be compensated for by dividing the memory space into smaller modules.

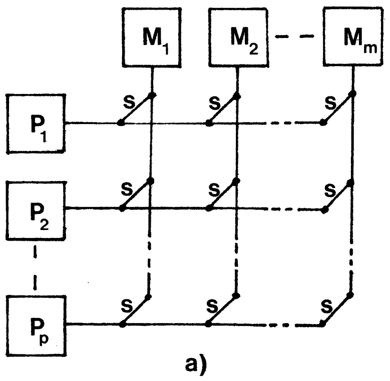
Hardware

Connection alternatives

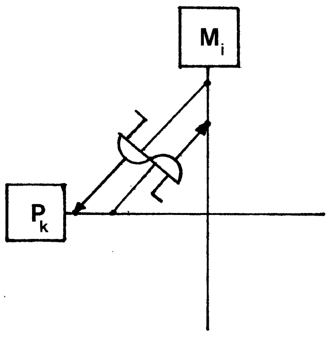
Due to some authors [1,2,3] there are three principally different ways of interfacing memories and processors in a multiprocessor system. These are by means of

- o a crossbar switch system
- o multiport memory modules
- o a time-shared common bus

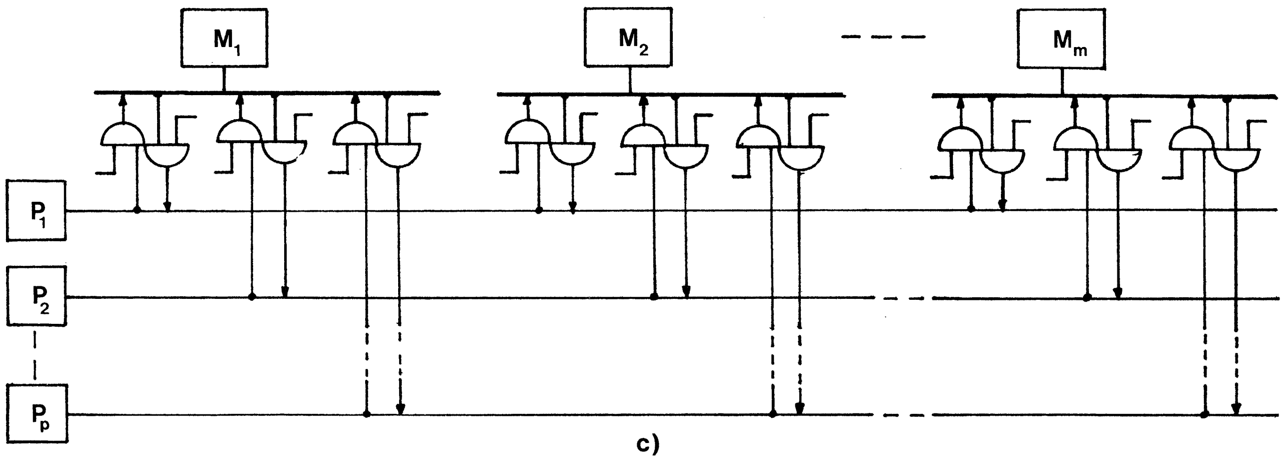
First, we demonstrate that the two first cases are essentially the same and that the real choice is between a multiport memory and one single time-shared common



a)



b)



c)

Fig 1. The crossbar switch reduced to multiports

bus.

The crossbar switch system is usually depicted as in Fig 1a). In a certain moment any processor k can be connected to any memory module i without interfering the communication to other modules. However, in the given context the S-switches are abstractions, concepts borrowed from the world of relays and contact networks. Electronic switches are unidirectional gates and the S-switches must ultimately be transformed into a bus system like Fig 1b). Needless to say, the gates have to be controlled by conflict resolving logic. Fig 1c) shows the same gates but the bus-system is somewhat redrawn with preservation of the topology. The subsystem of gates that performs the fanning-in and fanning-out at the outskirt of each memory module is now readily seen to be identical to a multiport. As far as the authors can see, this simple argument shows that the so called crossbar switch can be omitted from further discussions.

Fig 1c) has a two level structure with one bus for each processor and one bus for each memory module. Normally, only the processor busses are elongated as physically observable lines. The memory module busses are usually concealed in a device that is called a multiport or multiport switch, multiplexer, priority system, arbiter or the like.

Time conflicts has to be resolved at each one of these busses. In some computers the processors can be assumed never to make another memory request until the present one has been effectuated. It then immediately follows that time conflicts on the receiving processor bus are automatically resolved. For the memory module bus no such assumptions can be made. Any combination of the p processors can request the same memory module at the same time and therefore every memory module must be equipped with a time conflict resolver.

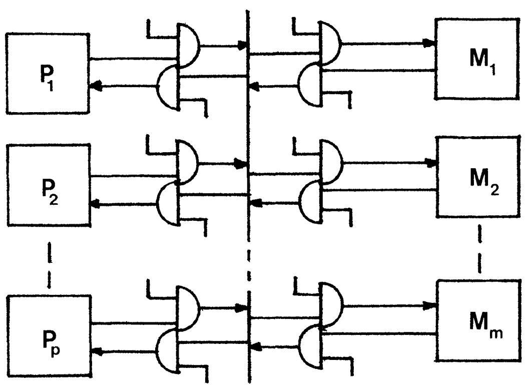


Fig 2. The single time-shared common bus

Multiport memory modules tend to be very expensive in a multiprocessor environment since the number of gates is 2(p·m). The time-shared common bus in Fig 2 contains only 2(p+m) gates. The rest of the paper deals with this alternative.

Serial versus parallel transmission

Questions may arise whether the single bus will bring a high penalty in terms of speed reduction and/or a very complex time conflict resolver. It will be demonstrated below that neither need to be the case. Also there are many trade-off situations in the actual bus design. A very significant choice is the bus width. If we are interested in "normal" processor speed the bit-by-bit serial transfer can be ruled out at once, but the alternatives in Fig 3 deserve a short discussion.

P	= processor
A	= address
M	= memory
D_{in}	= data to be stored in memory
D_{out}	= data fetched from memory

a) $P \; \begin{array}{c} \xrightarrow{A} \\ \xrightarrow{D_{in}} \\ \xleftarrow{D_{out}} \end{array} \; M$ b) $P \; \begin{array}{c} \xrightarrow{A,D_{in}} \\ \xleftarrow{D_{out}} \end{array} \; M$

c) $P \; \begin{array}{c} \xrightarrow{A} \\ \xleftrightarrow{D_{in},D_{out}} \end{array} \; M$ d) $P \; \xleftrightarrow{A,D_{in},D_{out}} \; M$

Fig 3

Fig 3a) is a maximally parallel system. One memory module i may be receiving data over the bus from processor k at the very same moment as another memory module j delivers data to another processor l. In Fig 3b) address and data are supposed to arrive in sequence to the memory. Now, it is generally agreed that in an ordinary job mix for a CPU no more than 20% of the memory references are WRITE operations. This means that in four out of five cases only address information is transmitted from the processor. When IO-processors are taken into account the relative number of WRITEs will increase and in the case Fig 3c) will give a more balanced situation. However, time conflict resolving in Fig 3c) is somewhat more complicated. In 3a) and 3b) the potential senders on one bus are the processors only (for A and D_{in}) or memory modules (for D_{out}). In Fig 3c) however, both processors and memory modules are acting as senders on the data bus.

A proposal for a synchronous bus

In our research project we decided to investigate the case of Fig 3a). In order to get a thorough understanding we designed and built a laboratory model of the bus and interface. The design is shown by Fig 4 in the case of four processors and four memory modules. We assume 16 bit for both address and data. The system contains an internal clock and may be regarded as a synchronized sequential network. All incoming signals and all registers and flip-flops change state only within the permitted interval of the clock cycle.

A transfer is initiated by WRITE or READ requests from the processor. Such a request is gated out to the bus lines when the following conditions are fullfilled:

- o No processor with higher priority is making a request in the same clock cycle.
- o The address points to a non-busy module. This is checked by the MPX-unit.

Address and data (if WRITE request) are recieved and stored at the interface of the appropriate memory module which starts its own cycle immediately. After a while, during which the bus may have been used many times for other purposes, the memory module signals READY and if the priority condition is fullfilled a previous READ

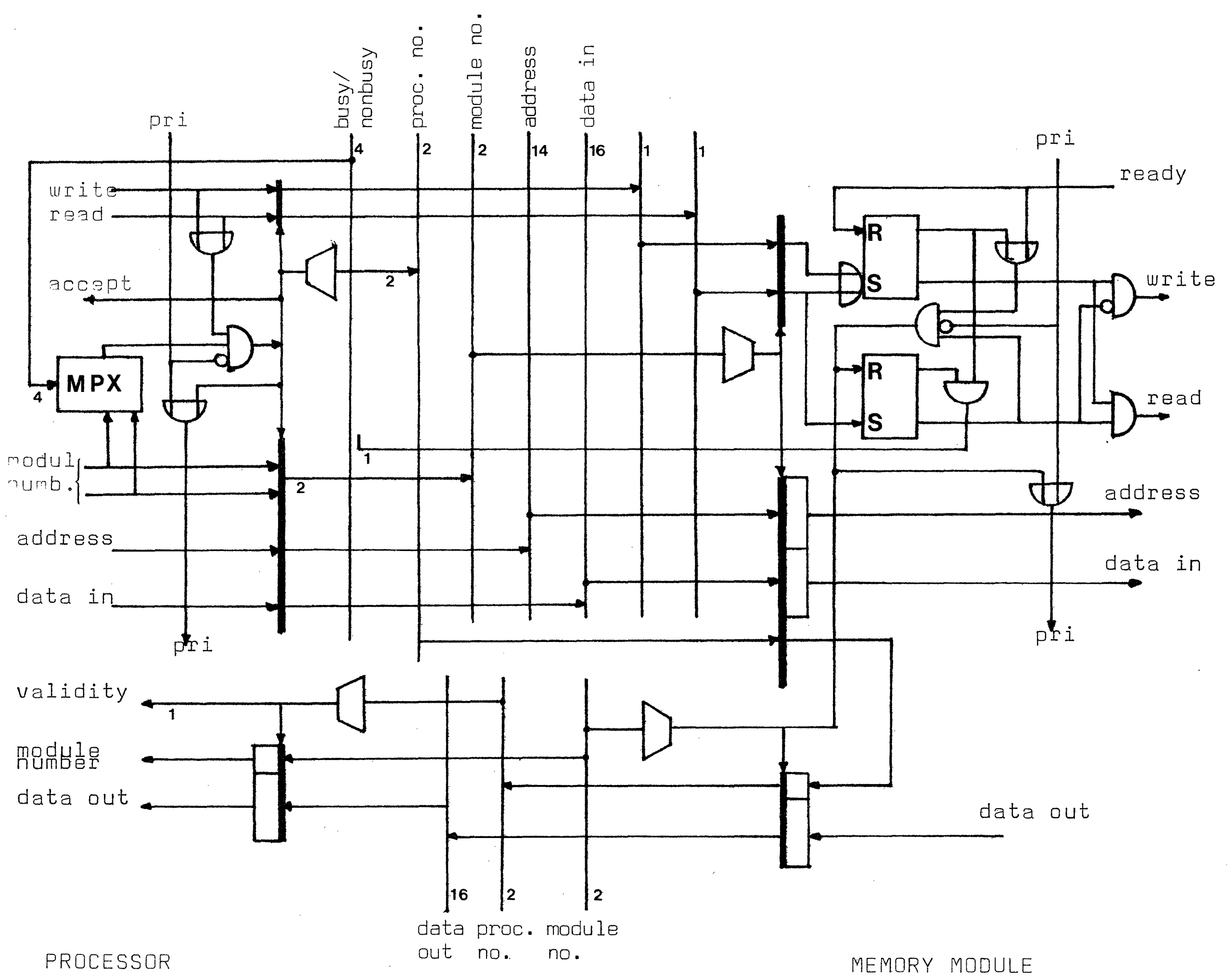


Fig 4. Time-shared memory-processor interface

request results in data transfer back to the appropriate processor. The two SR-flip-flops are reset and in the clock-cycle after READY the memory module signals NON-BUSY on its individual BUSY/NON-BUSY line.

In fact, this signalling is included in the longest delay path in the network. This path goes from the SR-flip-flops, back to the MPX-units, along the priority lines, via data output gates over the bus to the recieving register input. This signal path must settle in one clock cycle. In our experiment set-up we used a mixture of TTL and Shottky TTL-logic. The bus lines were about half a meter and we managed to get a realiable function at clock speeds up to 17 MHz. We conclude that a time slot of 80 ns is achievable in a time-shared memory multiprocessor interface. This is in accordance with results reported from real computer installations [3].

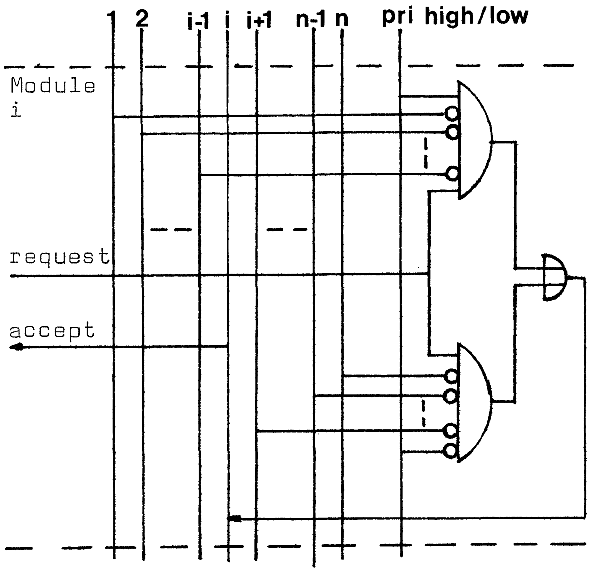


Fig 5. Accelerated two case priority logic

Fig 4 has a fixed priority between processors and memories. As will be shown below there are few cases where the low priority units will suffer from substantial delays. If necessary one may introduce a second priority chain counter-directed to the first one. A low frequency switch may couple and decouple these priority chains alternatively resulting in the same average priority for each unit. The priority chains of Fig 4 can be speeded up in a manner similar to carry acceleration in counters and adders. Fig 5 shows a double directed and accelerated priority logic.

A ring priority scheme can be implemented as shown in Fig 6. The central counter is part of the synchronized sequential network and in each clock cycle one and only one module is pointed at via the decoder. The priority chain is closed and, as can be seen in Fig 6, the module pointed at gets the highest priority. The counter "cuts" the chain at module i and by stepping the counter the priority is shifted among the modules in a round-robin fashion. A more general dynamic priority algorithm can be implemented if the counter is replaced by a register which can be loaded under program control.

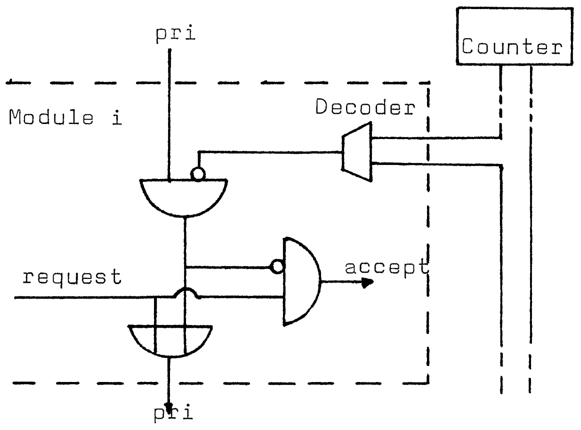


Fig 6. Ring priority logic

A completly reprogrammable (but time consuming) priority network has been proposed in [5].

Simulation experiments

Modelling

In order to get an idea of the performance of a time-shared memory interface a number of simulation experiments has been undertaken. In particular, the effects of varying the number of processors and memory modules connected to the interface has been studied.

The following parameters affect the performance of the system:

- o Clock cycle time (= bus time slot)
- o Distribution of processor requests
- o Memory cycle time
- o Bus configuration
- o Type of conflict resolving logic
- o Number of processors
- o Number of memory modules

The strategy in the simulation experiments has been to measure performance as a function of the number of processors and memory modules for different combinations of the remaining parameters. In order to get a manageable set of parameter combinations, the assumptions described below are made about the behaviour of the main parts of the simulation model.

Processors. Needless to say, this is the most difficult part to model. As our experiments are not intended to evaluate the effect of a time-shared bus on a particular processor, a fairly simple model of processor behaviour was used. It is hoped that the model chosen is general enough to give a rough picture of the dynamic behaviour of the time-shared bus.

The processors are identical (no I/O-activity) and they are modeled at the instruction execution level. There are four types of instructions which are selected in random order from a specified mix. Timing diagrams of the instructiontypes are shown below.

IR: Instruction Request
OR: Operand Request
WR: Write Request

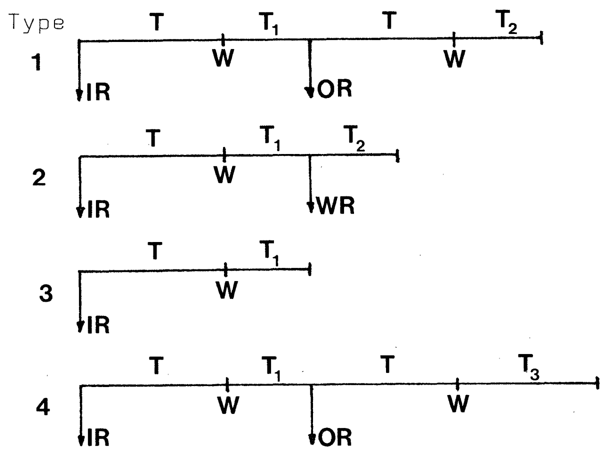


Fig 7. Instruction types

If the requested data has not been delivered at point W, the processor (microprogram) enters a wait loop. When the data arrives processing continues from point W. In the simulated configurations T is always less than memory access time. As can be seen in the timing diagrams there is no overlapping of consecutive READ-requests. This not being the case, serial correlation between destinations of successive requests will have little effect on the average access time as seen from the processor. Therefore, the memory modules are accessed completely at random. It is assumed that the length of all instructions and operands equals the wordlength of a memory module.

Bus configurations. Three bus configurations were simulated, and in the sequel they will be referred to as I, II and III.

I: Addressbus + one bidirectional databus (Fig 3c). Memory modules have priority over processors.

II: Addressbus + two unidirectional databusses (Fig 3a and Fig 4).

III: Multiport configuration. An analysis of this configuration with clock cycle time equal to memory cycle time is presented in [4].

Memory modules. The memory modules are assumed to be identical and of semiconductor type, i.e. accesstime equals cycle time. The variations in accesstimes are assumed to be so small that the accesstime as seen from the memory/bus interface is a constant multiple of clock cycles. This means that memory modules do not have to compete for the bus as only one module can be started in each clock cycle.

Results

The number of instructions that a simulated processor executes during a simulation run is taken as a measure of performance. This measure is normalized so that it relates to the performance of a single processor (no bus conflicts) connected via the time-shared bus to an infinite number of memory modules (no memory conflicts). Unrealistic as this configuration may be, it still serves as a reference point when comparing the relative effects of adding more processors or memory modules to a given system.

In the sequel the following abbrevations will be used:

p:	number of processors
m:	number of memory modules
Cl:	clock cycle time
A:	memory accesstime as seen from the processors. A = (nominal memory accesstime) + 2 · Cl
T_1, T_2, T_3:	processor speed (Fig 7)
η:	processor performance (as defined above)
C_m:	aver. number of clock cycles that processor requests are delayed by contention for memory modules.

C_b: d:o for delays introduced by contention for busses

λ: aver. number of memory modules started/clock cycle

Processor speed was kept fixed (T_1=200 ns, T_2=250 ns, T_3=1750 ns) while the other timing parameters were varied.

A fixed priority scheme was used and by averaging η over all processors we get the performance of the processors when the ring priority scheme shown in Fig 6 is used. Since all processors are identical, λ is proportional to the total number of instructions executed and we get $\eta = \lambda/p\lambda_0$ where λ_0 relates to the configuration with a single processor and an infinite number of memory modules.

When m is increased in a given configuration, C_m will of course decrease leading to a higher λ and thus a higher η. Since $C_b>0$ in bus configurations I and II we can never reach $\eta=1$ in these configurations although we can get arbitrarily close by reducing Cl and thus get $C_b \simeq 0$. However, in bus configurations I and II the capacity of the addressbus will set upper limits to λ and η if $p\lambda_0>1$, i.e. if there are enough processor to potentially saturate the addressbus. In these cases we get the following maximum values for λ and η:

Bus conf.	λ_{max}	η_{max}
I	<1	$<1/p\lambda_0$
II	1	$1/p\lambda_0$

The addressbus in configuration I can never become saturated (λ=1) since this would imply that no conflicts ever ocurred at the databus. As memory modules have priority over processors, a conflict at the databus means that in the next clock cycle a memory module will act as sender on this bus while a WRITE-request will be delayed and no memory module will be started in that clock cycle.

<u>Configurations where C_b is negligible</u>
In configurations with low p and with Cl short enough compared to processor speed, $C_b \simeq 0$. C_b is also negligible in configurations with short Cl and a high ratio p/m giving $C_m \gg C_b$. Diagrams 1, 2 and 3 relate to configurations with the following parameter values: A=960 ns, Cl=80 ns.

Instruction mix:	type 1	35%
	type 2	20%
	type 3	35%
	type 4	10%
Bus configuration:	II	

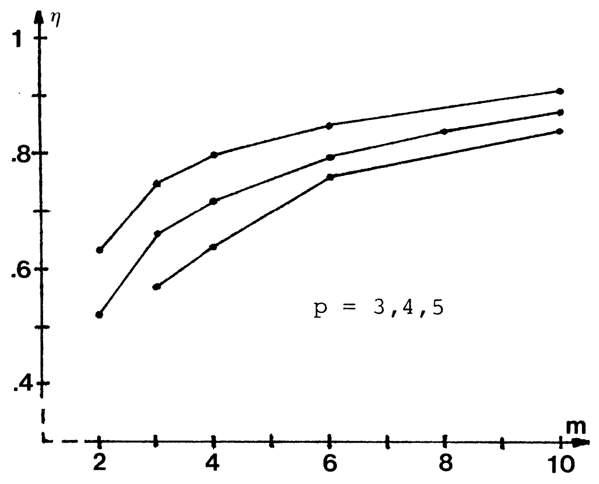


Diagram 1

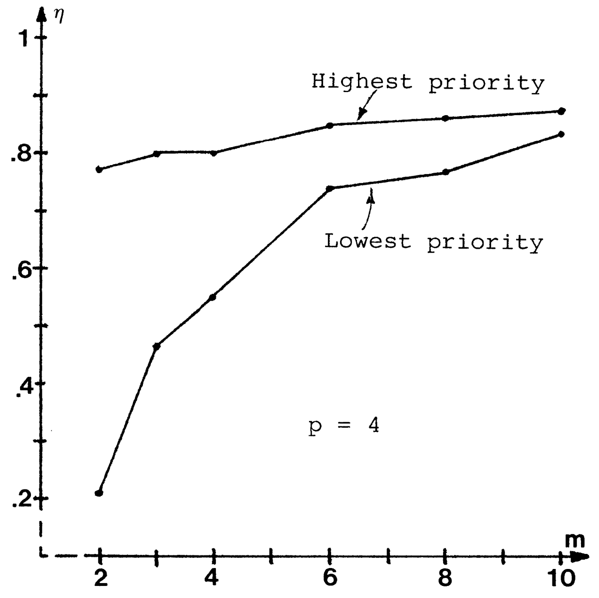


Diagram 2

Diagram 2 gives η for the processors with highest and lowest priority when a fixed priority scheme was used. For low m the processor with lowest priority suffers substantial delays. However, by increasing m the difference in performance between the low priority and the high priority processors is significantly reduced indicating that for the given parameter values C_b is negligible. When this is the case the following empirical expression seems to give a fairly good approximation of η:

$$\eta \simeq \exp(-\alpha p/m)$$

α is a constant that increases when the ratio between A and processor speed increases. In diagram 3 η is plotted versus p/m for the values of p and m in diagram 1. The solid line depicts $\eta = \exp(-0,32\ p/m)$.

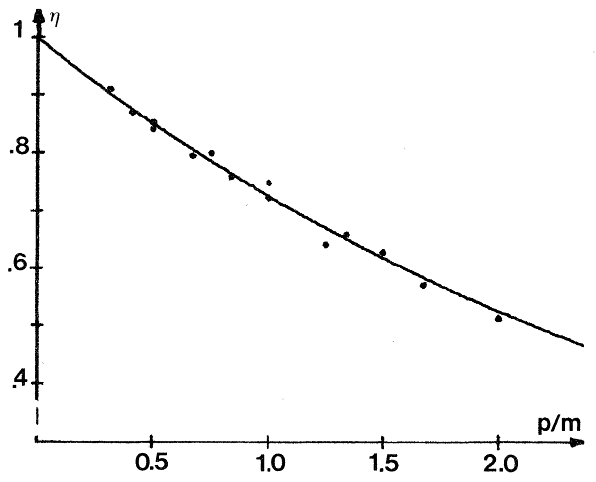


Diagram 3

Comparison of bus configurations I, II and III. For given p and m, the three bus configurations will give different C_b. Configuration I gives the highest C_b and thus the lowest η since it has a bidirectional databus. Diagrams 4 and 5 show a comparison of configurations I, II and III.

Instruction mix:	type 1	30%
	type 2	30%
	type 3	30%
	type 4	10%

Nominal memory accesstime was 400 ns, giving A=800 ns (diagram 4) and A=560 ns (diagram 5). Compared to processor speed and memory accesstime a clock cycle of 200 ns is somewhat unrealistic, but it was used in order to emphasize the effects of contention for the busses.

Increasing m leads to a higher λ which for configurations I and II means increasing C_b. However for high ratios p/m the differences in η will be negligible since here $C_m >> C_b$. By making Cl shorter we reduce C_b and as can be seen in diagram 5 the differences in η become very small.

Configurations with high C_b. We shall now look at some cases where λ, and thus C_b, is high. Also, we shall see how the performances of the individual processors are affected as λ is increased.

Diagram 6 relates to configurations with the same instruction mix and nominal memory accesstime as in diagrams 4 and 5.

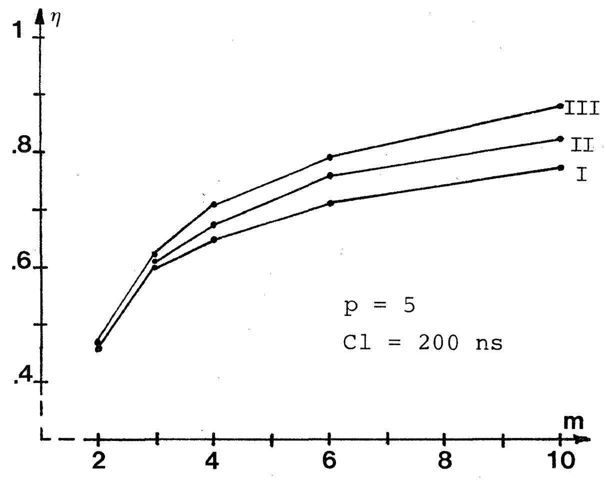


Diagram 4

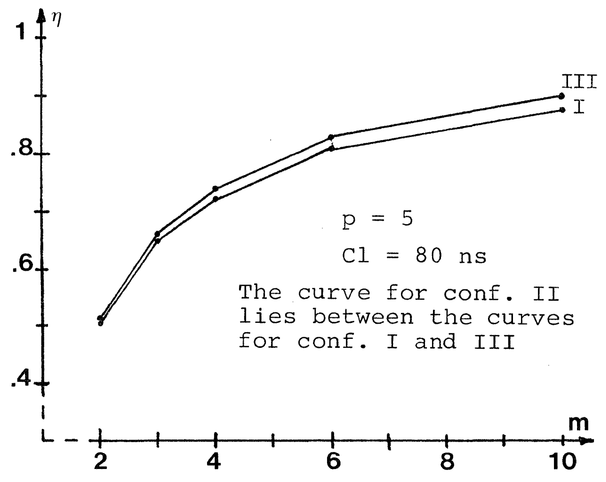


Diagram 5

Curve	p	Cl	Bus conf.
a	5	80 ns	I
b	5	200 ns	I
c	7	200 ns	II
d	6	200 ns	I
e	7	200 ns	I
f	10	200 ns	I

For Cl=80 ns, λ_0=0,10 while Cl=200 ns gives λ_0=0,19.

As was mentioned earlier, the capacity of the addressbus sets upper limits to λ and η if $p\lambda_0>1$. Also, we said that in configuration I we can never reach λ=1. This configuration with Cl=200 ns gives λ_{max}=0,81 for the cases in diagram 6. We have not yet been able to calculate λ_{max} but its value must depend on the ratio between READ- and WRITE-requests and it is probably safe to say that it has a minimum for READ/WRITE = 1.

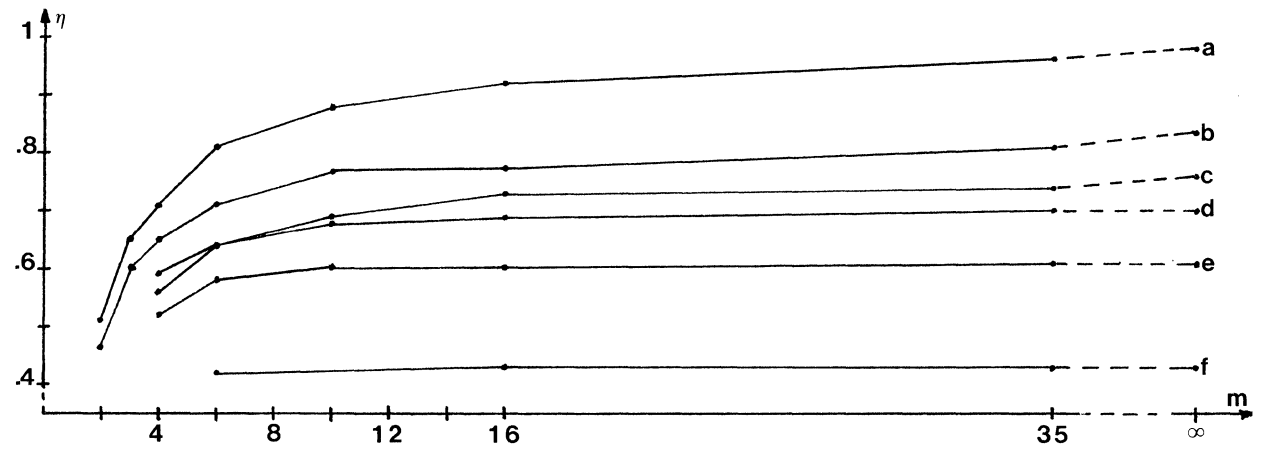


Diagram 6

With $\lambda_{max}=0,81$ and $\lambda_0=0,19$ we get

Curve	$\eta_{max}=\lambda_{max}/p\lambda_0$
b	0,85
d	0,71
e	0,61
f	0,43

These values for λ_{max} are also found in diagram 6.

As we increase m in configuration II, we will eventually reach $\lambda=1$ if $p\lambda_0>1$. For p=7 and Cl=200 ns (curve c) we get $\eta_{max}=1/p\lambda_0=0,75$, which can also be found in diagram 6.

Curve a shows a case where λ and thus C_b has been made low through a reduction of Cl. The addressbus is far from being saturated and η_{max} is close to 1. For comparison a few other configurations with p=7 have been simulated (not shown in diagram 6):

Bus conf.	Cl	m	η
I	80 ns	4	0,60
		16	0,87
III	80 ns	4	0,60
		16	0,91

Comparing these values to the values in curve c, we see that with Cl=80 ns, the reduction of C_b is such that η of configuration I is close to that ot configuration III where $C_b=0$.

For some m in a configuration with a high λ and a fixed priority scheme, C_b will be greater than C_m for the low priority processors. By adding more memory modules to the configuration we decrease C_m and increase C_b and thus the performance of the low priority processors actually goes down. Diagram 7 shows η for the individual processors in the configuration whose average η is shown by curve c in diagram 6. We observe that for m<6 C_m dominates over C_p for all processors. However, when m is further increased the performance of the lowest priority processor starts going down. The same effect for the processor with the second lowest priority is observed for m>35. In configurations where C_b is negligible, the performances of the individual processors rapidly converge as can be seen in diagram 2.

Concluding remarks

We have in this paper demonstrated that there are two main design principles in the processor-memory interface for a multiprocessor system: The multiport and the time shared single bus. Detailed hardware designs were given on some crucial points. The simulation of the dynamic behaviour reveals that the single bus contributes to little performance degradation provided the bus is reasonably fast (100 ns or less). Processor performance seems to decrease exponentially with the negative ratio between the number of processors p and the number of memory modules m. However, a fixed priority between the processors leads to low performance for the processors with low priority. This effect becomes very pronounced in the case of a slow bus. A bus configuration with only one (double directed) data bus would at first sight seem very attractive since the total number of data transfers (READs and WRITEs) never exceeds the number of address transfers. However, a READ-request from one processor creates a delayed request for the common data bus which will prohibit a WRITE (but not a READ) request in a certain later clock cycle. This effect is not fully investigated but it seems very likely that "waves" of pure WRITE-requests and READ-requests tend to develop.

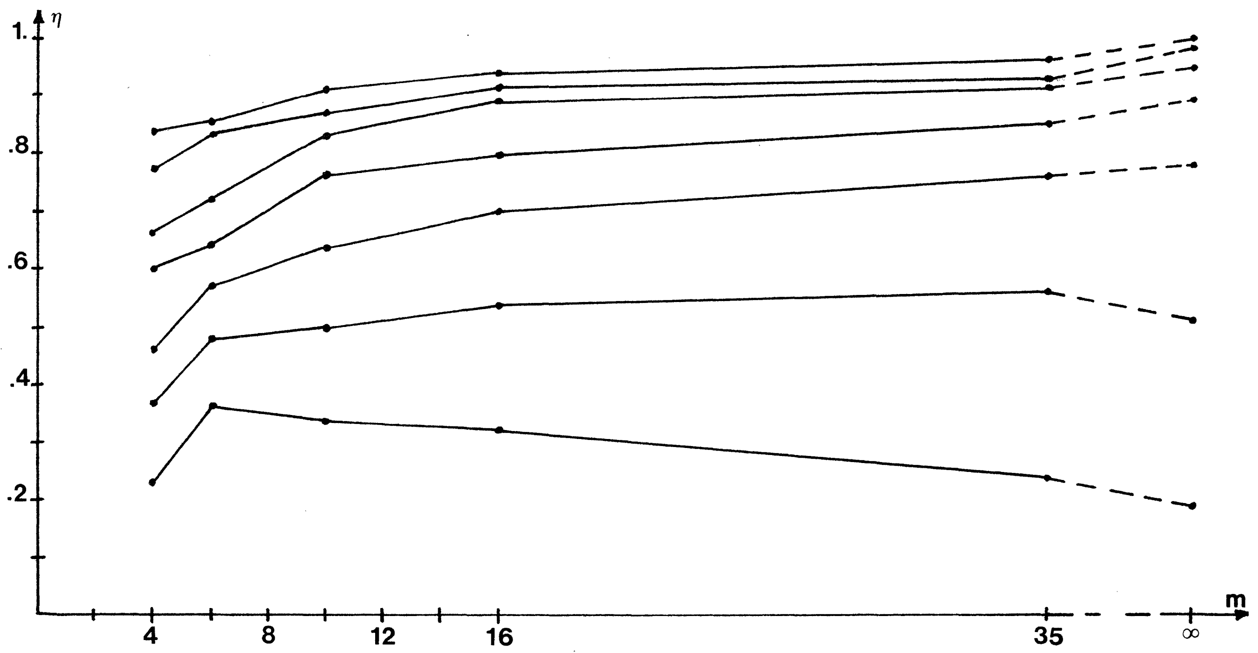


Diagram 7

References

[1] P. Enslow, Multiprocessors and Parallel Processing, Wiley, (1972).

[2] J.R. Brandsma, B.L.A. Waumans, A Common Bus Switch, Proc Second International Computing Symposium, Venice, (April, 1972), pp 446-454.

[3] S.H. Lavington, G. Thomas, D.B.G. Edwards, The MU5 Exchange, 1974 Conference on Computer Systems and Technology, IEE Conference Publication, No 121, pp 219-225.

[4] D.P. Bhandarkar, Analysis of Memory Interference in Multiprocessors, IEEE Computer Society Repository, R-74-216.

[5] A. Bodini, G. Chiabrando, R. Paul, Dynamic Access Priority Resolution in a Multi-Memory Multi-Processor Environment using a Supervisory Bus Controller, IEEE Computer Society Repository, R-74-1.

This work was supported by the Swedish Board for Technical Development

DYNAMIC TUNING IN AN ASYMMETRIC MULTIPROCESSING ENVIRONMENT

H. M. Nirsberger, S. C. Vestal
Honeywell Information Systems, Inc.
Rome, New York 13440

Summary

The environment in question concerns point to point synchronous communication between two central processors not necessarily equal in terms of hardware characteristics and/or software functionality.

Furthermore, each processor has a suitable front-end communications controller and appropriate software primitives for controlling data transmission to/from its front end. For such multiprocessing environments, it is desirable to optimize the effective speed of intercomputer data transfer. Effective line speed is affected by several factors, including number of housekeeping characters within messages, bit error rate, modem turnaround times, real line speed, and number of data characters in messages. An equation proposed by Martin [1] for relating these factors to effective line speed is given by:

$$E = D/\{T+[(D+H)/S]\}+\{[R+[(D+H)/S]][P/(1-P)]\}$$

where E is the effective line speed, D is the number of data characters in a message, T is turnaround time of the sender, H is the number of housekeeping characters in a message, S is the real line speed, R is the resynchronization time necessary for message retransmission, and P is the probability that a message is in error, which is given by:

$$P = 1 - (1 - P_b)^{B(D+H)}$$

where B is the number of bits/character and P_b is the bit error rate.

The above equations are normally applied in situations (e.g., character mode remote terminal communication) wherein T, R, S, P_b and H are constant. For such situations, the problem is one of finding the value of D for which E is maximized. The value (D+H) thus becomes the optimal block size. For multiprocessor communication, however, applications arise (e.g., transparent mode communication) wherein H is data dependent. Each permissible value of H is thus associated with its own optimal value of D. Regarding such applications, we first state and then prove a general result. The use of this result is then illustrated by a particular application.

RESULT: Suppose 2 messages M_1 and M_2 are to be transmitted. Also suppose M_1 contains H_1 housekeeping characters and D_1 data characters, where D_1 maximizes the effective line speed associated with H_1 (i.e., $E_1(D_1)$ is optimal for permissible values of D) and M_2 contains H_2 housekeeping characters and D_2 data characters, where D_2 maximizes the effective line speed associated with H_2 (i.e., $E_2(D_2)$ is optimal for permissible values of D). If $H_1 < H_2$ then $E_1(D_1) > E_2(D_2)$.

More simply, as the number of housekeeping characters in messages increases, the optimal effective line speed decreases.

PROOF: Suppose the contrary, i.e., $E_1(D_1) \leq E_2(D_2)$. Using the fact that $H_1 < H_2$, it is straightforward to show that $E_2(D_2) < E_1(D_2)$ which implies that $E_1(D_1) < E_1(D_2)$. This however, contradicts the assumption that D_1 maximizes the effective line speed for H_1, and the result is proven.

To illustrate the use of this result, consider transparent mode synchronous transmission using common ASCII control characters (e.g., DLE, STX, ETX, etc.) which are distinguished from pseudo characters by a prefixed DLE (ASCII data link escape), say. In this situation, if a DLE itself is to be transmitted as data, it too must be preceded by a housekeeping DLE. Thus if a large file is to be transmitted in transparent mode, the number of housekeeping characters in messages will depend on the data itself. At least two steps are required if the above result is to be used as the foundation of an algorithm. First, at interface initialization time, it is necessary to build, using the above equation, an array D(n) where D(i) is the optimal number of data characters for i housekeeping characters. Secondly, before transmission, the data of the file must be scanned until a character position is reached, in front of which enough data characters are found to optimize the required number of housekeeping characters. This position terminates the message, at which point the scan begins again. It is clear that other minor considerations may also be incorporated within this general scheme.

References

[1] James Martin, Systems Analysis For Data Transmission, Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1972.

PARALLELISM IN AUTOMATIC TESTING

Mathew N. Matelan
Digital Design Automation Project, Electronics Engineering Department
Lawrence Livermore Laboratory
Livermore, California 94550

Abstract -- The architecture of the DELTA (Distributed Electronic Test and Analysis) System, and its impact on test equipment control and software, is the topic of this paper. The system is based on distributing the major functions common to all testing among a group of function-specific microprocessors. These microprocessors, driven by test-mode selectable control programs, are linked in ways which allow run-time configuration of the tester on a by-test basis. The use of microprocessors to implement a high speed interpretive version of a high level test language (e.g., ATLAS) is described. The use of a firmware generator (MPACT) to develop function-defining control code is outlined.

Introduction

The increasing speed and complexity of electronic components is causing the time needed to test devices containing them to grow. Serial test of components has been the rule. Previously, when test times became exorbitant, either more testers were used, tests were degraded, or faster tester hardware was employed, but seriality was retained. The use of multiple testers is expensive and inefficient, while degrading tests is dangerous. Requiring an increase in tester circuit speed to keep pace with new UUT (Unit Under Test) circuit speeds will slow the increase in test times, but it will not halt it since UUT complexity is also increasing. Computer designers, faced with similar needs to increase execution speeds beyond those possible with existing circuit technologies, turned to architectural innovations to gain improved processing rates. Use of multiple processors, one of the innovations used by computer architects, to perform what were previously serial test functions in parallel seems a way to reduce test times. The DELTA approach is to use this method to perform test sequences at higher rates than would be possible using serial techniques.

A Multiprocessor Approach

The computational power and low cost of currently available microprocessors allows procedures and techniques developed for large computers to be applied to automatic testing. Specifically, the work done in multiprocessing and parallelism suggests that the use of multiple CPU's (central processing unit) may increase the testing capability of otherwise conventional test systems with only a change to the controller.

The general approach taken in the DELTA System is the division of a test into functions such as: switching, stimulus application, measurement data acquisition, comparison of test results with limits (Control), and user interaction. Each function is accomplished by a dedicated microprocessor, working in parallel with the other functions as circumstances permit. Each microprocessor performs a series of tasks in each test consistent with its function. The manner in which a task is performed is dictated by the current mode of the function. The mode of all microprocessors is the same during any particular test and is determined by the value of the "mode switch."

For example, the switching function microprocessor might be performing a task such as opening a relay, while the control function is (simultaneously) converting latched measurement data for limit comparisons. Both functions must be performed by methods consistent with the current mode. If the mode were analog, each task would be performed according to driver algorithms most suited to analog testing. If the mode were logical (digital), drivers optimized to perform logic testing would be used. This dependence on the value of the mode switch for the method in which functions are performed makes it possible for the test devices being controlled to be optimally configured, on a test-by-test basis, for a particular type of measurement.

Such a system allows devices which contain both analog and digital characteristics to be on the same station without degrading the quality of some tests because the tester is basically oriented toward one mode or the other. For instance, a digital module with integrated power supply would probably be assigned to a test station oriented toward logic testing. It is probable that such a tester would be less than ideal for testing power supply circuits than would be a low frequency analog oriented tester. With the DELTA approach, the digital tests would be made in digital mode, but the analog tests would be conducted after a forced mode change to analog.

The concept of mode need not be restricted to pure test functions. The inclusion of specialized modes of operation would enhance the capabilities of distributed system controller. Modes for facilitating manual fault isolation, and for collecting, editing and storing utilization statistics are two possible candidates. The dedication of a function to active circuit simulation, producing patterns for logic fault match is another distinctly attractive possibility.

Modes for testing at degraded levels of performance would increase tester maintainability. Suppose a microprocessor failed. Upon realization of the failure, the mode is switched (either automatically or manually) so that the failed unit is bypassed, and the work load distributed over the remaining processors. Testing could continue, at a slower pace, until replacements are obtained. This scheme is easily expanded to include modes to bypass two or more failed processors (or their associated circuitry). Further, since the only difference between the various functions is defined by the microprocessor's control programs, commonality in controller circuit packaging is possible allowing a single replacement subsystem.

System Architecture

A generalized DELTA controller is seen in Figure I. The diagram is divided into two sections: that of actuation, consisting of test devices interface; and control, the source of test directives. The makeup of the actuation section is irrelevant for the purposes of this paper, it being a sink for test commands and a source of test results. The control section and its interface are of primary interest here.

The system is designed around two main data busses: the world buss and the control buss. The world buss is the testing environment communications link; all data (whether stimulus commands or measurement results) are placed on the world buss accompanied by an alert code. This code signals the processor or device on the receiving end to accept data from the buss into its latch. All events are asynchronous; the scheme implementing the actuation section of the tester (i.e., the switching and device technologies used) is unaffected by the distributed architecture, modularizing the two sections. The control buss, using the same flagging (alert code) conventions, passes status and intra-control-section information between the several processors and the memories.

As discussed previously, each function is assigned to a microprocessor. Each microprocessor, at any particular time, is executing a control program suited to its function from its local control store ROM (read only memory). The particular set of control programs that the microprocessors are executing (indicated by a row of boxes, one for each microprocessor) is pointed to by the mode switch (indicated by the dashed line). Each control program consists of a local executive and a set of task-related subroutines. The executive is a looping program which scans the decode area waiting for a task code to be activated. Upon activation, control is transferred to the proper task routine where configuration codes are determined and sent to the proper device command latch for actuation.

This method allows the inclusion of a special task code to switch modes (and therefore control programs) on receipt of synchronizing signals from the other microprocessors. This is the way the test controller is switched under program control from the appearance of a dedicated analog device, to a logic-oriented one, for instance.

The need for system synchronization causes the assignment of the overall control function to that microprocessor used to check measurements against limits.

Dispatching Functions

The rationale for the DELTA approach is its capacity for taking advantage of the local parallelism that may be present in the structure of a particular test sequence. Even though a test is completely described by the commands in the decode area, the test controller must synchronize events so that execution of various parts of a series of tests may be performed in parallel.

The need for run-time detection of test subfunctions which may be performed in parallel stems from the imprecise timing caused by using generic test procedures or UUT's whose tolerances and timing differ. This makes run-time parallel activities difficult to predict, so actual initiation of potential parallel functions must be performed based on conditions as they exist during each test.

A priori function selection corresponds to global detection of parallelism as it applies to testing. Function selection is initially based on simulations of testers running benchmark ATLAS programs. Later assignments may be based on tester performance determined by individual microprocessor activity histories. The five functions discussed in this paper are derived from estimates of time spent in various activities during a general test and the probabilities of each function being performed in parallel with the others.

It is the duty of the Control function to recognize situations in test sequences which will allow functions to execute in parallel. For example, consider a single test divided into milestones consistent with the function assignments previously described:

I	Has the stimulus path been routed?
II	Has the stimulus been applied?
III	Has the measurement path been routed?
IV	Has the measurement been made?
V	Has the measurement been evaluated?

Depending on the relationships of these milestones, and their relationships to other tests in a test sequence, several tests would usually be in various stages of completion. The method used in the control microprocessor for determining which tests have active stages is a variation of the "scoreboard" (e.g., [8]). In this application, the scoreboard is a matrix in which the rows represent tests and the columns represent stages that have been reached (see Figure II).

It is assumed in this structure that on a "go" result of step V, the next test to be executed is the next set of milestones (row) in the scoreboard. Since any set-up of future tests which may not be needed (due to a no-go branch to fault isolation, for instance) would be done in parallel with the last test, no time would be wasted. The scoreboard is simply reset and testint resumed in the parallel mode at the branch target test number.

Uses of the scoreboard approach in testing, other than preliminary setup, are many. The scoreboard could be used as an indicator (pointing to new test numbers) for buffering data into pages of station memory for subsequent tests. Test numbers followed by branch instructions (as well as other special types) might also be profitably included in a special scoreboard column to indicate out-of-memory pages that would be needed if the branch were taken (a simple implementation of branch path pre-fetch).

Another area where test delays are open to reduction is the UUT adapter. Differences in UUT voltages and data representations frequently require several programs to be written which have only parametric differences. The assignment of a microprocessor to the test IO port (or "adapter box") supplied with active circuit components, allows preprocessing of raw data. The adapter box control microprocessor is shipped (by the I/O function) a control code block specific to the current UUT which configures the adapter box at run time. The control memory for the adpater control microprocessor must therefore be a RAM. Computing capability in the adapter would also allow adapter box/UUT verification (using identification resistors), and built-in adpater self-test (both initially and during testing).

Using the System

In the discussion of function control programs above, reference was made to the "decode area" of the RAM (random access memory). One might wonder why the control programs read and respond to codes found in a decode area rather than directly from the pages of memory containing the test program (e.g., PAGE 1). This is done to allow an improvement in user interface with the test system: an interpretive high-level test-oriented language (e.g., ATLAS) that executes at compiled object code speeds.

Test programs for large systems have usually been generated by two methods:

1) Compile a high level language (that is easy for engineers to use) off-line from the tester - a system that produces good test run-time code at the expense of poor development update turnaround; or

2) Use an on-station interpreter - this is commonly a lower-level language such as BASIC (ill suited to test description) which is easy to change on-line but executes slowly.

An on-station interpretive ATLAS which executes quickly would combine the best features of the two methods outlined. An interpretive language is easy to change. It is also easier for engineers to learn and use. Updating should be done through a symoblic-file management system with configuration control safeguards built in to ensure system integrity and provide automatic documentation. Once a test program is complete and accepted, the same code is used in the field; it will, however, execute at speeds usually associated with object code. The inclusion of a supervisory mode for program alteration would keep unauthorized field changes from being made. In short, the test system would appear to all users (design engineers as well as field personnel) as if it were executing ATLAS directly as its primary code.

In order to accomplish this goal, the power of the microprocessor is again exploited. The ATLAS test program, in symbolic form, is loaded as needed for execution into available pages of the RAM by the control microprocessor. This is done by using the scoreboard to determine when parts of the test program not in the RAM will be needed.

Since the next few tests in the program will always be available in the RAM, a new task is added to the control microprocessor. (Actually, several cooperating microprocessors may be required to implement this expanded control function.) This task is the interpretation of ATLAS statements which will logically follow the one currently executing. The interpreter task must determine, through added scoreboard columns, which test number to interpret. It must then interpret each ATLAS statement in the test number sequence, and produce task commands, which are stored in the decode area for the particular function for future execution. In this way, the symbolic ATLAS program is the only code ever seen outside the control function and its decode area, while the function microprocessors are executing commands at a relatively high rate since their decode areas are being filled in advance. System performance could degrade to interpreter-like speeds if several consecutive branches were performed, however, the "branch to fault-isolation on no-go" rule of programming rests would reduce the occurence of this problem.

Interpreting a large language such as ATLAS on-station, using microprocessors, is not a trivial exercise in software development. The DELTA approach is to make use of a table driven interpreter. The target language is then divided into manageable subsets, each with a table defining subset statements and corresponding function codes needed in the decode areas. These tables are supplied through the I/O (input/output) function from the test data base at the command of the control function. The tables are stored in a protected page of the RAM and are used by the interpreter task much as decode area commands are used by the various function microprocessors.

Research Directions

The realization of a DELTA based system may be divided into hardware and firmware development tasks. The redundancy of controller modules and the consistency of device/controller and interfunction communications makes the hardware task relatively simple. By far, the greatest developmental effort is seen in the firmware/software area. A system for easing the production of control programs called MPACT is under development, and its use in generating control firmware for DELTA is anticipated [4].

MPACT is intended to produce a complete microprocessor control-firmware program according to definitions of the microprocessor's capabilities, control environment, and related behavioral characteristics. Each microprocessor function in a multiprocessor configuration may be defined by a separate MPACT description. Each of these descriptions is partitioned into a system-oriented part and a function specific part. The system part, common to all processors, defines communications conventions and synchronization rules. The function-specific part defines the function of the microprocesor in terms of timing-independent, condition/response pairs. This scheme guarantees consistent function interactions while offering a convenient method for generating rapidly changing function requirements which ususally accompany system development.

Conclusion

The rapidly expanding complexity of electronic systems must eventually cause a demand for faster and faster testing devices. As has been found in the design of large, very fast computers, the physics of electronic circuitry places a bound on performance. One of the primary methods used to circumvent this problem has been the introduction of parallelism. Given the large amount of research already done in the area, and the power and decreasing cost of microprocessors, it is time to apply parallel techniques to the design of automatic test systems.

Acknowledgment

This work was performed under the auspices of the U. S. Energy Research and Development Administration.

References

[1] R. M. Holt and M. R. Lemas, "Current Microcomputer Architecture," Computer Design (February 1974), pp. 65-73.

[2] Harold Lorin, Parallelism in Hardware and Software, Prentice-Hall, Inc. (1972), p. 508.

[3] J. J. Horning and B. Randell, "Process Structuring," ACM Computing Surveys, Vol. 5, No. 1 (March 1973), pp. 5-30.

[4] M. N. Matelan, "MPACT -- Microprocessor Application to Control-Firmware Translator," ACM Special Interest Group on Design Automation Newsletter, Vol. 5, No. 1 (March 1975) pp. 13-41; also available as technical report UCRL-76990 from the Lawrence Livermore Laboratory, University of California.

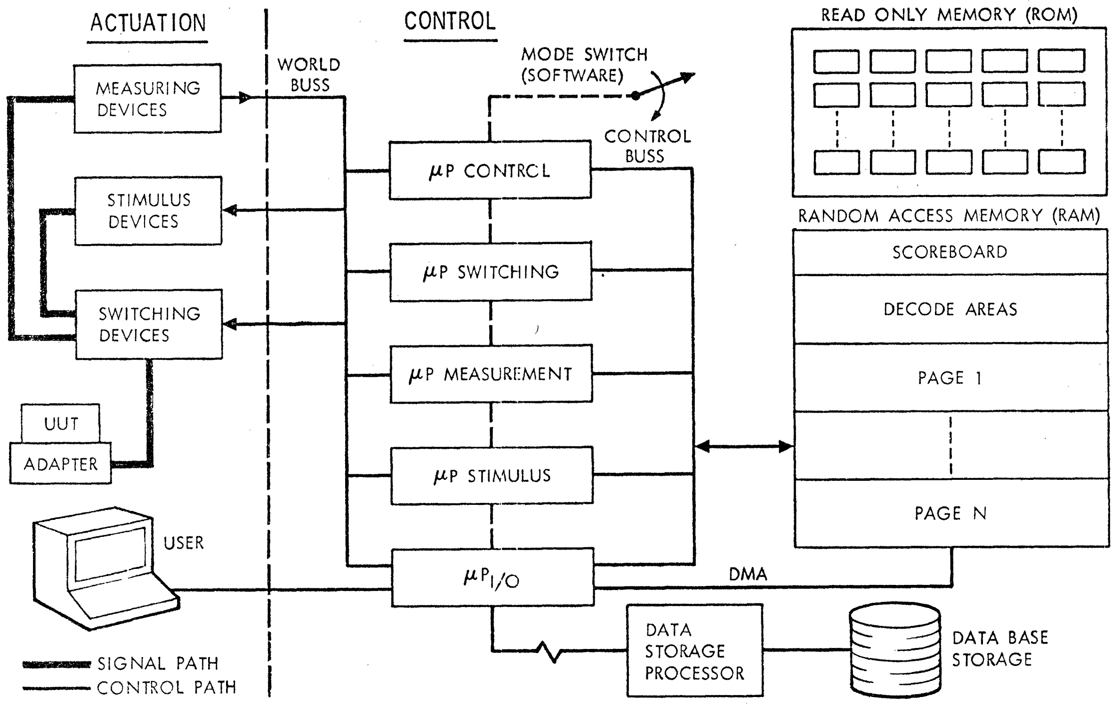
[5] M. N. Matelan, "DELTA - The Uses of Microprocessors in the Distributed Control of Electronic Testing," Joint Conference on 'Advances in Automatic Testing Technology,' University of Birmingham, Birmingham, England (1975), pp. 19-26.

[6] G. Reyling, Jr., "Performance and Control of Multiple Microprocessor Systems," Computer Design (March 1974), pp. 81-86.

[7] H. Smith, "Impact of Microcomputers on the Designer," WESCON Proceedings, Session 11/1 (1973), pp. 1-4.

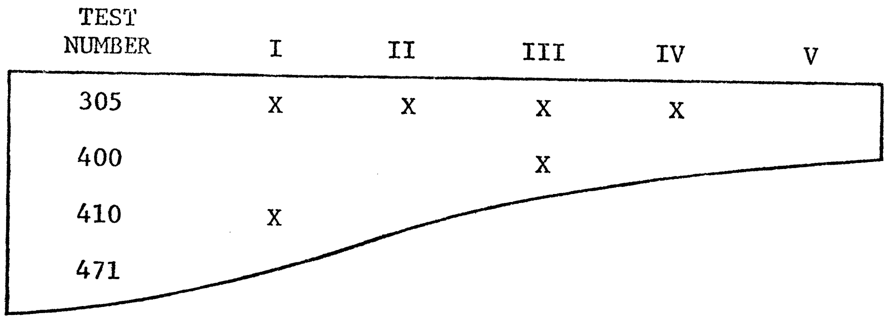
[8] J. E. Thornton, "Parallel Operation in the Control Data 6600," SJCC Proceedings (1964).

[9] Anon., "Abbreviated Test Language for Avionics Systems (ATLAS)," ARINC Specification 416-6, Aeronautical Radio, Inc., Annapolis, Maryland (1972+).



A GENERALIZED DELTA SYSTEM

FIGURE I



TEST NUMBER	I	II	III	IV	V
305	X	X	X	X	
400			X		
410	X				
471					

THE SCOREBOARD

FIGURE II

MULTI-MICROPROCESSOR SYSTEM FOR INDUSTRIAL CONTROL

Arthur C. M. Chen and William D. Barber
Corporate Research and Development
General Electric Company
Schenectady, New York 12301

Summary

The advent of low cost LSI microprocessors has made the promise of multiprocessors for industrial control extremely attractive. By replacing the numerous hardwired electronic functional modules with few standard microprocessor modules, greater economy can be achieved because of large unit production and greater product standardization. The control function would be tailored by software. Larger control requirements can be satisfied by having multiple modules in a multiprocessor system working together in a federated mode.

A possible multi-microprocessor configuration is shown in Fig. 1. Each microprocessor based module would have its own private memory and, if needed, device interfaces. It could operate in a semiautonomous mode performing various control functions such as sequencing, regulation or data logging as defined by the system generation process. The system as a whole would communicate through a common memory and be supervised by a designated Master Module.

At system generation time, the tasks necessary for the various control functions are assigned to the modules. These tasks would run under the control of a small real time operating system resident in each microprocessor module. The entire multiprocessor system would be supervised by the Master Module whose tasks are the coordination of intermodule communication, the assignment of common memory access priority and the control of the external peripheral devices. The system can be made fault-tolerant by providing spare modules which are activated by the Master Module to replace failed module.

The philosophy of decreasing system cost by replacing hardware modules by software modules "sounds good" - but it is fraught with pitfalls, especially in the area of programming cost. Thus, although the system shown in Fig. 1 represents no new concept, it does represent a real challenge to the development of a system design, generation and requisition discipline to minimize the problem of software modularity and programmer efficiency. Central to this challenge is the cost and performance constraints of industrial control and the particular characteristics of LSI microprocessor.

LSI microprocessors are characterized by its small word length, limited instruction repertoire and addressing modes. In addition, their control structures are limited to single processor mode.

The lack of master processor mode makes the implementation of secure real time operating systems by software alone difficult. To provide some protection from user program and hardware errors the operating system and other crucial procedures should be implemented in Read Only Memory to provide quick restart capability in the event of a crash.

Industrial control systems are characterized by short "process" cycles - 30 to 500 ms. Thus the operating system must be designed to respond quickly to external and internal events with a limited speed CPU. To have the ability of operating in a small stand-alone module or in a federated system, the operating system should have dual process modes; time base for stand-alone and supervisory base for federated system modes.

Finally, to minimize programming cost, the software development and requisition process must be done in a "disciplined" interactive environment. Thus the support of a large computer with extensive file editing and storage capabilities as well as a flexible cross-assembler and linkage loader for the target microprocessor is needed. A facility should be created on this computer which will enable engineers to link pre-assembled modules together with the unique system application package to fulfill various requisition requirements. Central to this facility is the cross-assembler or cross-compiler which will generate relocatable object code modules which can be binded together by the linkage loader into specific memory locations. The output of the facility in the form of an absolute formatted code can be easily loaded into the microprocessor module for test and acceptance.

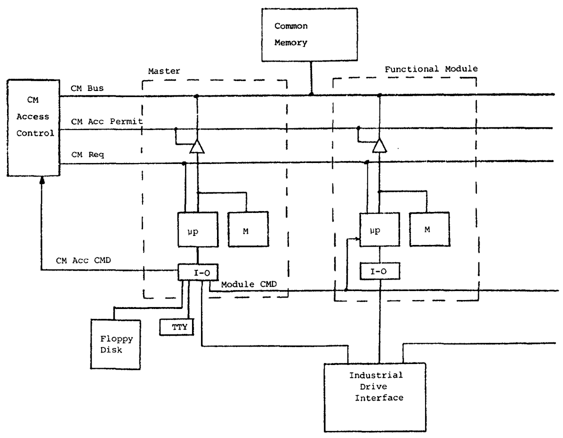


Fig. 1 Multi-Microprocessor System for Industrial Control

ANALYSIS TECHNIQUES FOR SIMD MACHINE INTERCONNECTION NETWORKS AND THE EFFECTS OF PROCESSOR ADDRESS MASKS

Howard Jay Siegel
Electrical Engineering Department
Princeton University, Princeton, New Jersey 08540

Abstract — We consider interconnection networks as permutations on the set of processor addresses. The relation with permutation groups is exploited to determine if a given network can simulate any arbitrary one. The effects of processor address masks, that determine which processors will be active, are examined. We also present model independent techniques for proving lower bounds on the time required for one network to simulate another.

Introduction

One problem in the design of SIMD (single instruction stream - multiple data stream [5]) or array machines is the construction of an interconnection network to pass data from one processor to another. The model of a SIMD machine used in this paper consists of $N = 2^m$ processing elements. Each processing element (PE) is a processor together with its own memory and is assigned an address from 0 to N-1. In this model, we consider no processor instructions other than those that transfer data to another processor.

An interconnection network is a set of interconnection functions, each a total function on the set of PE addresses. By applying sequences of functions, networks can transfer data between PE's. When a function f is applied, processor i passes its data to processor f(i) for all i, $0 \leq i < N$.

An m-position PE address mask may accompany any data transfer instruction and will determine which PE's are active; i.e., send data. When a mask is used, the only PE's that are active are those whose address matches the mask in the following way for each bit position: if the mask has a 0, then the PE address must have a 0; if the mask has a 1, then the PE address must have a 1; and if the mask has an X, then the PE address may have either a 0 or a 1.

Interconnection Networks

The following interconnection networks will be discussed. In the definitions of the interconnection functions and throughout the rest of the paper let $N=2^m$, let the binary representation of a PE address be $p_{m-1}p_{m-2} \cdots p_1p_0$, let $\bar{p}_i$ be the complement of p_i, and let the integer n be the square root of N.

(1) The Cube. This network consists of m functions defined by:

$$c_i(p_{m-1}\cdots p_{i+1}p_ip_{i-1}\cdots p_0) = p_{m-1}\cdots p_{i+1}\bar{p}_ip_{i-1}\cdots p_0$$

for $0 \leq i < m$; e.g., $c_2(7) = 3$, for $N \geq 8$. When the PE addresses are considered as the corners of an m-dimensional cube this network connects each PE to its m neighbors (see [13]).

(2) The Perfect Shuffle (PS). This network consists of a shuffle function and an exchange function. The shuffle is defined by:

$$s(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-2}p_{m-3}\cdots p_1p_0p_{m-1}$$

and the exchange is defined by:

$$e(p_{m-1}p_{m-2}\cdots p_1p_0) = p_{m-1}p_{m-2}\cdots p_1\bar{p}_0,$$

e.g., $s(3) = 6$ and $e(6) = 7$, for $N \geq 8$. The shuffle can be thought of as the result of perfectly shuffling (intermixing) a deck of cards (i.e., $0 \rightarrow 0$, $N/2 \rightarrow 1$, $1 \rightarrow 2$, $N/2+1 \rightarrow 3$, etc.) (see [6], [8], [15]). Note that $e = c_0$.

(3) Plus-Minus 2^i (PM2I). This network consists of 2m functions defined by:

$$t_{+i}(j) = j+2^i \bmod N,$$
$$t_{-i}(j) = j-2^i \bmod N,$$

$0 \leq i < m$; e.g., $t_{+1}(2) = 4$, for $N > 4$. (See [2], [4].)

(4) Illiac IV. This network has four functions of the form:

$$I_j(x) = x+j \bmod N, \text{ where } j = +n, -n, +1, -1;$$

e.g., $I_{+n}(0) = 4$, if N=16. When we discuss the Illiac IV we shall assume m is even, that is, $n = 2^{m/2}$ is an integer. If the PE's are considered as a n x n array, then each PE will be connected to its north, south, east, and west neighbors (see [1], [3], [11], [14]). This network is a subset of PM2I.

(5) Wrap-around Plus-Minus 2^i (WPM2I). This network consists of 2m functions defined by:

$$w_{+i}(p_{m-1}\cdots p_i\cdots p_0) = q_{m-1}\cdots q_i\cdots q_0,$$

where $q_{i-1}\cdots q_0q_{m-1}\cdots q_{i+1}q_i = (p_{i-1}\cdots p_0p_{m-1}\cdots p_{i+1}p_i) + 1 \bmod N$,

and: $w_{-i}(p_{m-1}\cdots p_i\cdots p_0) = q_{m-1}\cdots q_i\cdots q_0$,

where $q_{i-1}\cdots q_0q_{m-1}\cdots q_{i+1}q_i = (p_{i-1}\cdots p_0p_{m-1}\cdots p_{i+1}p_i) - 1 \bmod N$,

for $0 \leq i < m$. WPM2I is like PM2I, except any

This work was supported by NSF Grant DCR74-21939. It is a summary of Princeton University, Dept. of E.E., Computer Science Lab. TR-185.

"carry" or "borrow" will "wrap-around" to the p_{i-1} bit position: e.g., if N=8 and m=3, then

$$w_{-1}(001) = 110,$$

whereas $t_{-1}(001) = 111$.

Interconnection Networks as Permutations

The set of all bijections on the N PE addresses is the group of permutations on N elements, called S_N. A network is universal if some sequence of the functions in that network can, possibly with the use of PE address masks, generate S_N: i.e., simulate any interconnection function that is a bijection.

Every permutation can be uniquely represented as the product of disjoint cycles. The notation used to represent a cycle of the bijection f is:

$$(i_0\ i_1\ i_2\ \dots i_x)$$

where $f(i_0) = i_1$, $f(i_1) = i_2$, ..., $f(i_x) = i_0$. Cycles of length 1 (i.e., f(i) = i) are removed. A permutation in S_N is said to be an even permutation if it can be represented as the product of an even number of transpositions (cycles of size 2). Any representation of an even permutation as the product of transpositions uses an even number of transpositions. The product of two even permutations is an even permutation. S_N also contains odd (non-even) permutations. (See [7] or [12].)

Theorem 1: There does not exist a single interconnection function which is universal for $N \geq 3$.

Proof: If a single permutation A generates S_N, then it cannot have more than one cycle. If a mask other than XX...X is used, the resulting function would not be a bijection. If A with mask XX...X could generate S_N, then S_N would be a cyclic group, but for $N \geq 3$ it is not.

Theorem 2: Let F be the set of all distinct bijections obtained by applying each function of a particular network with every possible PE address mask. If the network is universal then a lower bound on the time required to simulate an arbitrary interconnection function is $\log_{|F|}((N!)(|F|-1)+1)-1$.

Proof: The elements of F must form N! distinct sequences. The length of the longest sequence must be at least x, where:

$$\sum_{i=0}^{x} |F|^i \geq N! \ .$$

Therefore, $x \geq \log_{|F|}((N!)(|F|-1)+1)-1$.

Theorem 3: The following table shows which networks are universal with and without PE address masks.

Network	With	Without
Cube	yes	no
PS	yes	no
PM2I	no	no
Illiac	no	no
WPM2I	yes	yes

Proof:

Cube:

With masks: The set of permutations (0 1), (0 2), ..., (0 N-1) generate S_N (see [7], page 69). Let j be a number from 0 to N-1 that has a 0 in the ith bit position. Then c_i with a mask equal to the bit representation of j except for an X in the ith mask position is equivalent to the cycle (j $j+2^i$) for $0 \leq i < m$. We use the following algorithm to construct a sequence of transpositions of the form (j $j+2^i$), the product of which will be (0 K), for fixed K, $1 \leq K < N$. Let the binary representation of K be $k_{m-1}\ k_{m-2} \dots k_1\ k_0$. Let (a b) represent a variable that is a cycle of size 2. Initially let (a b) = (0 0), which is just "do nothing."

```
for i = 0 to m-1 do
    if $k_i = 1$
        then let ( a b ) be the product
        of ( a b ) ( b $b+2^i$ ) ( a b )
end
```

For example, if K=6, (0 6) would be [(00) (02) (00)] (26) [(00) (02) (00)] .

Without masks: For $0 \leq i < m$, c_i is an even permutation.

PS:

With masks: Let R' equal the mask R cyclically left shifted m-i times. Let s^i represent i shuffles. Then the sequence s^{m-i} with mask XX...X, e with mask R' and s^i with mask XX...X, is equivalent to c_i with mask R. Thus, PS can simulate the Cube with masks, which is universal.

Without masks: (From [9] .) Each bijection performable by sequences of PS functions can be represented as $x_{m-1}\ x_{m-2} \dots x_1 x_0$, where the x_i's are literal symbols of the form j or $\bar{j}$, $j \in \{0,1,\dots,m-1\}$, such that if x_i = j or $\bar{j}$, then x_{i+1} = j+1 or $\overline{j+1}$ (mod m). The number of such bijections is $m(2^m)$. But $m(2^m) < N!$, for $m \geq 2$.

PM2I:

With or without masks: We can show that the

permutation $(0\ 1\ 2^{m-1}\ 2)$ cannot be generated, by using induction on the number of data transfers to prove that whenever 0 maps to j mod N, 2^{m-1} maps to $2^{m-1}+j$ mod N.

Illiac:
With or without masks: Follows from PM2I.

WPM2I:
With or without masks: $w_{+0} = (0\ 1\ 2\ \ldots\ N-1)$ and $(w_{+0})^2\ (w_{-1}) = (N-2\ N-1)$ generate S_N (see [7], page 69).

Bijections Obtainable Using Masks

When designing an SIMD machine a set of interconnection functions and a set of masking schemes must be chosen. The way in which the functions and masks interact is an important consideration. The next theorem analyzes this in terms of our model.

Theorem 4: The number of distinct bijections obtained by applying each function of a network with every possible PE address mask is given in the following table, along with the number of distinct functions in that network:

Network	# bijections	# functions
Cube	$m(3^{m-1})$	m
PS	$1 + 3^{m-1}$	2
PM2I	$2(3^{m-1}) - 1$	2m-1
Illiac	$2 + 2(3^{m/2})$	4
WPM2I	2m	2m

Proof:

Cube: For each c_i the mask must contain an X in the ith position and the other m-1 positions can be either 0, 1, or X.

Perfect Shuffle:
Exchange: Follows from Cube
Shuffle: The shuffle function, s, is not a bijection when used with any PE address mask other than XX...X, 00...0, or 11...1. In proof consider the following.
Case 1: There are no X's in the mask. Let the mask be $R = r_{m-1}r_{m-2}\ldots r_1r_0$, where $r_{i+1} \neq r_i$. Then $s(r_{m-1}r_{m-2}\ldots r_1r_0) = s(r_{m-2}r_{m-3}\ldots r_0r_{m-1})$.
Case 2: There is at least one X in mask R. Without loss of generality, let $r_{i+1}=0$ and $r_i=X$. Let p be such that it has a 1 in its ith bit and matches mask R. Then $s(p)=p'=s(p')$.

PM2I: The ith to m-1st positions of the PE address mask must be X's if used with t_{+i}. Thus, by setting the 0th to i-1st positions of the mask 3^i distinct bijections can be obtained. Note that $t_{+i} \neq t_{-i}$, except for i=m-1.

Illiac IV: Follows from PM2I.

WPM2I: w_{+i}, $0 \leq i < m$, is a single cycle of size N and therefore is not a bijection when used with a mask other than XX...X.

Theorem 5: Let D(f) be the number of cycles in the unique disjoint cycle representation of the interconnection function f. No matter what type of masking system is used the maximum number of distinct bijections obtainable by applying f with different masks is $2^{D(f)}$.

Proof: The number of bijections obtainable is equal to the size of the power set of the set of disjoint cycles of f.

Lower Time Bounds On Simulations

All of the results and techniques in this section are valid for all models of SIMD machines. In models in which PE's can save their data in their respective memories and later reload it, the transfers specified by interconnection functions with masks need not be bijections.

The way in which the actions of a network on one PE's data affect the other PE's data is a function of the model of SIMD machines being used, i.e., type of masking, allowable instruction set, etc. Therefore, in order to maintain the model independence of our results, the lower bounds we prove are based on the actions of two different networks on a single PE address.

We shall define a metric d to have the following properties: $d(x,z) \leq d(x,y) + d(y,z)$, (the triangle inequality); $d(x,y) \geq 0$; and $d(x,x) = 0$. If f is an interconnection function and d is a metric, then $d(x,f(x))$ is the "distance" that f can "move" PE address x. Let d_f be $\max_x d(x,f(x))$. Then, if $d_f \geq d_g$, there exists a PE address for which it must take at least d_f/d_g time for function g to simulate f.

Theorem 6: In the following table the entry in row x, column y, is a lower time bound for network x to simulate network y. A "c" indicates that with certain SIMD machine models the simulation can occur in constant time.

	Cube	PS	Illiac	PM2I	WPM2I
Cube	-	$2\lfloor m/2\rfloor$	m	m	m
PS	m	-	2m-1	2m-1	2m-1
Illiac	n/2	(n/2)+1	-	n/2	(n/2)+1
PM2I	c	$2\lfloor m/4\rfloor$	c	-	c
WPM2I	c	$2\lfloor m/4\rfloor$	c	c	-

Proof:

Cube → PS: Let d be the Hamming distance, i.e., let d(x,y) = the number of bit positions in which x and y differ. $d(x,c_i(x)) = 1$, $0 \leq i < m$, so $d_{Cube} = 1$. Let x = 0101...01, if m is even, and 0101...010 if m is odd. Then

$d(x,s(x)) = 2\lfloor m/2 \rfloor = d_{PS}$.

<u>Cube→ Illiac</u>: Let d be as above. Since $I_{+1}(11...1) = 00...0$, $d_{Illiac} = m$.

<u>Cube→ PM2I</u>: Follows from above.

<u>Cube→ WPM2I</u>: Let d be as above. Since $w_{+0}(11...1) = 00...0$, $d_{WPM2I} = m$

<u>PS→ Cube</u>: $c_{m-1}(1^m) = 01^{m-1}$. The shortest sequence for the PS to map 1^m to 01^{m-1} is es^{m-1}.

<u>PS→ Illiac</u>: $I_{+1}(11...1) = 00...0$. The shortest sequence for the PS to map $11...1$ to $00...0$ is $(es)^{m-1}e$.

<u>PS→ PM2I</u>: Follows from above.

<u>PS→ WPM2I</u>: Follows from above, since $w_{+0}(11...1) = 00...0$.

<u>Illiac→ Cube</u>: Let $d(x,y) = |x-y|$. Let $j = (m/2)-1$. Then $d(0,c_j(0)) = n/2$. I_{+n} and I_{-n} can not be used to move a distance of n/2, and $d(x,I_{\pm 1}(x)) = 1$, $0 \leq x < N$.

<u>Illiac→ PS</u>: $s(10^{m-1}) = 0^{m-1}1$. The only way to change the 1 in the m-1st bit position using less than $(n/2)+1$ steps is n/2 executions of I_{+n} or n/2 executions of I_{-n}. In both cases the 0th bit position remains unchanged.

<u>Illiac→ PM2I</u>: Let d and j be as in Illiac→ Cube. Then $d(0,t_{+j}(0)) = n/2$.

<u>Illiac→ WPM2I</u>: $w_{+(m-1)}(10^{m-1}) = 0^{m-1}1$. Illiac requires $(n/2)+1$ steps to perform this mapping (see Illiac→ PS).

<u>PM2I→ PS</u>: Let h be the Hamming distance. Let k(x) be a characteristic bit vector of address x, such that its ith bit is 1 if and only if the ith bit of x does not equal the i-1st mod m. Let $d(x,y) = h(k(x),k(y))$. Let $y = m \bmod 4$. Let $x = 00110011...0011(0^y)$. Then $h(k(x),k(s(x)) = m-y = 4\lfloor m/4 \rfloor$. By a case analysis it can be shown that $h(k(x),k(t_{\pm i}(x))) \leq 2$, for $0 \leq i < m$ and $0 \leq x < N$.

<u>WPM2I→ PS</u>: Similar to PM2I→ PS.

Conclusions

We developed several analysis techniques for evaluating interconnection networks and examined five particular networks. We described and studied the effects of a PE address masking system which would require $O(\log_2 N)$ bits, when a masking system which could specify any arbitrary set of PE's would require O(N) bits. Model independent techniques for determining a lower time bound on the simulation of one network with another were presented.

Future work in this area would include the design and analysis of other masking systems and interconnection networks. Hybrid networks, such as the PM2I with a shuffle function, present interesting possibilities. Further research would also include an examination of the added flexibility the use of store and load instructions would create by allowing mappings that are not bijections to be used.

Acknowledgements

I would like to thank Prof. J.D. Ullman for his help and guidance with this research. I would also like to thank Prof. R. M. Keller for his comments and suggestions.

References

[1] G.H. Barnes, et. al., "The ILLIAC IV computer," IEEE Trans. Comput., Vol. C-17 (Aug., 1968), pp. 746-757.

[2] K.E. Batcher, "STARAN/RADCAP hardware architecture," Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing, pp. 147-152.

[3] W.J. Bouknight, et. al., "The Illiac IV system," Proceedings of the IEEE, Vol. 60, No. 4 (Apr., 1972), pp. 369-388.

[4] T. Feng, "Data manipulating functions in parallel processors and their implementations," IEEE Trans. Comput., Vol. C-23 (Mar., 1974), pp. 309-318.

[5] M.J. Flynn, "Very high-speed computing systems," Proceedings of the IEEE, Vol. 54, No. 12 (Dec., 1966), pp. 1901-1909.

[6] S.W. Golomb, "Permutations by cutting and shuffling," SIAM Review, Vol. 3, No. 4 (Oct., 1961), pp. 293-297.

[7] I.N. Herstein, Topics in Algebra, Xerox College Publishing, (1964).

[8] P.B. Johnson, "Congruences and card shuffling," American Mathematical Monthly, Vol. 63 (Dec., 1956), pp. 718-719.

[9] R.M. Keller, private communication.

[10] D.E. Lawrie, Memory-Processor Connection Networks, Dept. of Computer Science, University of Illinois, Rep. 557, (Feb., 1973).

[11] S.E. Orcutt, Computer Organization and Algorithms for Very-High Speed Computation, Dept. of Computer Science, Stanford University, Ph.D. Thesis, (Sept., 1974).

[12] D. Passman, Permutation Groups, W. A. Benjamin, Inc., (1968).

[13] D. Rahmlow, "Parasim," Princeton University, unpublished paper, (1974).

[14] D.L. Slotnick, et. al., "The SOLOMON computer," 1962 Fall Joint Computer Conf., AFIPS Proc., Vol. 22 (1962), pp. 97-107.

[15] H.S. Stone, "Parallel processing with the perfect shuffle," IEEE Trans. Comput., Vol. C-20 (Feb., 1971), pp. 153-161.

DESIGN CRITERIA FOR A SWITCH FOR A MULTIPROCESSOR COMPUTING SYSTEM

Ian A. Davidson and James A. Field
Department of Electrical Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

Abstract. The characteristics of a switch connecting the processors and a common memory in a closely coupled multi-processor computing system are considered. The switch is assumed to be controlled from part of the address information, and the effect of signal blocking is considered. The performance of the switch is discussed in conjunction with the processor characteristics, and the two have to be matched to achieve efficient processor utilization. An example shows that for a 64 processor system, the presence of switch blocking only reduces the processor utilization by about 5%.

Introduction

In a closely coupled multiprocessor computing system where the processors share a common memory the switch interconnecting the processors and the memory modules is a critical component. This has been considered by Miller et al,[1] by Baer[2] and more recently by Chen and Frank[3]. An example of the type of system under consideration is shown in figure 1, where the switch makes a connection, between a number of processors and a multiple port memory system.

It will be assumed that the memory modules are identified by part of the data address and this information will also be used to route the path between the processors and the memory. Any switch architecture can be employed, but for simplicity of operation, the routing through the switch should be capable of being established easily from part of the address data. It is further assumed that control of the switch will be distributed, with each switch module being controlled by the address information associated with the data. There is no external control system for the switch.

Switch characteristics

The switch will be considered to be constructed with a number of identical binary modules, an example of such a module being shown diagramatically in figure 2. The modules are arranged to form a Rearrangeable Switching Network (RSN) as described by Opferman and Tsao-Wu[4]. The data path from left to right - representing the path from the processors to the memory, can contain either data or address bits. Certain of the address bits are used by each module for its control. At any time, each module can be in one of three states:

1. The module is not in use
2. The module is set in the parallel position, where the inputs A,B, C and D are connected to the outputs A', B', C' and D' respectively.
3. The module is set in the crossed position, where the inputs A,B,C, and D are connected to the outputs B', A', D' and C' respectively.

The path taken by the data through the switch is determined by the memory address issued by the processor. This when received by each module is used by that module to control the switch position and hence route the signal to the next layer of modules until a complete path between the processors and the memory is established. Data can then be sent through this path in both directions, the path being maintained while a block of data is transmitted, the data being read (or written) from successive memory locations. The size of a data block and the width of the data path are important factors in the design and performance of the switch and will be considered in detail later.

The switch configuration can be described as 'binary', and it is a compromise between a completely non-blocking switch (i.e. a single cross-bar switch) and the simplest configuration consisting of a single data bus. It also has the advantage that the routing of the data can be associated with the binary bits of the address, successive bits corresponding to the control of successive levels of the switch. The blocking characteristics of the switch are important as this determines the delay that can occur between the request for data by the processor and the time before that data is available.

Blocking can arise when a switch module is already in use by another connection, and there is a conflict in the state required in the switch. However with a non-blocking switch design there will still be blocking due to memory contention, which will have the same characteristics as switch blocking.

An analysis can be made of the performance of such a switch. For a switch with m layers, assume both 2^m input (processor ports) and output (memory) ports, and assume that an average of p processors are requesting access to the memory. Assume also that the requested addresses are random and uniformly distributed and are uncorrelated between the processors, assumptions that will be untrue in a practical system, but are necessary for a reasonable analytical result. The theory can be extended to the more general case where there is correlation between memory references. The number of the requests that are blocked by the switch is given by the recurrence relation

$$x_{i+1} = x_i - S/4$$

where $S = \dfrac{x_i(x_i - 1)}{2(2^m - 1)}$ which is the number of switches in which both input terminals are in use and where x_i is the number of paths at level i in the switch, the number of paths through the switch will be x_{m+1}. The probability of a given call being blocked by the system b(p) is:

$$b(p) = 1 - \frac{x_{m+1}}{p}$$

where x_{m+1} is the number of completed connections through the switch. Values of b(p) have been computed using the above equations, and the result is shown in figure 3.

When the data transfer between the processor and memory is complete, the path is released, starting at the memory side of the switch and going back to the processor. As soon as a switch module is released, if it is blocking another path, it is switched to enable routing of the previously blocked path. The finite blocking probabilities show up as a delay in the connect time. The time taken to transfer data is the sum of three components, i.e.:

$$T_{TOTAL} = T_{SWITCH} + T_{MEMORY} + T_{BLOCKING}$$

where T_{TOTAL} is the total time between the the request for data and its availability

T_{SWITCH} is the time taken to route the signal through the switch

T_{MEMORY} is the time to access memory in the memory block

$T_{BLOCKING}$ is the extra delay due to blocking in the switch

Processor Characteristics

In order to assess in more detail the performance of the switch, more consideration must be given to the overall system, in particular to the configuration of the processors. This will determine the optimum width of the data path and the average size of the data blocks being sent through the switch.

It has already been established that delays can occur when a processor requests information from the memory, but that once a path is established data flows continuously at a rate determined by the technology. This implies that for maximum processor utilization the tasks performed by the processors should be at a 'high' level, i.e. should contain sufficient data to enable the processing time for an individual task to be long compared to the time taken to establish a connection in the switch.

Various typical macro-processor tasks can be identified, the following are some examples:

a) Arithmetic processing sequence
b) Communication data processing (communication protocol)
c) Output data processing (Format)
d) Sorting and Hashing routines
e) Search programs
f) Matrix operations (i.e. inversion)
g) Transforms (FFT etc)
h) Analogue Simulation (differential equations etc).

The processors would therefore require local memory, both for the data and for local instructions. These memories should be of high speed and matched to the technology used in the processor.

The processor local instructions would be in three categories:

1. Instructions and tasks that are frequently used. These would be stored in Read only memory (ROM), and would be available for use at any time.

2. Instructions that have a high probability of being used, but are stored in Read write memory (RAM). These would normally be available for immediate use, but which could be overwritten if the memory space were required for specialized instructions.

3. Specialized instructions that are infrequently used, and would normally have to be loaded from the main memory at the start of each task.

There would normally be a data transfer at the start and end of each task. However the data would be available in the processor memory for the subsequent task if required.

Data format

The format for the data has to be consistent with established computing practices. The most usual data 'units' are characters, integers and real numbers, and characters are usually assigned 8 binary bits. However any data format should easily be expandable to include in one extreme binary bits, and in the other extreme double precision and complex numbers.

The data block would be preceded by a header that would include bits to indicate the nature of the data being transferred and the size of the data block. This is shown in figure 4. Furthermore the data and the header should be easily subdivided into 'words' for parallel transmission down the data path. The cost of the switch will be proportional to the width of the data path, whereas the time to transfer data will be inversely proportional to it. As a compromise three possible configurations are shown in the table in figure 5: - a data path of width 4 with a record size of 2060 bits, or data path of width 8, with record sizes of either 136 or 32784 bits. The total transfer time is shown in terms of the factor T_N, which is the time interval between successive data words in the data path. This will probably be determined by the design of the memory, but should be matched to the technology used in the switch, as well as to the rate at which the processor can accept the data.

The block size required by the processor tasks outlined above can be estimated. Preliminary investigation shows that the record size of 32784 bits (32768 data bits) would probably be optimum for the tasks outlined for the processors.

Optimization of the system

It is now possible to collect these facts together and calculate the overall performance of the system. It is considered that there are sufficient tasks available to keep all the processors active continuously, and the problem is to calculate a figure of merit that describes the utilization of the system.

A given processor will take a time T_{TASK} to complete a task, at which point it will access the main memory, and require a time T_{TOTAL} to get more data before it can continue processing. Now the quantity T_{TOTAL} contains the term $T_{BLOCKING}$, which is the time the processor has to wait while other units are accessing the memory. $T_{BLOCKING}$ can be calculated from the probability of a request being blocked b(p) and the time taken for tasks T_{TASK}

$$T_{BLOCKING} = b(p) \times T_{TASK}$$

It is also possible to define a factor R which is the proportion of time for which a processor is executing a task. R can be derived in two ways:

$$R = 1 - \frac{p}{2^m}$$

from the probability of the port being used

and
$$R = \frac{T_{TASK}}{T_{TASK} + T_{TOTAL}}$$

from the time taken to complete tasks, then

$$\frac{T_{SWITCH} + T_{MEMORY}}{T_{TASK}} = \frac{1}{2^m/p - 1} - b(p)$$

and this is a function of p. This expression can be used to calculate the ratio of the time to propagate signal through the switch together with the time to obtain data from the memory ($T_{SWITCH} + T_{MEMORY}$) to the time for the processor to complete a task. Values for this ratio have been calculated, and some of the results are shown in figure 6. In this values are given for different size switches,m 4 to 8 (2^m ranging from 16 to 256) in which the processor utilization is assumed to be 0.90 and 0.95.

Discussion of results

The results have shown that the utilization of the processors in a multiprocessor system will be reduced compared to a single processor system due to blocking in the switch, but this 'blocking' will also be present in a non-blocking switch configuration due to memory contention. It has also been shown that it is possible to increase the processor utilization to as near one as required by increasing the task time (T_{TASK}) compared to the access time $T_{SWITCH} + T_{MEMORY}$. It should be noted that T_{SWITCH} represents the time for a memory request to propagate through the switch, and as it does not include delays due to blocking, will be small. T_{MEMORY} however, is the time taken for the transfer of data in or out of memory, and can be significant if the size of the data block is large.

The results can be illustrated by a practical example. For a system with 64 processors, i.e. m = 6, if the average data block size is 2000 - 8 bit words, being transferred at a data rate of one word every 50 nanoseconds (T_N), the value of T_{MEMORY} would be 100 microseconds. If the average task were 2.3 milliseconds, the value of R would be 0.90, and if 3.7 milliseconds, R would be 0.95. In these two cases the average number of processors requesting use of the switch at any time would be 5.8 and 3.2, and the blocking probability 0.054 and 0.025 respectively. This example illustrated the reason for assigning processor tasks to as high a level as possible. A task lasting 4 milliseconds would correspond to about 5000 machine instructions, which corresponds to the complexity of the tasks outlined under processor characteristics.

Various approaches can be made to reduce the task time and still maintain a high processor utilization. The switch can be made more complex in order to reduce the blocking probability in the switch. Also the number of memory ports could be increased so as to reduce the incidence of memory contention. The above analysis is based on the use of random memory addressing. However the incidence of memory contention could be reduced if the memory allocation was ordered correctly.

Conclusion. The performance of a switch for use in a multiprocessor system has been discussed, and various properties of such a system presented. It has been shown that a high processor utilization can be achieved under realistic conditions.

No consideration has been given in this paper to the allocation of tasks to the processors, the introduction of interrupt facilities or the inclusion of input/output devices. This is being considered separately, but it is not expected to change significantly the concepts presented in this paper.

References

1. Miller, James S. et al, "Multiprocessor Computer System Study-Final Report," Intermetrice Inc., Cambridge, Mass March 1970.
2. Baer, Jean-Loup, "Large Scale Systems," Chapter 5 in Computer Science, Wiley-Interstate, New York, 1972.
3. Chen, C.J. and Frank, A.A. On the Programmable Parallel Data Routing Networks for Multiple Element Computer Architecture. Proc. 1974 Sagamore Conf. on Parallel Processing.
4. Opferman D.C. and Tsao-Wu, N.T. On a class of Rearrangeable Switching Networks, Bell System Technical Journal, No. 50, May-June 1971, pp. 1579-1618.

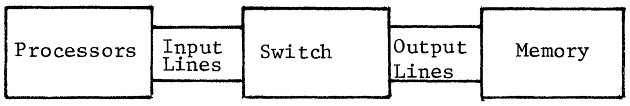


Figure 1. System configuration

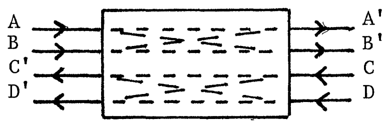
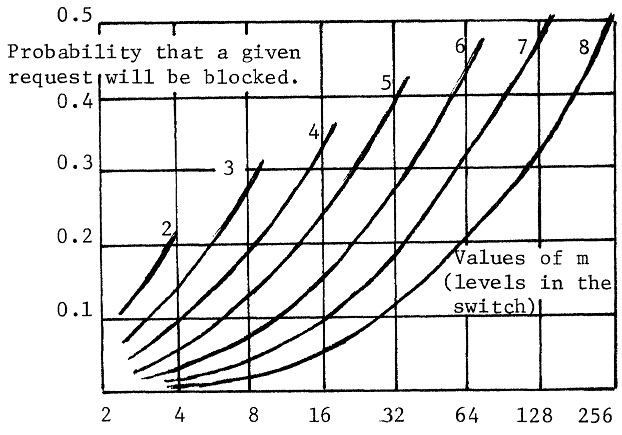


Figure 2. Example of a switch module



Number of lines in use. Figure 3. Blocking probabilities

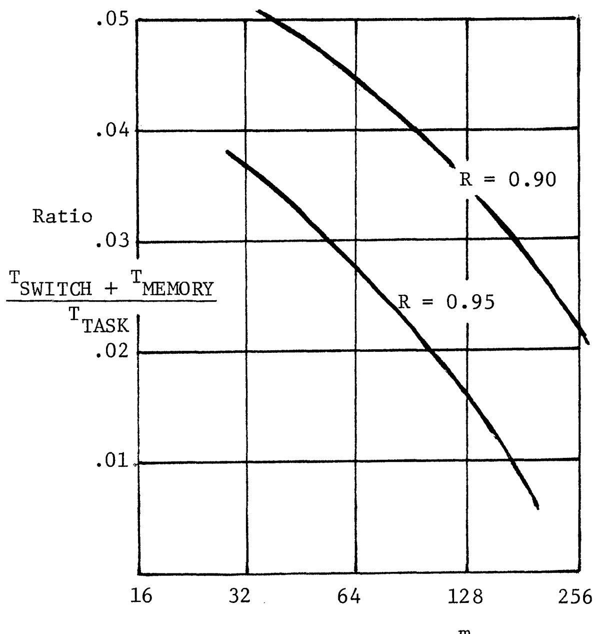
On . . . . n	Bit vector
100 x x x 101 x x x 110 x x x	Processor instructions
1110n . . n	Character string
11110n . . n	Small Integer vector
111110n . . n	Vector of Integers or Real numbers
1111110n . n	Vector of Double precision or complex numbers
11111110n . n	Double precision complex

n . . . n denotes a binary number giving the size of the data block

Figure 4 Proposed format for the data header.

Width of data path	8	4	8	bits
Record size (bits)	136	2060	32784	
Binary bits	128	2048	32768	
Characters (8 bits)	16	256	4096	
Small Integers (16 bits)	8	128	2048	
Integers and Real (32 bits)	4	64	1024	
Double Precision and complex (64 bits)	2	32	512	
Double precision complex	1	16	256	
Transfer Time T_{MEMORY}	17 T_N	515 T_N	4098 T_N	

Figure 5. Possible configurations of data blocks



Number of processors = 2^m

Figure 6. ratio of Switch and Memory delays to Task delay to achieve 0.90 and 0.95 processor utilization.

A CELLULAR DATA MANIPULATING ARRAY

I-Ngo Chen
Visiting Professor
Department of Electrical & Computer Engineering
Syracuse University
Syracuse, NY 13210

Summary

A cellular array for realizing data manipulating functions and general switching functions is presented. The basic cell of the array is similar to the comparison element of Batcher's sorting network [1]. Fig. 1 shows the basic cell and its state diagram. A cellular array is composed of identical cells arranged in a rectangular or square grid. Each cell in the array can be set initially to one of the 4 states. Since the cell covers those proposed by Akers [2], Kautz et al. [3], and Kukreja et al. [4], the array can be used to realize any switching function. Also, the array can be decomposed into 2 or more regions and a faulty cell can be detected and isolated.

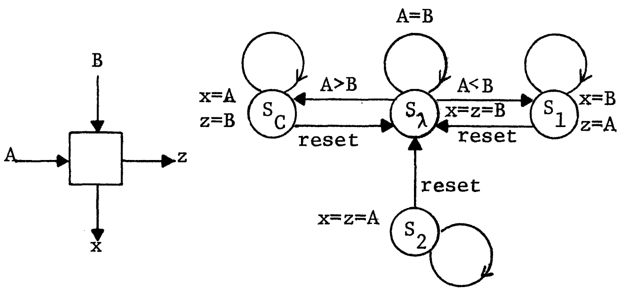


Fig. 1 The basic cell and its state diagram

For data manipulating, the control windings are illustrated in Fig. 2.

C_9	C_1	C_3	C_1	C_3	C_1
C_5	C_{10}	C_2	C_4	C_2	C_4
C_7	C_6	C_9	C_1	C_3	C_1
C_5	C_8	C_5	C_{10}	C_2	C_4
C_7	C_6	C_7	C_6	C_9	C_1
C_5	C_8	C_5	C_8	C_5	C_{10}

Fig. 2 Control Windings for data manipulating

The control function for shuffle is:
$C_1=C_3=C_5=C_7=C_9=0$
$C_2=C_4=C_6=C_8=C_{10}=1$
or interchanged;

for take-even:
$C_1=C_4=C_5=C_8=C_{10}=0$
$C_2=C_3=C_6=C_7=C_9=1$
or interchanged;

for duplication:
$C_1=C_2=C_3=C_4=C_5=C_6=C_7=C_8=1$
$C_9=C_{10}=2$;

for merging:
$C_1=C_2=C_3=C_4=C_5=C_6=C_7=C_8=C_9=C_{10}=\lambda$;

For compression and expansion, the skeleton vector has to be used as input to the top of the array. For compression, the control function is:
$C_1=C_2=C_3=C_4=C_5=C_6=C_7=C_8=1$
$C_9=C_{10}=$.

For expansion, the control function is:
$C_5=C_6=C_7=C_8=C_9=C_{10}=\lambda$
$C_1=C_2=C_3=C_4=1$

and a proper control sequence has to be applied. For sorting, shift, and flip, only the lower left triangular region will be used. All the cells within the region will be set to initially while those cells along the slant line are set to 0. For shift and flip, a control sequence has to be applied afterward.

Compared with the Batcher sorting network, our array has the same cell complexity. But the sorting algorithm employed in our array requires far more cells than those by Batcher's. The trade-off is uniform intercell connection and other data manipulating capability. Compared with Feng's data manipulators [5], the disadvantages of our array are in cell complexity and the number of cells required. The advantages are, again, uniform intercell connection and other computing capability like switching function realization and sorting.

References

[1] Batcher, K.E., "Sorting Networks and Their Applications", AFIPS Proc. 32, 1968, pp. 307-314.

[2] Akers, S.B., "A Rectangular Logic Array", IEEE Tranc. Computers, vol. C-21, No. 8, Aug. 1972, pp. 848-857.

[3] Kautz, W.H., et al., "Cellular Interconnection Arrays", IEEETC May 1968, pp. 443-451.

[4] Kukreja, N. and Chen, I.N., "Combinational and Sequential Cellular Structures", IEEE Tranc. on Computers, vol. C-22, No. 9, Sept. 1973, pp. 813-823.

[5] Feng, T.Y., "Data Manipulating Functions in Parallel Processors and Their Implementations", IEEE Tranc. Computers, vol. C-23, No. 3, March 1974, pp. 309-318.

A TWO DIMENSION PIPE-LINED PROCESSOR FOR COMMUNICATION IN A PARALLEL SYSTEM

V. Cordonnier
departement d'informatique
Université de LILLE
FRANCE

Abstract -- We present a two dimension network composed with identical cells. It gives good communication performances between a great number of processors or various units into a parallel system. Automatic routing of data along the network is done by a pipe-lined organization. Parallelism of communication allows a very high rate of data transfer and a low transfer time.

Introduction

The ability for fast and easy communication seems necessary within large parallel systems. The management of a set of diversified units,the access control into distributed memories and the use of new languages with parallel capabilities requires permanent and numerous mouvements of data.

The communication tool becomes as important as memory,arithmetic and logical unit or input and output controller. Even more,it is the link between them and has to be considered as an universal interface of the whole system.

A first family of solutions consists in mutiplying indépendant and specialized branches. Consequently,this leads to a developpement of an horizontal microprogramming technic : A set of different fields insures commands for groups of independant communication branches . However,there is a limit due to synchronization problems into a large amont of indépendant activities.

Bus solution represents a second family of devices. One bus can serve a number of different units in spite of a serial distribution of its tasks. Its data rate is limited by technical capabilities of the used circuits. Moreover,the conflicts and priority management reduces its performances.

The ideal characteristics for a communication tool appear to be the following ones:

1. User's point of view. From a strictly functional point of view,a communication processor is considered by users as a black box with as many inputs and outputs as necessary. Moreover,it it is able to recognize adressing commands and to perform routing of data according to these commands.

2. Tranparence. An exchange of data between two units and passing through the communication processor must take place exactly as if these two units where directly connected to each other.

3. Simultaneity. If many units require the services of the communication processor at the same time,it should be possible to serve all their demands instantaneously. The most usual way of communication into the system is " one unit towards one other"; other modes as " one unit towards many others" may be obtained by repeating the first one according various rules. So the rate of the departures and the rate of the arrivals are equal.

4. Assynchronism. At any time the communication processor has to accept a demand of service for a data transfer from any unit and to move this data immediatly towards any other one.

5. Modularity and extensibility. The units within the system must be of various kinds and interchangeable. So, interface between them and the communication processor must be standard. The number and the nature of these units can be altered just by adding new modules to the existing set of circuits into the communication processor. New modules are them considered as the other ones.

6. Speed and capacity. The time necessary for transmission of a data from a unit to another one must be short. Tipicaly il will be the same as the access time of a main memory into a classical system. The capacity of transmission is a direct result of the speed and of the level of simultaneity . For a speed of one micro-second per word and the possibility of ten simultaneous transfers,the capacity is of ten mega-words per second.

The STUD system

The communication processor we are presenting now has been studied for a system gathering one or several hundred of independant units of anay kind. Its name is STUD (Système de traitement à unités distribuées)

With such a number of users,the entire simultaneity cannot accept a direct connection between inputs and out puts. Each connection should require too much circuits and the cost should be too high.

We have be driven to study a pipe-lined solution inside which the transmission is ensured by intermediate cells with memory capabilities. Every cell takes part to one path for the propagation of data from input toward output. Switching operations on transmitted data are performed all along their way.

A direct solution must be prefered. It uses special intermediate circuits all along the route of data. In that way,intermediate processors or units are not affected by the exchange (fig 1a, 1b)

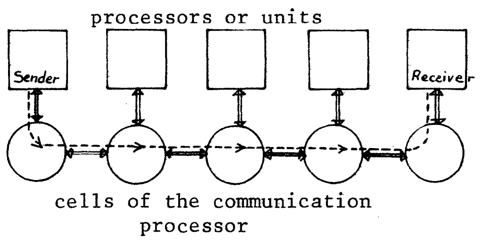


Fig 1a: Direct solution

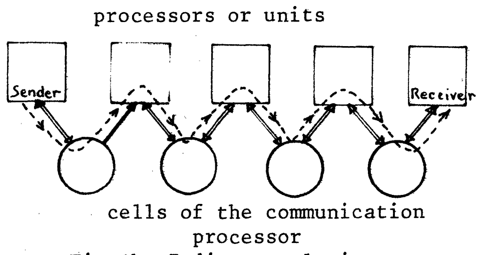


Fig 1b: Indirect solution

Another consideration has directed the choice of the solution: Adresses generation command and analysis must be simple. Indeed every cell of the structure must be encharged with the routing decisions into its own switching possibilities. For rapidity,microprogrammed analysis of adresses has been excluded and hard-wired solution seems absolutely necessary.

Among simple stuctures,the array has been adopted because it agrees with linear modes of adresses construction used into all data processing organizations. Another reason is that all the cells are absolutely identical. This is not possible with some other stuctures like trees for example.

For a N-dimension array with P cells in each dimension one may obtain the following results:

A- One cell has 2N neighbours.

B- The mean distance between two cells is:

- B1- NP/4 if the array is side-closed. (opposite cells may directly communicate)
- B2- NP/3 if the array is side-opened.

With our goal of one hundred users,a two dimension array seems efficient. For N=2 and P=10,every cell has four neighbours and the mean distance is seven.

However,we have examined one-dimension and three-dimension arrays. The first one gives good results with about twenty users. The second one is only justified for thousand of users.

The communication processor is designed according to the drawing of fig.2

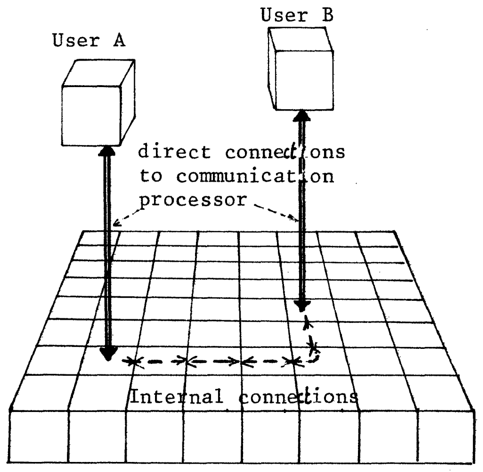


Fig.2 Array communication processor

With a two-dimension array,every cell has actually five neighbours: Four cells, up,down,right and left and its own user. When it receives a message,it is the only owner of it. This message and its track have been lost into the previously crossed cells. These cells become free for other messages.

In that way,the whole system runs like a two-dimension shift register with local decision of transmission according to routing rules and priority management.

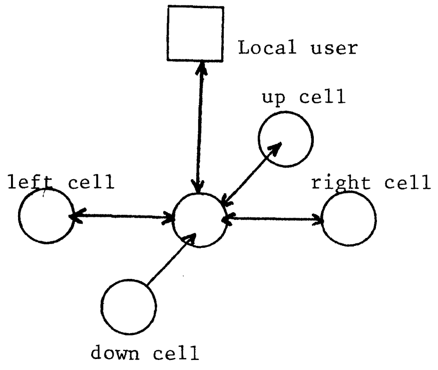


Fig 3. Cell's connections.

Routing rules

They are many possible ways between one sender and one receiver. The routing decision is taken for every path by the owning cell. we have to be sure that this sequence of independant decisions will be successful.

It would be possible to set connections between opposite cells of the array in order to reduce the number of steps for a message. This interesting possibility has been eliminated because of "dead lock" effects that appear into closed loops. In addition the routing rules are greatly complicated.

First we eliminate non-minimal routes: A minimal route between the sending user I,J and the receiving user K,L is determined by a number of steps:

ABS(I - K) + ABS(J - L)

Because of the reduced amont of information of a cell,a non-minimal routing should not be convergent.

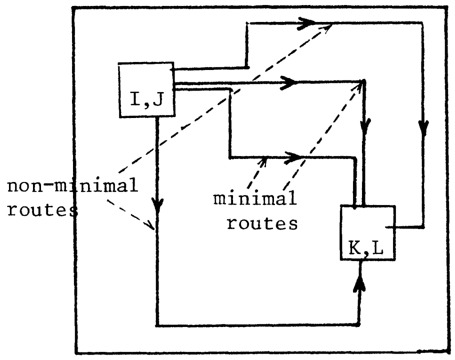


Fig 4. Possible routes into the communication processor

Then we eliminate adaptative routing for three major reasons:

A- If the user I,J sends the following sequence of data to te user K,L:

$M_1, M_2, M_3 \ldots\ldots M_j, \ldots\ldots M_1;$

message M_3 may arrive into K,L before message M_1 because of a successful route.Yet,it is necessary for the receiver to receive this sequence in the order of departure. So,a sorting operation would be undertaken after reception. That produces a loss of time. More,every message M_j ought to contain its index in the sequence. Otherwise this information is not necessary: The order of departure and the order of arrival are the same.

All the messages comming from I,J to K,L follow exactly the same way in the network and the routing rules are independant of the state of the cells.

B- Another consideration goes in the same direction: Good routing techniques result from a complex analysis about the whole state of the network. Such analysis is difficult to perform with hard-wired solution.

C- Eventually,routing techniques may introduce local "dead locks". In order to suppress these locks,special procedure of recognition and decision must be added. They bring a complication of circuits and a loss of time.

The routing rule we have chosen in the network is simple:

ALL HORIZONTAL STEPS ARE PERFORMED BEFORE VERTICAL ONES.

with this rule,inter-active tasks are made easier because questions and answers do not follow the same route.

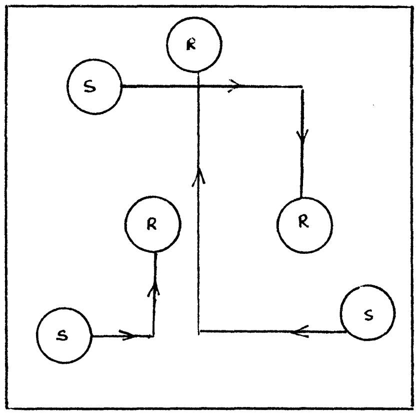


Fig 5. Examples of autorized route in the network

When a cell I,J contains a message bound to a cell K,L,it obeys the following algorithm:

```
BEGIN TRANSMISSION PROCEDURE;
   IF(J-L)10,20,30
10 IF CELL(I,J-1)= EMPTY THEN TRANSMIT
   TO CELL(I,J-1);
   GO TO END;
30 IF CELL(I,J+1)= EMPTY THEN TRANSMIT
   TO CELL(I,J+1);
   GO TO END
20 IF(I-K)40,50,60
40 IF CELL(I+1,J)= EMPTY THEN TRANSMIT
   TO CELL(I+1,J);
   GO TO END;
60 IF CELL(I-1,J)= EMPTY THEN TRANSMIT
   TO CELL(I-1,J);
   GO TO END;
50 IF LOCAL = EMPTY THEN TRANSMIT
   TO LOCAL INPUT OF USER
   END;
```

This decision rule may be applied to any message contained into the cell; it is not necessary to know where it comes from.

As a cell has five suppliers,it may have five messages during one cycle. It is possible to apply this algorithm in parallel over the five inputs and to supply the messages to the corresponding outputs.

However conflicts may exist between demands. This point will be discussed latter.

Format of messages

A message is the amount of data handled at a time by a cell of the network. It is considered as a single word.

This word contains:

1- The data to be exchanged.
2- Adress of receiver. This adress must be built by the sender.
3- Adress of the sender. This information is absolutely necessary because one unit may indertake various exchanges with several others. Then it has to know where the answer is comming from.

Parts 1 and 2 are issued from the sender, parts 1 and 3 are presented to the receiver.

As there is no trace of the route established for a message,a long sequence of data must be divided into individual words. Each of these words must contain the two adresses.

source adress	D A T A	destin. adress

Message format in the communication proc.

	D A T A	destin. adress

Message format in the sending unit

source adress	D A T A	

Message format in the receiving unit

Fig.6 Formats of messages

Realization

A cell is divided in two groups of circuits:

Registers.The registers contain the messages of the cell. The logic command will try to present these messages to:

Ports. A port is the way by the which a message can leave the cell.

They are five registers corresponding to the five inputs of the cell;they are five ports corresponding to the five outputs of the cell.

According to the previous statements, we find sixteen possible routes within one cell between registers and ports.

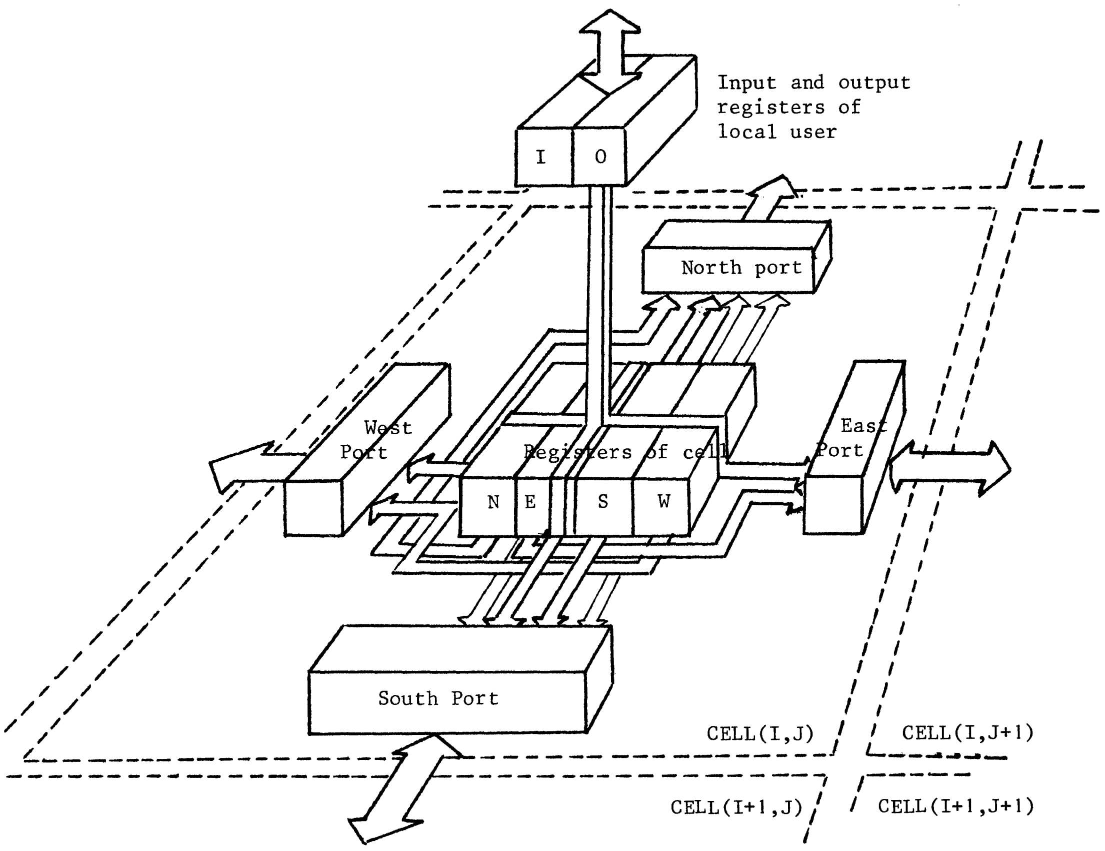


Fig. 7 - The cell organization.

Some of these sixteen routes may be activated simultaneously but,sometime,conflicts appear when one port is demanded by more than one register.

The first solution for resolving this problem of conflicts is to interrogate sequentialy the five registers with regard to the five possible ports. All registers may be gathered into a single memory.

In order to get better performances,one may examine with more care the nature of the conflicts. The matrix of connection between registers and ports is represented in fig. 8.

Registers

Ports	N	E	S	W	LOCAL
N	2	4	/	3	1
E	/	3	/	/	2
S	/	1	4	2	3
W	/	/	/	1	4
LOCAL	1	2	3	4	/

Fig.8 - Connection matrix.

Four cycles only are necessary to eliminate conflicts. The values in the matrix represent a possible distribution of these four cycles . Since every row and every column does only contains once each cycle number, conflicts are suppressed as well between registers and between ports.

This organization is very fast but registers with simultaneous read and write capabilities are required.

A clock is distributed all along the network and so,all the cells run with the same rythm. During one cycle it may send four messages and accept four other ones.

Before the realization with hard-wired circuits,a rigourous simulation of the entire network was essential in order to valuate the following points:

1- The total transfer rate of the communication processor.
2- The propagation time with regard to inter-cells conflicts.
3- The risks of saturation.
4- The relative speed of the network's and the user's circuits and clocks.

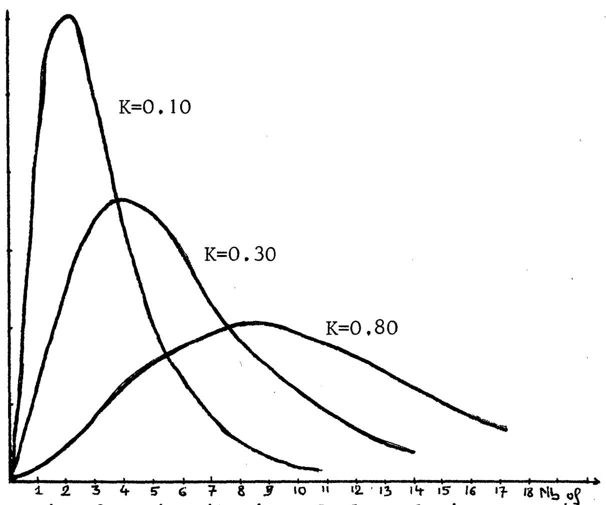


Fig. 9 - Distribution of elapsed time between departure and arrival

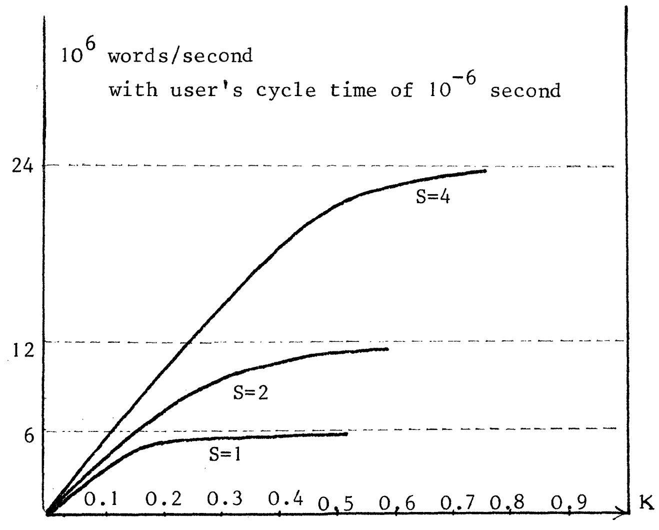


Fig. 10 - Total transfer rate in the network with three possible speeds of the circuits.

The results presented here concern a network of 64 cells (8 X 8). The first curves represents the distribution of elapsed time between the cycle of departure and the cycle of arrival into receiver. This time is a relative one. It is counted in number of cycles.

K Is the rate of demands from the cells. K=0.30 means that during 100 cycles of the network,the cell presents 30 demands.

The second figure represents the total rate of transfers. The variable is K. The S parameter represents the relative speed of the network on the one hand and of the users on the other hand. S=2 means that they are two cycles of the network during one cycle of demands.

The most important result of the simulation is that the network cannot be saturated and dead locked. When the rate of demands becomes highter,cells begin to refuse new messages and the flow keeps steady.

Simulation has been done for various networks till 256 (16 X16) and for various internal organizations of the cells.

Hard-wired realization.A network with 64 cells is now under project.A message is a word of 32 bits: 8 bits for adress of departure, 8 bits for adress of arrival and 16 bits of data. With actual LSI and MSI circuits,one cell requires about 30 chips. The processor will be installed as the back panel of a cabinet. The only obligation for using circuits is to interface with the two input and output registers of the cell into appropriate format. In addition every user must take out immediatly any message present in its input register.

The users of the communication processor.

From the point of view of the communication processor,one may characterize three types of users:

Passive and fast users. Such a unit has been designed for only one kind of task,always the same. In addition it can send its response before the arrival of the next demand. A typical example is a core or semiconductor memory. The access time must be shorter than the cycle time of the network.

Input register of this unit is adress register or write input,output register of the unit is read output ot the memory.

This definition possibly agrees with other types of units such as fast and specialized arithmetic and logical units or associative memories.

Passive and slow units. Here too,it is not necessary to undertake an analysis on comming messages. They are self-defined. But when the unit is occupied,other messages may be put into its input register by the network. This register will be connected to a push down stack. If the stack comes to be full, a "REFUSE" message must be send toward the demander. I/O controllers,specialized ALUs, slow memories belong to this family.

Active units or users. Such a unit is runing under the control of a program or microprogram. During this activity it will send demands to passive units or to others active units. Consequently it may receive responses or demands comming from other points of the network. For this matter we have characterized two classes of messages:

- Data messages;
- Control messages.

The difference between them appears in a few bits field of the message. When the field is zero,the word's other fields are considered as data. Otherwise,they are taken as commands.

In the realization four bits are devoted to this field.

According to the organization of the receiver,the status of the processor and the value of the field,different actions are possible:

- Interrupts;
- Start or stop the program;
- Send status;
- Activate DMA if existing;
- ......

Conclusion

New directions of investigation. We have distinguished the project into two parts:

1- A work about the communication processor itself. The main works are the definition of some well adapted LSI circuits and a new simulation program for showing how performances are affected by the use of a locality principle between senders and receivers.

2 - A work about the users. We consider the STUD communication processor as a tool for different experimentations about parallel processing.

The first one is the study of parallel operating system and languages.

Another direction is the study of a permanently adapted system. Units are ALUs, programm controllers,memories and I/O processors. One group of these units is a permanent builder. It receives external demands for jobs and builds "ephemeral" machines well suited to the character of the jobs. This organization is specialy devoted to real time applications.

At least,special applications that claim a great amount of identical processors may use the STUD processor; for example,telephone commutation or radar tracking.

In other respects,we think that the network structure of the communication processor allows easy reconfiguration procedures to run and circuit failures detection and correction mechanisms to be added.

So far,communication problems within computers had found local and specialized solutions. With the increasing importance of parallel techniques,we think that it is essential to develop general tools. This paper tries to present one solution but they are many other directions for investigation about this problem.

-o-o-o-o-o-o-o-o-o-o-o-

THE DESIGN AND IMPLEMENTATION OF A HIGH/LOW MAGNITUDE SEARCH INSTRUCTION ON PEPE

M. C. DIVECCHIO
Burroughs Corporation
Paoli, Pa. 19301

Summary

PEPE, the Parallel Element Processing Ensemble, is a highly parallel content addressable computer, capable of executing three independent instruction streams concurrently[1]. A need was identified in PEPE for a high speed algorithm to perform a maximum or minimum search over a set of data values in the element ensemble. The algorithm is discussed in this paper.

Each of the three units in PEPE contain sequential control logic and a parallel instruction control unit (PICU). The PICUs each control their respective third of each processing element (PE) [2]. The PE contains three independent processors, two of which are able to perform this search function. Each PE contains an Element Activity (EA) flip-flop which controls its participation in this operation. PEPE also includes a set of logic called distributed logic. This logic is not contained in any one PEPE unit but is spread over the control unit and the element bays. The comparison logic involved in the search instruction is this type of logic. The two search instructions are Select Highest and Select Lowest (SH/SL). The SH/SL instructions execute as follows: in the set of active PEs, reset element activity in each PE whose "A" register contains a nonmaxial/minimal value relative to the set of active PE. This value comparison is done for the set of PEPE integer or normalized floating point numbers [3].

The algorithm executes in the AU and the AØU and requires a maximum of 35 100 ns micro-steps. Execution consists of two parts, Conversion and Search. The first micro-step of SH/SL converts the set of values to be compared from the valid set of PEPE 2's complement number system into an ordered set of operands. This conversion is a mapping of PEPE numbers onto a 32 bit pure magnitude number line. In the case of Select Highest, the largest (most positive) PEPE number is converted to all ones and the smallest (most negative) PEPE number is converted to all zeros. For Select Lowest the mapping is reversed. The conversion is done in the ALU of the element. Following this mapping, each element contains a value in pure 32 bit magnitude representation with the same relative "value" as the original PEPE number. All that remains is to compare this value in all active elements and leave the element(s) with the maximum/minimum value active.

This work was supported by the Ballistic Missile Defense Advanced Technology Center (BMDATC), Huntsville, Alabama, under Contract No. DAHC60-73-C-0060.

The basic principle behind the search is as follows: two bits of the converted value in each PE "A" register are decoded into three signals named A, B and C and are gated out of the PE and into the Signal Distribution System (SDS). In the SDS each of the three lines are "OR"ed together with the corresponding lines from all of the active PEs. The three outputs of the final OR gates are called the X, Y and Z lines and are sent back to every element. The element then compares its local A, B, C lines with the global X, Y, Z lines and any elements whose ABC lines indicate its 2 bits are less than the XYZ lines show for the entire system, will reset its element activity.

When a and b are the two bits being examined, the equations for the search lines and reset EA function are:

$$A = ab$$
$$B = a\overline{b}$$
$$C = \overline{a}b$$
$$\text{RESET} = \overline{A}X + \overline{AB}Y + \overline{ABC}Z$$

The PICU does not directly participate in the actual search. It sends controls to the elements directing the element to generate the A, B and C lines, to do the mapping, to compare the X, Y and Z lines and to reset the element activity if the compare so indicates. The PICU can end the search early. If during the execution of the search instruction, the element activity count goes to one, the PICU terminates execution of the instruction. This early end is possible because, if there is only one element left, it has the highest or lowest valued "A" register in the set of active elements.

References

[1] J. A. Cornell, "Parallel Processing of Ballistic Missile Defense Radar Data with PEPE", IEEE COMP CON 1972 Digest.

[2] Alf J. Evensen and J. L. Troy, "Introduction to the Architecture of a 288-Element PEPE", 1973 Sagamore Computer Conference on Parallel Processing, 1973.

[3] PEPE System Functional Design Specification, Vol. II, Hardware Specification, System Development Corporation, Revision D, August 1974.

THE ASSOCIATIVE LINEAR ARRAY PROCESSOR

Charles A. Finnila
Data Systems Division
Hughes Aircraft Company
Culver City, California 90230

Summary

The Associative Linear Array Processor (ALAP) is designed to make very large associative memories practical. (See reference [1] for a survey of associative memories.) In order to achieve this goal, many techniques are used. The data storage is in shift registers. The arithmetic and data transfers are bit-serial. Arithmetic and control logic are combined with each data storage shift register to form a word cell. The word cells are fabricated using metal-oxide-silicon (MOS) technology to form a type of large scale integrated (LSI) circuit. Enough matching and arithmetic capability is included to service large, real time data bases within ALAP memory. Exact match and arithmetic limit match capability is provided as well as all of the basic arithmetic operations, including square root. Electronic fault isolation is provided so that defective word cells may be deactivated and will not interfere with the use of other good cells in the array.

The basic ALAP configuration is shown in Figure 1. The word cells form a line of processing-plus-memory elements. Most data transfer and control communication is by means of the group of bit-serial busses, which are common to all words. The extensive use of common busses is made practical by the multi-use chaining channel, which is the only bus which is not common. This provides bit-serial communication between adjacent elements in the ALAP linear array. The common data and control block interprets the program to be executed in the ALAP memory array. The program is stored in a random access program memory.

Like the other resources of each word cell, the current function of the multi-use chaining channel is determined by the combination of the state of common control busses and control flag flip-flops in each word cell. In general, different words will be in different chaining channel modes. Some of the chaining channel modes are "relay" (word chaining input to chaining output), "chain" (chaining input to shift register and shift register to chaining output), "clocked relay" (chaining input to head flag flip-flop and head flag to chaining output), and "recirculate" (data recirculated in shift register and output on chaining channel). Usually arithmetic (using arguments from either the chaining channel or the common input) and data input and output on common busses are taking place simultaneously with the chaining channel operations.

Combinations of chaining channel modes have many uses. For example, when it is necessary to output (on the common output bus) several words which all match, a marker flag bit can be passed down the chaining channel (using relay and clocked relay) to signal the next matching word to be output. As another example, new data can be sorted as they are loaded into a contiguous block of ALAP memory cells. A limit match in the field to be sorted can flag all of the words with that field equal or larger. These words can then be put in the chain mode while the other words are in recirculate. The words in the chain mode move over to leave room at the same time that the new word is loaded into its sorted location.

For more on ALAP see reference [2]. A multi-cell LSI wafer has been fabricated. A complete processor with software has been built and tested.

References

[1] B. Parhami, "Associative Memories and Processors: An Overview and Selected Bibliography", Proceedings of the IEEE (June, 1973), pp. 722-730

[2] B. F. Meyers, The Hughes Associative Processor, Computer Science Dept., Univ. of Calif. at Los Angeles, Modeling and Measurement Note #26 (May, 1974)

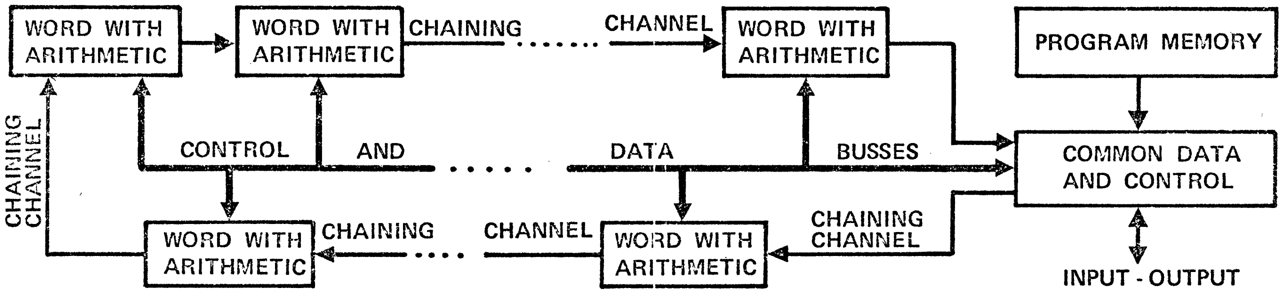


Figure 1. ALAP Configuration

PROGRAMMING THE ASSOCIATIVE LINEAR ARRAY PROCESSOR

Hubert H. Love
Hughes Aircraft Company
Culver City, California 90230

Two stand-alone application programs for the ALAP have been programmed and checked out, using a symbolic assembler and a functional simulator, both of which execute on the XDS Sigma 5 computer. One of these is a track-while-scan program which performs correlation, association and track prediction. The second, described here, is a fact-retrieval demonstration program. This program retrieves and outputs the names of all items which are in a given set of attribute-value relationships with other items, as defined by a query input by the user.

The data base, resident in the ALAP memory, is in two parts. One of these consists of a set of records, called "A-V records". Each record consists of a subject item and all of its attribute-value item pairs. Each distinct item in an A-V pair is also the subject of a record, thus making the file structure complete. All items are represented in these records by fixed-length binary "item numbers." The second part of the data base is a directory which corresponds each item number with an alphanumeric "item name" of arbitrary length for representing the item to the user.

Figure 1 shows two entries from a sample directory. The first and last words of an entry contain the item number for the entry. The intervening words contain the item name, divided into segments of seven (or fewer) characters. The eighth (last) character position in every directory word contains a tag denoting the top header (&), bottom header (#), first item name segment (*), last (or only) segment ($), or intervening segment (@).

Figure 2 shows two sample A-V records having as subjects the two items of Figure 1. The last word of each record, tagged by "%", is the header word. It contains the item number for the subject of the record. The other words in the record, tagged by "+", each contain the pair of item numbers for an A-V pair belonging to the subject.

1 0 4 6	&
C A T b b b b	$
1 0 4 6	#
2 4	&
H I P P Ø P Ø	*
T A M U S b b	$
2 4	#

Figure 1

9 1 0 7	6 3 6	+
2 7	2 3	+
3 9 9 1	4 0 2 6	+
1 0 4 6		&
1 2 1	4 8	+
2 4		&

Figure 2

The user's query consists of a set of pairs of item names. These are the A-V pairs which any retrieved item must contain in its A-V record. The program first substitutes item numbers in the query in place of the corresponding item names. This is done for each item name by comparing each successive seven-character (or less) segment of the given item name against the corresponding segments of all item names in the directory simultaneously. The ALAP memory logic AND's the results of the successive comparisons. The chaining channel simultaneously transfers the results of each such compare/AND operation from the directory words at which the operation is performed to the next following words in the entries. After the last segment of the given item name has been thus processed, at most one word in memory will still be tagged. The chaining channel is used to advance this tag to the next word, which is the bottom header word. The item number is then retrieved and substituted for the item name in the query.

The program now processes in turn each A-V pair from the query against the A-V records. First, a content-addressing operation is performed to simultaneously tag all words in ALAP memory containing the first A-V pair. Next, a single chaining channel operation transfers the flag settings from all tagged words to the corresponding header words, where they are saved in one of the flag bits. This entire process is then repeated for each of the remaining A-V pairs in the query, except that the flag settings denoting the results of the comparisons are AND-ed with the flag settings previously saved at the header words, the results replacing the latter flag settings.

After the last A-V pair has thus been processed, the only words still tagged will be header words for the items satisfying the query. The program now retrieves each tagged item number and makes a single comparison against the directory. This locates the top header for the corresponding directory entry. The item name is then retrieved, a segment at a time, and output to the user.

The program does not perform modification to the data base. However, the chaining channel can be used to shift the contents of any specified set of contiguous words any desired distance in a single operation, exactly as though they constituted a single long shift register. The execution time for such a shift depends only on the length of the shift, and is independent of the number of words involved. This operation can be used to open space in the middle of a record or between records or entries for the insertion of new data.

ARCHITECTURE FOR A HIGHLY RELIABLE PARALLEL COMPUTER SYSTEM

W. W. Gaertner
W. W. Gaertner Research, Inc.
Stamford, Connecticut 06903

Summary

The work described in this paper had the objectives of determining (1) what levels of performance can be achieved in a parallel-processor computer system making extensive use of off-the-shelf building blocks in main-stream technology; (2) what architectural features would make such a computer system highly fault tolerant; (3) what architecture would maximize processing speed per unit cost.

The study resulted in the following conclusions:

Most applications which benefit from the high throughput of the parallel processors also require a mass-memory system of high capacity and very high data-transfer rate. To achieve highest throughput and hardware utilization, the architecture should allow "total-activity parallelism" of the hardware, in the sense that e.g. the execution of an instruction in the ALU, the fetching of the next instruction, the fetching of the next data, the storing of the previous outputs from the ALU, the rearrangement of data in the memory, performance monitoring, fault location and recovery, all occur simultaneously. At the same time, the rates of all activities should be matched to each other for a given class of algorithms.

Using these guidelines, a parallel computer has been designed [1] which is organized into a Control Computer Complex (CCC), a Processing Element and Data Routing Element Array (PE&DRE Array) and a Mass Memory System (MMS). The Control Computer Complex is an off-the-shelf mini or midicomputer with standard peripherals. It is made triple redundant, if necessary to achieve system availability specifications [2]. The parallel Processing Elements (PEs) carry out all parallel and associative computations and are implemented by off-the-shelf microcomputers with certain significant modifications. The Data Routing Elements (DREs) provide simultaneous high-speed communication capability between all PEs, and allow a pipeline operation of the machine. The Mass Memory System (MMS) is also organized into parallel modules, implemented by off-the-shelf disks and/or CCD memories. A Mass-Memory Buffer/Multiplexer matches the mass-memory transfer rate to the PE processing rate. With an instruction rate of 0.25 to 4 MIPS (16-bit word) per PE (depending on internal hardware implementation) computer systems between 32 and 1,024 PEs will achieve 8 to 4,096 MIPS. The mass-memory transfer rate is 1.2 Mbytes/sec per disk module or as high as 40 Mbytes/sec per CCD module.

The architecture achieves very high reliability and availability through the use of redundancy, built-in switchable spares [3], [4] and on-line maintenance. Specifically, in the PE&DRE Array there are two spare PE&DREs assigned to each group of 32 PE&DREs and one spare is provided for every 8 Mass Memory Modules. If faulty modules are replaced every four hours, a system with 128 PEs (32-512 MIPS) and 32 Mass Memory Modules has a reliability, at 20,000 hours, of 0.9817 times the reliability of the Control Computer Complex.

References

[1] W.W. Gaertner, D.B. Ellingham, Jr., L.T. Fiore, C. Hung, R.J. Tolmie, Jr. and W.M. Schreyer, Architecture for a Highly Reliable Parallel Computer System, W.W. Gaertner Research, Inc., Final Report under Contract F30602-72-C-0462 (June 1975)

[2] W.W. Gaertner, D.B. Ellingham, Jr., L.T. Fiore, C. Hung, R.J. Tolmie, Jr. and W.M. Schreyer, Computer-Aided Design of Digital Systems with High Availability and Maintainability, W.W. Gaertner Research, Inc., Final Report under Contract F30602-73-C-0304 (Nov. 1974)

[3] W.W. Gaertner, T.C. Booth, F.L. Hajdu and R.A. Reiss, Multifunction Module Design for Improved Logistics Support, W.W. Gaertner Research, Inc., AD Nos. 886485L and 886486L (June 1971)

[4] W.W. Gaertner, ed., Adaptive Electronics, Artech House, Inc., (1973) 279 pp.

PARALLEL PROCESSING IN A CELLULAR LOGIC ARRAY

D. R. Smith, K. S. Lin and Y. S. Shen
Department of Computer Sciences, S.U.N.Y.
Stony Brook, N.Y. 11794

Summary

Cellular logic has been studied for a number of years as an attempt to anticipate and exploit the peculiar characteristics of integrated circuit technology.

The actual construction of a hardware model forces a confrontation with any inherent weaknesses of theoretical proposals [2, 3, 4] as well as providing an experimental laboratory with which to study the software appropriate to such an architecture.

A prototype cellular array implemention with the proposed cell model shown in Fig. 1 (a) is currently under construction in the Department of Computer Sciences, S.U.N.Y. Stony Brook. The cell model here is a compromise designed to avoid basic performance limitations of those in [2] and [3]. The internal control C_I offers a combination of semi-permanent and dynamic control of the cells in the array. All the functions inside a cell are realized by using a 256x4 PROM and a Dual D-type Flip Flop as shown in Fig. 1 (b). A design for a microprogrammed controller which attaches to the direct memory access bus of a PDP-15 computer is shown in Fig. 2. The cellular array itself is arranged in two 18-bit wide and 1-bit tag column sets. These are necessary to serve functions analogous to the AC-MQ registers and Link of a conventional computer for arithmetic operations, and act as key, tag and data field in the associative operations. A latch register and two majority gates per row are used to offer both external and internal feedback control over the horizontal control and input lines, C_H and Z_o. Vertical control lines, (C_y) and clock lines (P1, P2) are grouped and controlled in zones by bits in a microprogram instruction register (MIR). The vertical control/data lines (C_D) are used as a one way direct access bus to every row in the array.The array is paged into suitable size to reduce the cycle time of array operations. A selection of array algorithms has been microprogrammed and simulated. These algorithms which use extensively the bulk data processing capability of the array show a speed-up improvement, as compared with the best known algorithms in serial computers, which often exceed the Minsky bound [5]. (See Appendix).

The above results are based on the current architectural structure, i.e., the cellular array is an add-on unit to a general purpose computer. It is conceivable that more powerful and general applications could be achieved if the cellular array were embedded inside the CPU of a computer as a functional unit or a mixed memory, and functioning cooperatively and in parallel with the CPU. It is observed that a cellular array has essentially all the bulk processing capability of the SIMD type of computer [5].However, in terms of performance/cost a cellular array can be much superior to any SIMD computer.

Appendix

Operations	Speed-up Factors
	Using Burst Shift I/O
Vector Addition & Multiplication	αK^*
Recurrence Relations	
Addition-Recurrence	αK
Binomial Coefficient	αK
Factorial Function	$\alpha K/\log_2 K$
Sorting	$\alpha \log_2 K$
Searching	$\alpha \log_2 k$ to K
Symbol Table	$\alpha \log_2 K$ to K
Histograms	$> \alpha \log_2 K$
List Processing (substring search)	αK

* K is proportional to the number of rows

References

[1] Minnick, R. C., "A Survey of Microcellular Research", J. ACM, (April, 1967) pp. 203-241.

[2] Kautz, W. H., "An Augmented Content-Addressed Memory Array for Implementation with Large-Scale Integration", J. ACM, (Jan., 1971) pp. 19-33.

[3] Jump, J. R. and Fritsche, D. R., "Microprogrammed Arrays", IEEE Trans. Computers (Sept., 1972) pp. 974-984.

[4] Cornel, R. C. and Torng, H. C., "A Cellular General Purpose Computer" 2nd Annual Sym. on Computer Architecture, ACM-SIGARCH, Conference Prodeedings (Dec., 1974) pp. 207-213.

[5] Flynn, M. J., "Some Computer Organizations and Their Effectiveness", IEEE Trans. Computers (Sept., 1972) pp. 948-960.

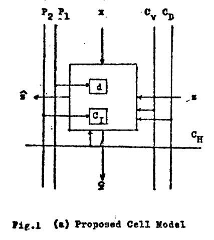
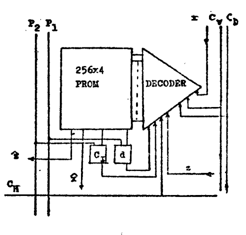
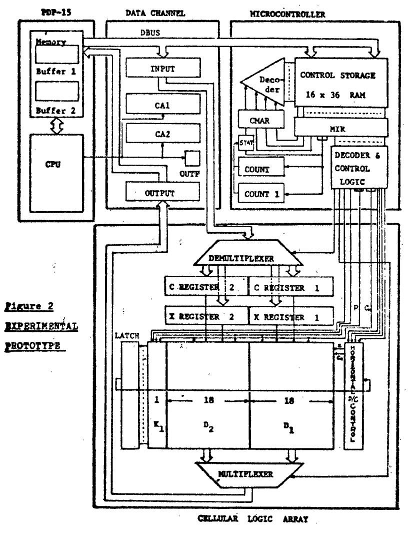


Fig.1 (a) Proposed Cell Model



(b) Actual Implementation

Figure 2 EXPERIMENTAL PROTOTYPE



THE DESIGN OF PROGRAMS FOR ASYNCHRONOUS MULTIPROCESSORS[a]

Philip H. Mason
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Summary

This research develops both a methodology for enhancing the design of programs to be composed of concurrently executable subparts, and a set of tools to support that methodology. The execution environment which we shall be concerned with consists of several processing units operating under the control of separate instruction streams. Intuitively, when parts of a program are processed in such an environment, the real time[b] required to execute the program should decrease[c]. For this reason, as well as others, much current research effort addresses program structure for just such a multiprocessing environment. This paper addresses the task of decomposing problems for concurrent execution in such a way that the decompositions are efficient with respect to certain specifiable performance criteria. The approach is to provide a set of tools with which a system designer can manipulate and analyze a program model. The model is then simulated to predict the performance of a system designed for a multiple asynchronous instruction stream environment. The tools are applicable to both the early design of a program and later tuning of a program under construction.

There are several reasons why researchers are considering multiprocessing and problem restructuring instead of just building faster computer hardware without explicit concurrency. First, certain problem applications overwhelm current and projected technology when programmed for single instruction stream computers. An example is the problem of weather forecasting for any single place on earth. A second reason is the benefit that may be achieved, through explicit multiprocessing, in lower cost for similar computational power. A set of 16 minicomputers can have the same computational power as a uniprocessor costing 4 to 8 times as much as the minicomputer system and may be used to solve problems as complex as those designed for the uniprocessor environment. Perhaps the most compelling reason (probably a consequence of the first two) for wanting to decompose programs for multiprocessing environments is that soon such environments will be available and it will be important to use them properly [2, 3, 4, 5].

There are, at present, no guidelines for decomposing a problem for multiprocess execution. This research is directed towards answering the following questions:

1. How can interactions among concurrent computations be modeled?
2. Are the interactions safe, i.e. deadlock free? For example, can the program arrive at a state in which one process is trying to communicate with a second process while the second process is waiting to send a communication to the first process?
3. Where will most of the process and communication activity occur?
4. Where may bottlenecks occur and how may they be relieved? For example, will the introduction of buffers or additional processors help?
5. Are there working sets of processes? If certain subsets of processes tend to be activated at different times then fewer processors will be required for a program.
6. What are the effects of restricting the number of processors? What are the effects of alternative scheduling algorithms?
7. How do alternative program decompositions compare with each other, and can good comparison standards be identified?

A number of questions relating to guidelines for the decomposition of programs for multiprocessors have been investigated. Previous research has progressed mainly along two directions: theoretical models of computation [6, 7, 8, 9], and construction tools for multiprocessor programs [10, 11, 12, 13]. The theoretical model that has been studied the most is the Petri net model [14, 15]. The main drawback in using Petri nets is that in order to study any reasonably interesting structures the number of nodes necessary often becomes very large. Another model of concurrent computation that explored questions of mean path time traversal and determinism was the UCLA graph model [16]. A problem with this model is that many of the results that were obtained utilized acyclic graphs. All loops were expanded by some suitable or estimated repetition factor. This property of the model diminished its usefulness due to the large numbers of nodes that were required.

The present research approaches the decomposition problem with graph tools, analysis tools, and simulation

(a) This work was supported by the Department of Defense Advanced Research Projects Agency (F44620-73-C0074) and is monitored by the Air Force Office of Scientific Research.

(b) "Real time" is the time elapsed between the start of computation and the time the final result is available. It is different from the total processing time since operations may be performed concurrently.

(c) This does not always occur. Graham [1] has shown that adding more processors can increase real time due to scheduling anomalies.

tools. The graph structure is a directed graph with the activity of its nodes being determined by parameters, as well as by the flow of tokens. A certain amount of analysis can be performed upon a model and finally a model can be simulated and data collected concerning the performance of a particular decomposition. All of these tools exist as an interactive computer system called STEPPS (Some Tools for Evaluating Parallel Programs).

The STEPPS model contains two components. These are a connected directed graph representing potential communication paths, and sets of attributes which determine how nodes operate and set other limitations on a model's operation. There are two types of nodes in the STEPPS model: process nodes and link nodes. Process nodes are only connected to link nodes and visa-versa. Tokens, called messages, traverse the graph and may be deposited in the links between processes.

All processing is modeled as occurring within process nodes. The operations of the process nodes are defined to be similar to semi-Markov process descriptions. Each process is composed of a set of states, a set of transition probabilities between states, and compute times between state transitions which are dependent on each intraprocess state transition. The state of a process is identified as the last input/output operation performed by the process. Thus each connection between a process and a link node represents a different process state. Each process has an initial state.

A link node functions as a finite queue for messages sent by processes and as a queue of requests for messages required by processes. In addition, a link can be used to designate delay due to interprocess communication. Messages are tokens that contain only immediate information such as which process sent the message and what time it was sent. There are no other data associated with a message; not even a message's destination. Each link can have an initial volume of messages.

It has been shown that both Petri nets and the UCLA graph model can be subsumed by the STEPPS model [17].

A very important feature of the STEPPS system is the implementation of an algorithm to test for the existence of potential deadlocks in the model. This algorithm consists of a set of graph reductions that preserve the message flow structure of a model, but ignore the particular process transition values. The algorithm consists of iteratively applying a set of four graph reductions to the graph until no reduction is applicable. If the resultant graph is empty then the original model represented a safe (non-deadlock) structure. The reductions are: combine adjacent processes, combine parallel processes, eliminate states of a process, and remove single state processes. The algorithm has been proven correct [17].

The STEPPS system provides features to facilitate entry, manipulation, display, saving and recalling of a model. The STEPPS simulation system provides facilities to restrict the number of available processors and compare models' performance using a variety of scheduling algorithms. Data are collected to determine such attributes as expected queue lengths, wait times, process states, number of active processes, number of ready processes, and working sets of processes. These data are used to determine alternate model structures for comparison.

The STEPPS design methodology including the STEPPS model and interactive system has been applied to several examples. The two major examples are a multiprocess Bliss/11 compiler [18] and the Hearsay II speech understanding system [19]. The results of experiments with these models demonstrate the ease with which these design tools can be used to predict performance properties before commitments are made to particular multiprocess structures.

Bibliography

[1] R.L. Graham, "Bounds on Multiprocessing Anomalies and Related Packing Algorithms," SJCC, (1972), pp. 205-217.

[2] W. Wulf, and C.G. Bell, "C.mmp -- A Multi-mini-processor," Proc. AFIPS 1972 FJCC, vol. 41, AFIPS Press, Montvale, N.J., pp. 765-777.

[3] F.E. Heart, S.M. Ornstein, W.R. Crowther, and W.B. Barker, "A New Minicomputer/multiprocessor for the ARPA Network," Proceedings of the National Computer Conference, (1973), pp. 529-537.

[4] S.H. Fuller, D.P. Siewiorek, and R.J. Swan, "Computer Modules: An Architecture for Large Digital Modules," ACM SIGARCH 1st Annual Symposium, Gainesville, Fla. (Jan. 1974).

[5] J.T. Quatse, P. Gaulene, and D. Doge, "The External Access Network of a Modular Computer System," Proc. SJCC, (1972), pp. 783-790.

[6] R.M. Karp, R.E. Miller, "Properties of a Model for Parallel Computations, Determinancy, Termination, Queueing," SIAM J. Appl. Math., vol. 14, #6, (Nov. 1966), pp. 1390-1411.

[7] J.A. Gosden, "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-processor Computing," FJCC, (1966), pp. 651-660.

[8] W.L. Miranker, "A Survey of Parallelism in Numerical Analysis," SIAM Review, vol. 13, #4, (Oct. 1971), pp. 524-547.

[9] R.M. Keller, "Parallel Program Schemata and Maximal Parallelism I: Fundamental Results," JACM, vol. 20, #4, (1973), pp. 514-537. [10] D.A. Adams, "A Model for Parallel Computations," in Parallel Processor Systems, Technologies and Applications, ed. by L.C. Hobbs et al., (1970), pp. 311-333.

[11] D. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, Computer Science Department, Carnegie-Mellon University, Report (Aug. 1971).

[12] V. Lesser, The Design of an Emulator for a Parallel Machine Language, Stanford University, (1972), Ph.D. Thesis.

[13] W. Riddle, The Modeling and Analysis of Supervisory Systems, Stanford University, (1972), Ph.D. Thesis.

[14] C.A. Petri, Kommunication mit Automaten, Translated in Project MAC-M-212 Report, originally published in 1962.

[15] A. Holt, and F. Commoner, "Events and Conditions," in Concurrent Systems and Parallel Computation Conference, ACM, (1970), pp. 1-52.

[16] J.L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing," Computing Surveys, vol. 5, #1, (Mar. 1973), pp. 31-80.

[17] P.H. Mason, Design Tools for Asynchronous Multiprocessor Programs, Computer Science Department, Carnegie-Mellon University, (1976), Ph.D. Thesis.

[18] W. Wulf, et al., The Design of an Optimizing Compiler, (1975), American Elsevier.

[19] R.D. Fennell, HSII: The Asynchronous Version, Computer Science Department, Carnegie-Mellon University, (1975), Ph.D. Thesis.

A CRITERION FOR SYNCHRONIZATION SCHEMES

Robert C. Chen and James E. Coffman
The Moore School of Electrical Engineering
The University of Pennsylvania
Philadelphia, Pennsylvania 19174

Summary

This paper discusses a criterion useful for judging the range of applicability of schemes for representing process synchronization. The development of this criterion, motivated by a basic shortcoming of several well known representation schemes [1] - [4], raises the issue of capability of representation schemes as a consideration separate from other issues such as convenience or efficiency.

The basic limitation of the scope of representation of these schemes arises from the fact that though events may be enabled by the fulfilling of a condition it is never the case that an event is enabled by the absence of a condition. A reader - writer system can be described [5] which cannot be modelled by any of these schemes. Kosaraju [6] has shown the same limitation for Petri nets and the P and V scheme.

To overcome this limitation we simply add a very simple primitive. The resultant schemes do not suffer from the limitation. However, it is clear that there might still be concurrent systems which cannot be modelled even by the augmented schemes. To argue that this is not the case, we must define the general class of systems to be modelled and find a convenient scheme capable of representing all systems in the class. We define our criterion for representation schemes: a scheme is complete if it can represent the same class of coordinations as nondeterministic Turing transducers, where by "represent" we mean to generate the same set of "output tapes" for a given "input tape". This criterion is exactly that developed independently by Agerwala [7]. We have shown [8] that the augmented schemes are complete in our sense.

References

[1] Anatol W. Holt and Fredrick Commoner, "Events and Conditions," Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM (1970), pp. 3-52.

[2] F. W. Dijkstra, "Cooperating Sequential Processes," Programming Languages, ed. F. Genuys, Academic Press (1968), pp. 43-112.

[3] P. Wodon, "Still Another Tool for Controlling Cooperating Algorithms," Carnegie-Mellon University Report, (1972).

[4] H. Vantilborgh and A. van Lamsweerde, "On an Extension of Dijkstra's Semaphore Primitives," Information Processing Letters I, (1972), pp. 181-186.

[5] James E. Coffman, "Synchronization Schemes and Completeness," (Master's Thesis, Moore School of Electrical Engineering, University of Pennsylvania, 1975).

[6] S. Rao Kosaraju, "Limitations of Dijkstra's Semaphore Primitives and Petri Nets," Hopkins Computer Research Report #25, (May 1973).

[7] T. Agerwala, "A Complete Model for Representing the Coordination of Asynchronous Process," Hopkins Computer Research Report #32, (August 1974).

[8] Coffman, (1975).

A Comparative Analysis of Two Parallel Computation Models

by

Susan C. Meyer
Department of Mathematics
Clarkson College of Technology
Potsdam, New York 13676

J. Robert Jump
Laboratory for Computer Science and Engineering
Department of Electrical Engineering
Rice University
Houston, TX 77001

Summary

A computation graph, as proposed by Karp and Miller [1], models parallel computations by means of a finite labeled directed graph. Each vertex of the graph represents an operation and each edge is interpreted as a queue which may contain data. Performance of an operation then causes operands to be removed from incoming edges and results to be placed on the edges directed away from the vertex representing that operation. An operation may occur only if there are a sufficient number of operands available on each of its input queues.

One of the models studied in this paper, called a bounded computation graph, differs in only one respect from the Karp-Miller computation graph. A finite, though arbitrarily large, bound is imposed on the length of each queue. Then an operation may occur only if there are a sufficient number of operands available on each of its input queues and none of the output queues will overflow as a result of the performance of that operation.

Though imposing a bound on queue length does indeed restrict the Karp-Miller model, the class of computations which can be described within this model is still quite useful. For if one wishes to implement the computation represented by a computation graph, the lengths of the queues must be bounded in some way. Furthermore, Karp and Miller derive a test for the boundedness of queue lengths in a computation graph, and it is clear that by making the imposed bounds sufficiently large, we may represent within the restricted model any computation graph having bounded queue lengths. In addition, it has been shown [6] that if the restriction on queue lengths is removed, a comparative evaluation may be carried out along very similar lines to those presented in this paper.

It has been observed [3]-[5] that every finite marked graph [2], [3] may be viewed as a special case of the Karp-Miller computation graph. We have shown that the class of computations which can be modeled by finite marked graphs and the class represented by bounded computation graphs are very nearly identical. Hence the bounded computation graph and the marked graph are, effectively, equally powerful models for parallel computation. This result is established by showing that any set of constraints represented by a computation graph can also be represented by an infinite marked graph. Then necessary and sufficient conditions are given for the existence of a finite marked graph which is equivalent to this infinite graph. It is further shown that when there is no equivalent finite marked graph, the constraints on the order of operation occurrence exhibit a "transient" behavior followed by a "steady state" behavior, each of which may be represented by a single finite marked graph.

References

[1] Karp, R.M., and Miller, R.E., Properties of a model for parallel computations: Determinacy, termination, queueing. SIAM J. Appl. Math. 14 (1966), 1390-1411.

[2] Holt, A.W., and Commoner, F., Events and conditions. Information systems theory project. Research Report of Applied Data Research, Inc., New York, 1970.

[3] Commoner, F., Holt, A.W., Even, S., and Pneuli, A., Marked directed graphs. J. Comput. Syst. Sci. 5 (1971), 511-523.

[4] Miller, R.E., A comparison of some theoretical models for parallel computation. IEEE Trans. Comput. v. C-22 (1973), 710-716.

[5] Miller, R.E., Some relationships between various models of parallelism and synchronization, IBM Research Report, RC5074, October, 1974.

[6] Meyer, S.C., An analysis of two models for parallel computation. Ph.D. Thesis, Department of Electrical Engineering, Rice University, December, 1974.

STARAN COMPLEX
DEFENSE MAPPING AGENCY
U.S. ARMY ENGINEER TOPOGRAPHIC LABORATORIES

By

Lawrence A. Gambino, Director
Computer Sciences Laboratory
U.S. Army Engineer Topographic Laboratories
Fort Belvoir, Virginia 22060

Roger L. Boulis, Sr., Development Engineer
Digital Technology Department
Goodyear Aerospace Corporation
1210 Massillon Road
Akron, Ohio 44315

Abstract: Hardware, software, and applications are described for a STARAN* associative array processor (AAP) that has been installed by the Defense Mapping Agency (DMA) and the U.S. Army Engineer Topographic Laboratories (USAETL) at the USAETL facility in Fort Belvoir, Virginia. STARAN - developed and built by Goodyear Aerospace Corporation - is now being used in a stand-alone mode, which will prevail until the end of 1975 when part of the interface to the ETL CDC 6400 sequential computer is completed. As designed, the STARAN interface to the CDC 6400 is accomplished with a command channel to the 6400 mainframe and a high bandwidth data channel connection to the CDC's extended storage system (ECS). The ETL image processing system is connected to STARAN through the CDC 6400. The fact that the CDC 6400 has the high bandwidth characteristics to match those of STARAN avoids the usual host computer data rate bottleneck that STARAN encounters; this allows a critical review of the cost-effectiveness that STARAN can provide. Applications being investigated include digital image processing, automated cartography, stero-photogrammetry, and storage and retrieval.

Introduction

The Defense Mapping Agency (DMA) and the U.S. Army Engineer Topographic Laboratories (USAETL) have installed a Goodyear Aerospace Corporation STARAN* associative array processor (AAP) at the USAETL facility in Fort Belvoir, Virginia (see Figure 1). The equipment, delivered to USAETL in October 1974, included the STARAN AAP with four arrays of associative memory, a parallel input/output (PIO) unit, a 64-track parallel head disk (PHD), and a variety of peripherals. The installation was completed and subjected to a one-month availability demonstration during November 1974. The STARAN system achieved an availability figure of 99.58 percent, which was well above the required 90 percent.

Since November 1974, STARAN has been used in a stand-alone mode. Thus, programmers have had valuable time available to develop their programs and familiarize themselves with STARAN.

*TM, Goodyear Aerospace Corporation, Akron, Ohio

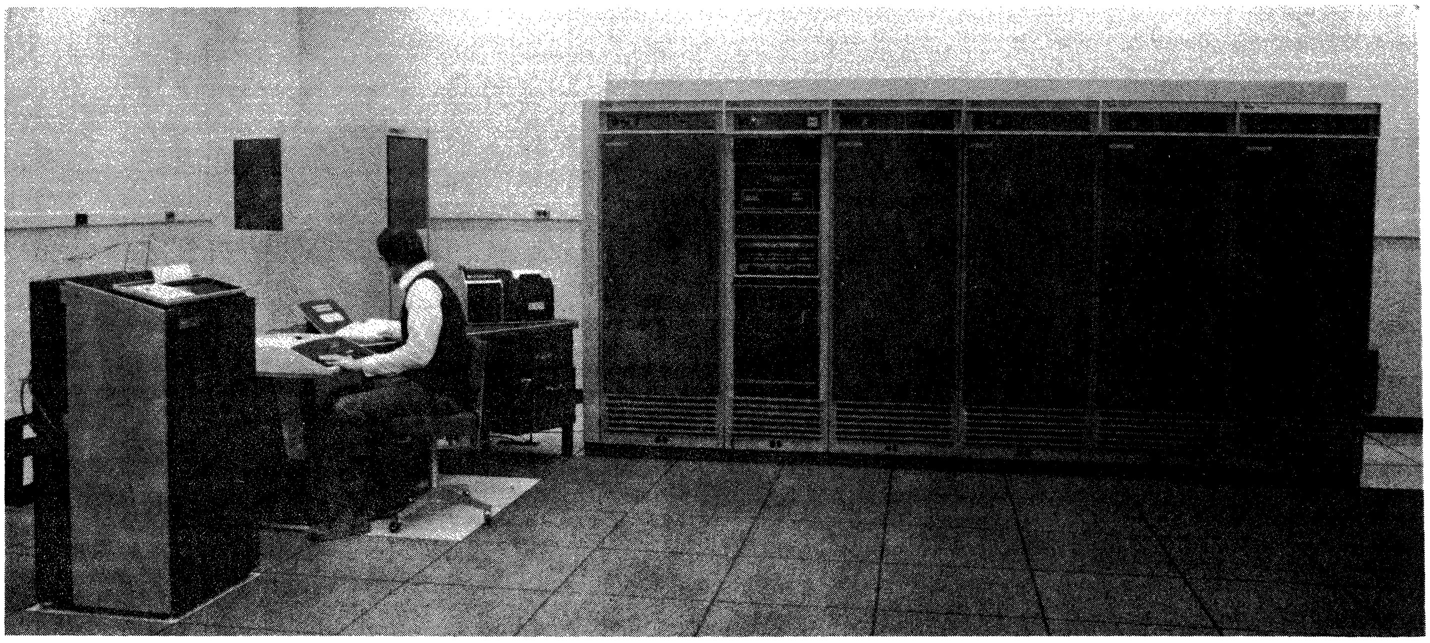


Figure 1. STARAN Associative Array Processor at DMA/ETL

The custom interface unit (CIU) between STARAN and the CDC 6400 consists of two parts. The first part, installed in late July 1975, is a command channel interface unit (CCIU) that is capable of transferring data as well as command information. The second part, which is planned to be installed later, is a data channel interface unit (DCIU) that will provide an extremely high bandwidth path between the STARAN arrays and the CDC's extended core storage (ECS) memory.

An interface almost identical to the CCIU between STARAN and the CDC 6400 makes possible the attachment of special image processing equipment to the CDC 6400.

Figure 2 shows the interconnections of the DMA/ETL STARAN complex. In the remainder of this paper, the complex is discussed from the standpoints of hardware, software, and applications.

Hardware

System Description

Standard STARAN Elements. The heart of the STARAN AAP is the associative array. The array's unique multi-dimensional access capability allows search, arithmetic, and logical operations to be performed simultaneously on either all or selected words of the array. The elements that support the array's operation as well as the array itself are shown in Figure 3 and are described below.

- Sequential Controller and Memory. The STARAN sequential controller is a Digital Equipment Corporation (DEC) PDP-11 minicomputer. It performs maintenance and test functions, controls peripherals, maintains job control, provides for operator communciation between various STARAN elements, and assembles STARAN programs. Sixteen-thousand words of 16 bits per word memory are contained within the sequential controller.

- AAP Control Memory. All assembled AAP control instructions are stored in AAP control memory. The several different types of AAP control memory include magnetic core memory, solid-state page memories, solid-state control memory, and host computer control memory. Any AAP control memory can also exist as a data buffer between various STARAN elements.

- AAP Control. Data manipulation within the four associative arrays is performed by AAP control as directed by programs stored in AAP control memory. Data within each array may be operated on separately or in parallel as a function of AAP control. Two 8-bit field pointers and three 8-bit field length counters control the array address and number of bit slices to be operated on in sequence. All timing in AAP control is asynchronous and directly controls the operations performed in the arrays.

- Associative Arrays. Each of the four arrays contains 256 by 256 bits of multidimensional access (MDA) memory. The memory is a patented organization that permits word-oriented accesses, bit-oriented accesses, and accesses with mixed orientation. In one operation, it is possible to read or write all bits of one word, one bit of all words, a few bits of many words, or many bits of a few words. Each array communicates with three 256-bit registers - M, X, and Y - through a flip (permutation) network. The M register allows masked writes into the array. Each bit of the X and Y registers forms a small processing element whose logic can perform the 16 Boolean functions of two variables.

- External Functions. The external function (EXF) logic allows any STARAN control element to transfer control information to any other STARAN element. The typical STARAN

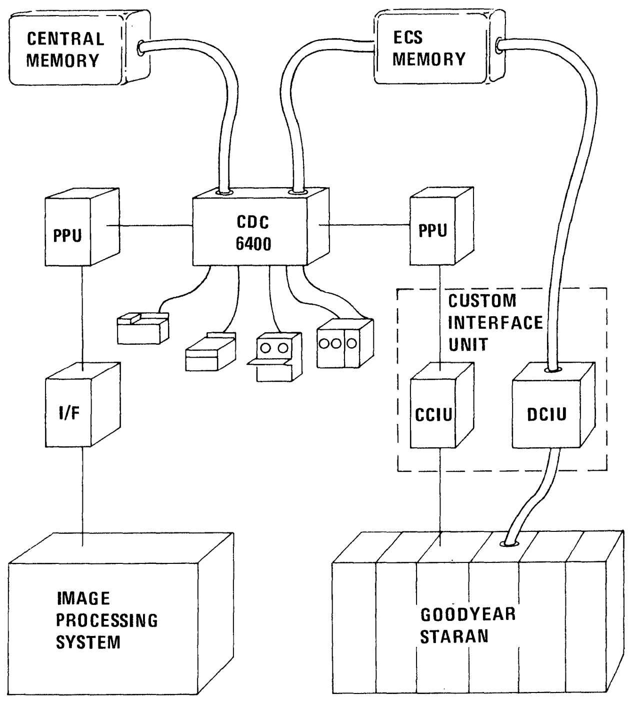


Figure 2. DMA/ETL STARAN Complex

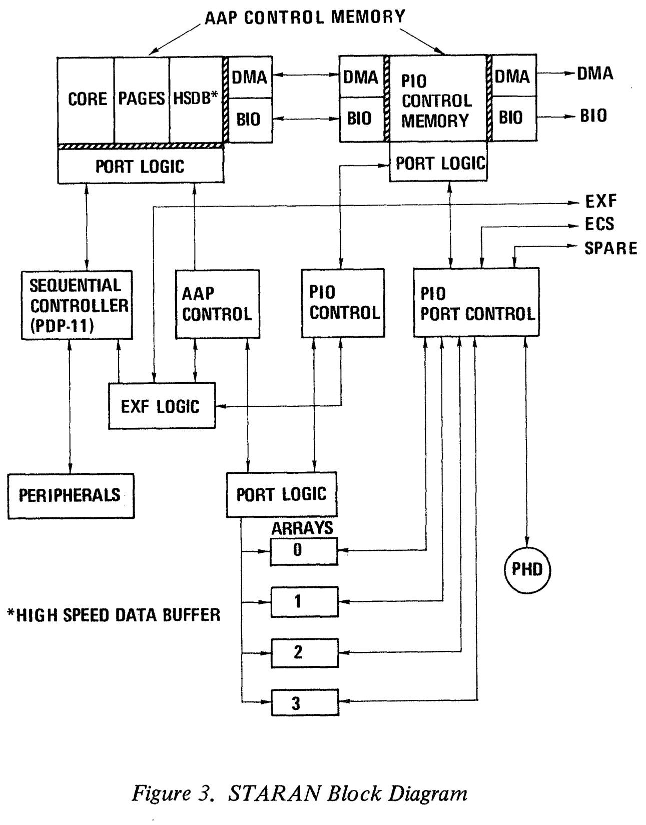


Figure 3. STARAN Block Diagram

connection to a host computer includes the EXF capability to ensure an efficient connection. Control and status sensing of EXF's are accomplished by issuing a 19-bit EXF code and receiving a one-bit sense signal in return.

Standard STARAN Input/Output. The standard peripherals delivered iwth STARAN include a cartridge disk drive and controller, line printer, card reader, paper tape reader and punch, and high-speed keyboard printer. Optional equipment includes magnetic tape transports and keyboard CRT's. All peripherals interface with STARAN's sequential controller, the PDP-11 minicomputer.

In addition to peripheral communication, STARAN provides complete facilities for interfacing with other processors. Figure 3 illustrates these features. The four input/output (I/O) buses provided include (1) direct memory access (DMA) bus, (2) buffered I/O (BIO) bus, (3) external function (EXF) bus, and (4) parallel I/O (PIO) bus.

The DMA is a 32-bit wide bus for STARAN AAP control or sequential control to address any external memory. The BIO bus is a 32-bit bus provided for processors to address STARAN's memory. The EXF bus is used to transfer control/interrupt information. The PIO bus is a special bus provided by the optional PIO cabinet. It is a very wide and fast data bus used for STARAN array I/O. Up to seven 256-bit wide ports can be connected directly to seven arrays for the fast movement of array data. Only four of these ports are utilized as array connections in the DMA/ETL facility STARAN. One of the remaining three ports is used as a 64-bit wide port to the 64-track parallel head disk (PHD). In the near future, the sixth port will be dedicated to a data channel connection between STARAN associative arrays and the CDC 6400 ECS. This will be 240-bit wide, high bandwidth data channel.

STARAN/CDC 6400 Interface. The STARAN-to-CDC 6400 custom interface unit (CIU) will allow programs executing in the two machines to transfer commands and data to each other. The CIU design, shown in Figure 4, consists of:

• A command channel interface unit (CCIU), which provides a command and control interface path between the

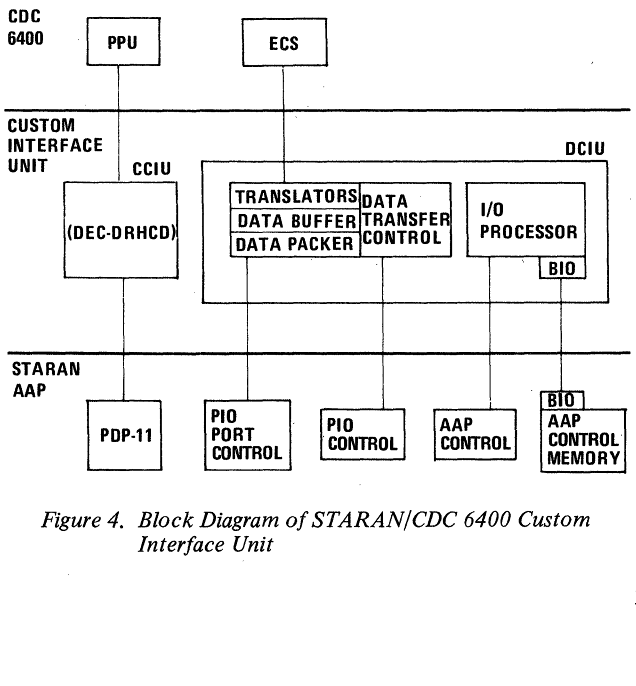


Figure 4. Block Diagram of STARAN/CDC 6400 Custom Interface Unit

STARAN PDP-11 sequential controller and the peripheral processing unit (PPU) of the CDC 6400. It is off-the-shelf, self-contained equipment that ties directly into STARAN's PDP-11 unibus and CDC's PPU channel without modification of either.

• A data channel interface unit (DCIU), which provides a high transfer rate data path connecting one port of STARAN's PIO module to the CDC 6400's ECS controller.

The CCIU serves as an interprocessor buffer between STARAN and the CDC 6400. Because of the nature of the CDC I/O system, the CCIU responds to requests from the CDC PPU and, therefore, is considered passive by the CDC system. The CCIU connects directly to the 12-bit data channel of the PPU. Connection to the STARAN PDP-11 is via the unibus, a 16-bit channel. The CCIU contains a buffer to match the data transfer rates of the two processors. All required packing and unpacking required by the processors for a maximum data transfer rate is done by the CCIU hardware.

The DCIU appears to the CDC 6400 channel as a standard computer coupler. Requests for access to the ECS are initiated by the DCIU in accordance with transfer parameters supplied by STARAN. Data are transferred as 480-bit super words (SWORDS) multiplexed over a 60-bit wide channel on the CDC side of the DCIU. On the PIO side of the DCIU, 240-bit segments are buffered into the 256 bit wide PIO port connection. Packing and unpacking of the 480-bit SWORD at the DCIU are accomplished by the hardware.

Interactive Digital Image Processing System. Image processing systems, in the past, have been special purpose analog or optical-electronic hardware. Most of these devices are designed to handle only one application. Some can handle several applications. System requirements can change - new applications can arise. When this happens, expensive equipment becomes obsolete. At best, costly, time consuming hardware changes can marginally upgrade the system.

Digital image processing, on the other hand, offers flexibility as one of its main advantages. Once the basic hardware and software systems are developed, only minimal software effort is required to apply the system to new applications as they arise. Further, the system can be used to simulate and analyze the feasibility of special purpose hardware for various tasks before any hardware is built.

Data organization and the ability to perform complex operations with minimized information losses are other attributes contributing significantly to its popularity. For instance, the digital system can access, sort, and collect data from an entire imagery data base using some common element as the search basis. Moreover, any complex operation for which a definitive algorithm exists can be performed on that data base. The information losses usually experienced when processing imagery data can be minimized and controlled if necessary.

However, digital image processing has its drawbacks. Massive quantities of data play the burdening role. Large images result in very large quantities of data. These data must be stored and then transferred to the processing element(s). From the processing elements, the data must be stored one more time before being displayed or made available as hard copy. Therefore, when considering an interactive digital image processing system, transfer rates and processing times must be considered in detail.

Measures are under way at the DMA/ETL STARAN complex to resolve the problems posed by digital image processing. Very large quantities of data are being stored in the CDC 6400 ECS memory. STARAN will be used as a special processing element to reduce processing times. Transfer times between STARAN and the CDC 6400 will be greatly reduced when the DCIU connects STARAN's associative arrays to the CDC's ECS. Finally, the on-line digital image processing facility will provide the interactive capability, which is an extremely important aspect of image processing. With the DMA/ETL STARAN complex, the pattern recognition capabilities of the human being can be married more closely to the computational power of the associative and sequential digital computer.

Images from the interactive digital image processing system (Figure 5) are input to the system from magnetic tape or from digitized transparencies made on the DICOMED D56. Once images are in the system, they are stored on CDC's 844 disks. The CDC 6400 will contain the majority of the processing software. STARAN will eventually assume the majority of the processing load as the software is developed. The interactive capability results from an operator working through the Tektronix 4014-1 alphanumeric console and the COMTAL image display unit. Images and optional functions are presented to the operator, who specifies the function he wants performed. The CDC 6400 then reads the image from disk, initiates those functions to be done in STARAN, performs those functions to be done internally, and sends the resulting image to the COMTAL for inspection. Hard copies, prints or transparencies of the processed imagery data are available on-line by the DICOMED D47 or off-line by copying to magnetic tape for recording on other hard copy devices.

Performance Figures

General. STARAN is a special breed of computer - an associative processor. Because its architecture is unique to that of any sequential processor, presenting summarized performance characteristics might be misleading. Performance can best be judged when STARAN is compared directly with another computer for cost-effectiveness in algorithm execution. However, care must be taken when using an algorithm as a tool for comparison. Almost always, the algorithm must be recreated to take full advantage of STARAN's associative architecture. A specific example of this performance comparison technique is presented later in the paper. As a start, some baseline performance figures and detailed characteristics of STARAN's major elements are presented below.

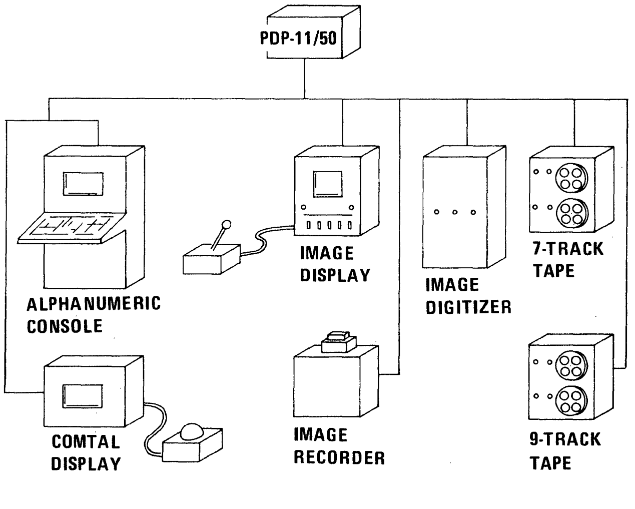


Figure 5. Digital Image Processing System

Associative Array. Any STARAN system can have up to thirty-two associative arrays. Typical array operations with their corresponding execution speeds are:

Search	150 nsec/bit slice
Add or subtract	800 nsec/bit slice
Read (multidimensional, 256 bits)	150 nsec
Write (multidimensional, 256 bits)	300 nsec

Control Memory. The largest portion of control memory within STARAN is one microsecond per cycle core memory. Currently there are 16K words (32 bits/word) that will be expanded to its design maximum of 32K words by Fall 1975.

There are two types of solid-state control memory. The first type operates at 150 nanoseconds per cycle and is referred to as "page" memory. Three, 512-word, 32 bits/word memories make up this page memory. While instructions are executing out of any one page simultaneously, new instructions can be loaded into one of the other two. Page memory may be doubled in size, if desired.

The second type of solid state memory operates at 350 nanoseconds per cycle and does not offer the paging feature. It consists of two 512-word by 32-bit memories of STARAN control memory. One is referred to as the high speed data buffer, the other as the PIO control memory.

Parallel Input/Output. The parallel I/O module was designed to provide very high bandwidth data paths to and from each associative array. In fact, the PIO module at ETL is capable of handling an inter-array data transfer of up to 1024 bits in approximately one microsecond. The transfer involves only four 256-bit wide ports out of the possible seven to which arrays could be attached. Of the three remaining ports at ETL, one is a 64-bit wide connection to the PHD, one is reserved for a 240-bit wide connection to CDC's ECS, and one is a spare.

In addition to this high bandwidth capability, PIO control is also capable of processing within one set associative arrays while mainframe control is operating within a different set of arrays. During this mode of operation, both PIO and mainframe control are operating autonomously.

STARAN/CDC 6400 Interface. As with any computer-to-computer interface, rapid command and data transfer are of extreme importance. It was with this thought in mind, that the STARAN-to-CDC 6400 DCIU design was undertaken. The 6400's ECS provides the ideal medium for high bandwidth data transfers for fast on-line digital processing of data by STARAN.

The ECS memory can consist of up to four banks of 125K, 60-bit per word memory. Each bank can be accessed at a 150-megabit per second rate and linearly adds to the data transfer capability of the channel (see Figure 6). The maximum data rate capability of the DCIU is 300 megabits per second. Therefore, beyond two banks of ECS, the bandwidth of the data channel cannot be increased. ETL currently has only one bank of ECS.

Since the DCIU is not very well suited to transferring commands efficiently, the CCIU provides the necessary direct link between programs resident in STARAN's sequential controller (PDP-11) and those in one of CDC's eight PPU's. The hardware proper is a DEC DRHCD channel coupler. The minimum effective transfer rate that will be achieved with the DEC DRHCD hardware and the associated GAC software is 4.0 megabits per second. The maximum to be expected is 4.8 megabits per second. These rates are achieved by performing DEC "TRAN" type direct data transfers between the CDC 6400's PPU buffer and STARAN's bulk core control memory. Block transfers are limited to a 4K word by 32-bit block. Once begun, the transfer is controlled strictly by hardware and continues at 4.0 to 4.8 megabits per second until completion of the block.

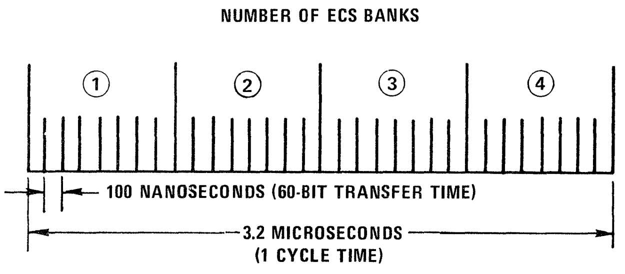


Figure 6. CDC 6400 ECS Memory Timing

Physical Description

All major elements of STARAN are built using proven packaging techniques and modular design. These philosophies are backed up by almost exclusive use of standard, commercially available, integrated circuits (IC's), regulated power supplies, laminated power buses, cabling material, and electronic enclosures. The majority of the IC's are Motorola's Emitter Coupled Logic (MECL) II and MECL 10,000 series IC's. They are mounted on multilayer printed circuit boards designed and built at Goodyear. Goodyear's first multilayered boards were built eleven years ago. From that point on, GAC has benefitted from carefully controlled board impedances, properly terminated transmission lines, and readily available replacement components.

Interprinted circuit board communication takes place over multilayered backpanels and twisted pair wires with carefully controlled impedances. Intercabinet communication is via twisted pair cables contained within the physical confines of STARAN.

Each STARAN cabinet is air-cooled, requires standard 220 3-wire, single phase power, and is automatically sequenced up or down depending on various self-portect features. These include automatic sensing of undervoltage, under-airflow, overtemperature, and operator intention conditions. Total power consumption is approximately 12 kilowatts. Each STARAN cabinet can contain seven "nests" each containing 34 printed circuit boards. In the DMA/ETL STARAN, there are a total of 612 printed circuit boards representing approximately 24,700 IC's.

As shown in the photo of Figure 1, the STARAN system at ETL consists of six cabinets (described as follows, from left to right):

- Cabinet 1, PHD control, contains the 64-track PHD and has provisions for accepting a second system disk drive scheduled for the Fall of 1975. The controller for the STARAN/CDC 6400 CCIU also resides there.
- Cabinet 2, sequential control, contains the PDP-11 minicomputer, system disk unit, paper tape reader/punch, and one-microsecond core memory.
- Cabinet 3, AP control, provides all the necessary control signals for the associative arrays.
- Cabinets 4 and 5, array, contain the four associative arrays (two in each cabinet). Provisions are made to add an extra array in each cabinet if so required.
- Cabinet 6, PIO, is the most heavily populated cabinet. Almost every circuit board slot of every nest is occupied.

System Reliability. Since STARAN is relatively new, it is important to publish the field performance figures that have accumulated to date. Including the STARAN installation at the DMA/ETL facility, the STARAN mainframe hardware has now accumulated about 20,000 hours of field operation. It has demonstrated an average availability of 99.8-percent (MTBF of 702.5 hours and MTTR of 1.2 hours). At the system level, which includes the peripherals (disks, line printers, paper tape readers/punches, card readers, and keyboard terminals), the average availability is 99.3 percent (MTBF of 288.4 hours and MTTR of 1.95 hours). The MTBF figure represents the average "machine" time that has accumulated between each malfunction of the equipment. The MTTR figure represents three time elements: (1) the time to diagnose the problem, (2) the time to repair the problem, and (3) the time to verify that the problem has been corrected. Availability is defined as MTBF/(MTBF + MTTR).

Software

STARAN Software

Disk Operating System (DOS). STARAN's system software is based on a disk operating system (DOS) that provides ready access to system programs, a device independent I/O, and a file system. Operation of STARAN can be under direct control of the console user, or it can run in a batch mode with a command stream from any character input device, such as the card reader or the system disk. The disk is a file-structured bulk storage device in which all system software is resident for rapid user access.

STARAN Assembly Language (APPLE). A complete set of stand-alone system programs are provided to enable STARAN users to generate programs. The software used to convert source language programs into executable machine language programs includes a macropreprocessor (MAPPLE), assembly language (APPLE), and a relocating linker. All programs for STARAN are written in APPLE.

The mnemonics of APPLE generate from 1 to 12, in-line machine language instructions. About one-third of all available mnemonics generate calls to subroutines. Consequently these mnemonics do not generate in-line code. The one-to-many mnemonics generally implement a parallel algroithm for arithmetic or search operations using the arrays. Thus, APPLE is at a higher level than sequential machine assembly languages.

MAPPLE increases the user's flexibility at assembly time. It provides a large set of arithmetic, logical, relational, and string manipulation operators. User benefits include the ability to define new mnemonics, redefine existing mnemonics, and generate standard instruction sequences.

The STARAN linker generates an absolute load module from a relocatable object module. Multiple object modules may be input to the linker since it has the capability of resolving symbols defined across object module boundaries as well as adjusting addresses for relocation.

STARAN Execution Control Aids. Execution control software covers loading, executing, and debugging programs on STARAN. The "loader" moves absolute load modules into STARAN control memory, beginning at a specified address. Overlay modules can be brought in dynamically through program calls from an executing program.

The STARAN program supervisor (SPS) is the software interface between the associative and sequential control portion of STARAN. This module provides services for system users when programming in APPLE and when programming a PDP-11 routine to interact with an APPLE program.

The STARAN debug module (SDM) helps the user debug APPLE programs by giving him control of the exeution of the program being debugged and access to memory and registers. Single step, trace, breakpoint, and memory dump features provide good execution control.

The STARAN control module (SCM) is the interface between the user and execution of a STARAN program. STARAN control commands, such as start and halt, are made available to the user by this module.

STARAN Diagnostics. Diagnostics round out the available system software for STARAN, providing an off-line trouble-shooting aid that is capable of pinpointing faults in the hardware to the module, the group of printed circuit boards, or even the single printed circuit board level. The diagnostic package is organized so that minimum user intervention is required. Each diagnostic test is made into a subroutine called by the executive controller. Each routine uses common error handling routines in the executive to limit redundant programming and conserve memory. The executive also intercepts all stray interrupts and reinitializes the machine before calling the next diagnostic program. The diagnostics run under the disk operating system environment, so that all DOS functions are available to the diagnostics.

STARAN/CDC Interface Software

Command Channel. As of this writing, the interface software development has just begun. Scheduled acceptance for the interface software is January 1976. Software control of the channel will involve system programs in STARAN's PDP-11 and in both the CPU and PPU of the CCD 6400. The PDP-11 program follows DOS conventions for interface sofware; since the program enables communication with the CDC 6400, it is called the CDC 6400 device driver (see Figure 7). In the 6400

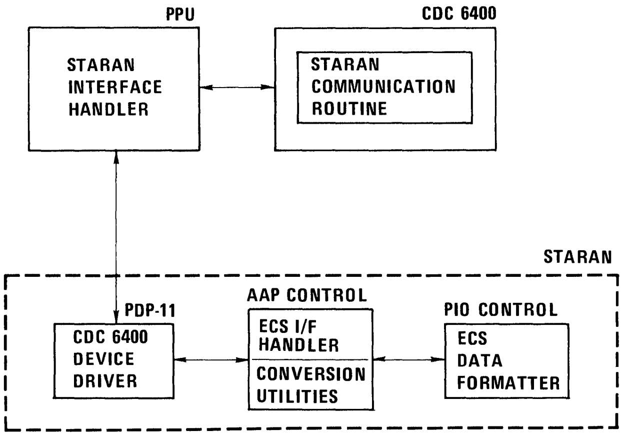


Figure 7. Block Diagram of STARAN/CDC Interface Software

system, the PPU program will handle the interface hardware and the application program I/O requests; it is called the STARAN interface handler. The CPU program will control command stream and multiple user requests for access to STARAN; it is called the STARAN communication routine.

From a software viewpoint, the command channel will appear to be three logical devices:

- The first device will transfer character information from the CDC 6400 to the PDP-11. The characters will be treated as the batch command stream input for batch DOS in the PDP-11. This device will be like a card reader to the DOS.
- The second logical device will transfer the job log output produced by batch DOS from the PDP-11 to the 6400. This device will be like a printer to the DOS.
- The third logical device will transfer data between STARAN bulk core and CDC central memory. It will be like a disk in that the user program may issue read and write requests to the device, although it will not include a file structure.

The CDC 6400 device driver will be written in PDP-11 assembly language. Implementation of the logical devices mentioned above will be by several software pseudo channels. The general scheme will reserve one pseudo channel for command stream input, one for job log output, and several for data I/O. Conversion of character data from CDC 6400 display code to PDP-11 ASCII will take place on all command stream input. The opposite conversion will be implemented for job log output.

The device driver will include an interrupt handler to allow 6400 users to quit processing. This routine will halt STARAN, initialize it, and flush the current job in the batch stream. The pseudo channels for data I/O will move data between user buffers in STARAN control memory and CDC 6400 central memory. Data will be stored in control memory such that eight CDC central memory 60-bit words (480 bits) are packed into 15 STARAN 32-bit words (480 bits). For transfers that are not a multiple of 480 bits, the final word being written will be padded with zeros. Several buffers may be defined in each memory by using several pseudo channels. Thus, the user may read from one buffer and write into a different buffer with both channels open simultaneously.

The STARAN interface handler and communication routines will be the CDC 6400 part of the command channel interface software. The interface handler will be written in PPU Compass assembly language for execution in a PPU. The communication routine will be written in CPU Compass for execution in the CPU. It is expected that user programs in STARAN and the CDC 6400 will cooperate in issuing I/O requests. When one processor issues a read request, the other must issue a write; that is, the programs issue complementary requests so the system does not reach deadlock with both programs waiting to complete I/O operations. To reduce the chance of deadlock, the PPU program will be written with multiple queues, corresponding to the multiple pseudo channels in the 6400 device driver. This will permit, for example, a 6400 program to issue a write on channel 1 and read on channel 2. Then the STARAN program can, in arbitrary order, issue the complementary read on channel 1 and write on channel 2. The first STARAN request issued will be satisfied first.

The STARAN communication routine will provide a means for a single 6400 user to gain control of STARAN and keep other users from accessing it until control is relinquished. It will notify the PPU of user I/O requests and will contribute to quit function processing. Batch commands to STARAN will be issued by this communication routine. Interactive jobs will be allowed one of three options:

- If STARAN is available, the interactive routine will be allowed to "attach."
- If STARAN is not available, a busy indication will be returned to the calling program. This gives the interactive user the option of branching to an alternate procedure which requires only the CDC 6400.
- If STARAN is not available, the interactive user will be allowed to place his job (command stream) in an interactive queue that will have priority over the "normal" batch job queue.

As long as a command stream file exists, the communication routine will issue write requests to the command stream channel. The batch processor in STARAN will issue read requests as it needs commands. Similarly, the job channel will have read requests pending continuously by this routine. Since the STARAN interface handler in the PPU has a queue for both of these channels, no conflicts on I/O will result.

The STARAN interface handler will be written to support the three logical devices that the CDC 6400 device driver supports. The batch command stream will be a character mode, write-only pseudo channel. Data I/O will use several pseudo channels. The quit function will be implemented.

Data Channel. This channel is intended as a high bandwidth data path between STARAN arrays and 6400 ECS. The hardware for this interface will exist between the PIO unit in STARAN and ECS in the CDC 6400. A path for interface control information is provided between the interface hardware and STARAN mainframe.

Software control of the channel involves system programs in the STARAN mainframe and PIO unit. Since the interface can directly read or write ECS, it will not be necessary to develop software for exeuction in the 6400.

There is no change in the way CDC 6400 programs access the ECS. In STARAN, mnemonics will be macros that use interface software routines in both the mainframe and PIO processors.

The primary functions of the mainframe routine will be to (1) provide control information to the interface hardware, (2) return status information to the user program, and (3) provide array formatting parameters to the PIO routines. The mainframe routine is called the ECS interface handler. The primary function of the PIO routine will be to move data between PIO buffers and the associative arrays according to formatting parameters provided by the mainframe routine. The PIO routine is called the ECS data formatter.

Applications

General

Prior to installing STARAN at ETL, various DMA applications considered amenable to STARAN processing were studied under contract to Goodyear Aerospace. The applications were listed, and the sequential computer formulations were submitted to Goodyear. In cooperation with ETL, Goodyear personnel familiarized themselves with these formulations. Purposely omitted from this list were large scale data reduction problems that require double precision arithmetic on most sequential machines, such as the UNIVAC 1108's at DMA production centers. For example, a class of DMA problems dealing with geodetic satellite reduction schemes was among that type. These require double precision arithmetic for the numerical integration of orbits and for the inversion of very large matrices in order to obtain required accuracies in orbit reconstruction and tracking station coordinates.

In many instances, however, it is not the large scale data reduction problem that consumes large amounts of sequential computer time. Instead, the simple arithmetic problem involving repetitive operations on massive amounts of data usually consumes even more computer time. This is characteristic of the problem under investigation.

The applications being researched for the DMA fall generally into four functional areas: (1) digital stereo-photogrammetry, (2) automated cartography, (3) digital image processing, and (4) storage and retrieval. Not every aspect of these areas is being researched. The following discussion describes what is being researched in each of these functional areas and explains why they were chosen as candidates for STARAN implementation.

Stereo-Photogrammetry

Photogrammetry involves many processes in which the ultimate objective is to extract precise, three-dimensional information from stereo-pairs of photographs. This involves the process of correlation. But prior to this, the photographs must be measured and triangulated to reconstruct the geometry of the photographs at the instant they were exposed in the aerial mapping camera. The process of mensuration involves the mathematical elimination of distortions, such as lens distortion and film shrinkage. All measurements are exacting and performed on precision measuring instruments. After all photos are mensurated, the strips or blocks of photos with all their pass points (tie points between photos) are then triangulated using large scale, photogrammetric, computer programs. The geometric parameters derived from this process are used to correlate the digitized imagery from the overlapping stereo-pairs of photographs. It is only the correlation portion of the problem to which STARAN is being applied.

Stereocorrelation is categorized as a simple arithmetic, but repetitive, type problem that involves massive amounts of data. For this reason, the entire stereocompilation process at the DMA production centers has always been relegated to analog/electronic scan and correlation equipment. The large scale sequential digital computer cannot compete with these sophisticated instruments both in cost and speed. This is not to say that the process cannot be performed digitally. Digital correlation from digitized, stereophotographic data was proven feasible years ago (on the IBM 7090), but it was not practical. Electronic correlation instruments, such as DMA's UNAMACE and AS-11, automated stereocompilation instruments, were built for this job almost ten years ago and have undergone various stages of upgrading since then. They remain the work horses for the extraction of three-dimensional stereophotographic mapping data for the DMA. The only way current digital technology can be competitive is through the development of parallel/array digital architecture to perform the correlation function.

Experiments thus far at ETL have indicated that the linear correlation coefficient algorithm is the most reliable indicator of correlation for all types of digitized, stereo-scenes; that is, urban scenes, drainage scenes, mountainous scenes, etc. However, a much simpler algorithm, such as the absolute difference between scenes, works well with certain scenes. The algorithm is simpler in the sense that not as many arithmetic operations are required to evaluate it. This situation might result in the implementation of both algorithms in a digital stereocompilation system where the simpler algorithm would be used until it begins to break down whereupon a switch is made to the more sophisticated algorithm.

The linear correlation coefficient algorithm is as follows:

$$RXY = \frac{\Sigma\Sigma(X - \overline{X})(Y - \overline{Y})}{\sqrt{\Sigma(X - \overline{X})^2 \; \Sigma(Y - \overline{Y})^2}}$$

where X and Y pertain to arrays of pixels, one from each photo, whereby each pixel is represented by six to eight bits. Additional information can be obtained from Reference [1]. The equation is not implemented on STARAN, or even a sequential computer, as it is written here. There are computational overlaps that are exploited on STARAN to even a greater degree than on a sequential machine, and several scenes can be correlated simultaneously in each of the four arrays. Suffice it to say that at any given time during correlation only a very small portion of entire digitized stereoscene is being processed. There may be hundreds of millions of bits representing the digitized stereo area of two overlapping 9 by 9 inch aerial photographs.

Of six different algorithms investigated, the above algorithm offers the greater reliability over all types of scenes. Unfortunately, it is also the most time consuming to evaluate. The absolute difference algorithm works well with certain scenes, and it might be implemented together with the correlation coefficient algorithm in a digital stereocompilation system. This could be a means of cutting down correlation times, and it also indicates a degree of flexibility in the digital approach. DMA/ETL is just now beginning to implement the algorithms on STARAN, and first indications are that timing is impressive and competitive.

This is a very brief and general description of the digital approach to the stereo-compilation process of map making. The entire digital correlation process must be couched in the formulas of analytical photogrammetry, whereby the geometry of the overlapping photographs is input data. DMA/ETL hopes to exploit the flexibilities afforded by digital techniques to produce highly accurate mapping data along with the resulting statistical byproducts which, heretofore, have been obtained only after the fact. Also, DMA/ETL hopes that quality control may be more easily implemented in a digital environment as opposed to the current analog/electronic production approach to the problem of stereo-compilation.

Automated Cartography

Generally speaking, the process of making a map follows the stereo-compilation process from which an orthophotograph is produced. This is a photographic image of the stereo area in which all parallax has been removed and adjusted for other distortions due to the camera lens and atmosphere. This orthophotograph forms the basis for tracing required data to produce the map.

Modern techniques of obtaining numerical data while tracing have been implemented at all DMA production centers. After tracing, say the road overlay, the numerical data is usually run-length coded and then it must be thinned to cartographic standards. Each type of feature one views on a military map has a standard to which it complies. Generally, then, the process for converting an orthophotograph to a map sheet is as follows:

- Tracing, resulting in an overlay (transparency) per each feature class.
- Digitization of each overlay.
- Data processing.
- Color separation.
- Printing five-color map.

Handling the digitized data in a sequential computer to conform it to cartographic standards is a time consuming process. Looking at only one function, it takes hours to perform line thinning operations on digitized data. The objective of line thinning is to reduce a line to a single-cell thickness along its centerline. In general, lines are thinned by removing edge cells until unit thickness is attained. On sequential machines, each cell is tested with respect to the status of its eight neighbors. A cell will be removed if one of the orthogonal neighbors is zero, and removal of the cell does not destroy continuity of a line. In contrast, the STARAN approach to line thinning utilizes the digitized overlay data stored in the arrays as a binary image. Simple logic operations are used between each 256-bit of data and the data slice on either side of it. Details of this operation can be obtained from reference [2].

Table I, from Reference [2], gives the reader a quick overview of the cartographic functions and their timing estimates relative to an IBM 360/40 computer. Since this table was produced, verification of STARAN timing has been derived. The estimated times closely approximate the actual time and this looks to be one of the most promising areas of STARAN applications for the DMA. It is also one of the most significant steps in the production process.

One other point concerning the IBM 360/40. The only reason for comparing against this machine is that it was the only one upon which these cartographic functions were implemented at the time. It represented a significant step forward in the automation of these functions, and these timing comparisons should not detract from that effort.

TABLE I

TIMING RESULTS FOR RASTER PROCESSING STUDY*

Operation	No. of over-lays	IBM 360/40 (actuals)	STARAN S-1000 (est)
Registration mark detection	8	7 min	0.084 sec
Line separation	1	?	21.25 sec
Character recognition	1	1.0 hr	1.25 sec
Line thinning and (vectorization)	7	4.2 hr	12.5 sec
Skew correction	8	1.4 hr	0.25 sec
Line break detection	7	1.4 hr	6.25 sec
Line smoothing	7	1.4 hr	
Line symbol generation	7	?	5.4 sec
TOTALS		9.5 hr	51.98 sec+

*12 by 16 in. map area with 8 overlays and 4-mil resolution.

+Requires approximately 5.5 min of tape I/O.

Digital Image Processing

Investigations in digital image processing have been going on at ETL for several years. One of the milestones in this endeavor was the implementation of a user oriented image processing system on the CDC 6600 computer. The system is called DIMES, an acrononym for digital image manipulation and enhancement system, and represents an evolution of these type systems over the last five years. It has been in use at ETL for almost two years, and it has been put to many useful applications. Since then, however, DMA/ETL has acquired a CDC 6400 computer, the STARAN, and digital image processing equipment such as the DICOMED scanner/hardcopy/display/tape equipment, COMTAL color/black and white CRT display, PDP-11/50 minicomputer and TEKTRONIX alphanumeric/graphics terminal. These have been described in the hardware section of this paper.

All of these components are currently in the process of being interfaced to provide a comprehensive, interactive digital image processing system. The CDC 6400 computer is equippped with eight peripheral processing units. One of them will be dedicated to the STARAN and another to the image processing system. Two special pieces of hardware have been built to interface the STARAN to the 6400 and the PDP-11/50 to the 6400. Software is currently being written for the systems. Therefore, compared to the DIMES, batch oriented system, a much more sophisticated system is being developed which will provide fast response to user requests for processing digitized grey shade data from aerial photographs and other imagery.

The system is being set up partially in response to our Digital Photogrammetry functional area where we expect to use it as a test bed for the digital stereocompilation operation mentioned earlier.

There is no need to reiterate the potential of STARAN in that area, but the system will also serve as an interactive measuring device where again STARAN will be used to correlate measurements of pass points on stereopairs of photographs.

There are numerous image processing functions that will be implemented on this system, and STARAN will play a significant role in the operation. This includes such functions as the Fast Fourier transform and digital filtering of imagery in the frequency domain. Other functions of interest to DMA will include interactive digital rectification of aerial imagery. This again is a production operation relegated to electro-optical and analog devices. In order for digital techniques to be practical, they must be performed on parallel /array processors. Again, the sequential computer can perform this operation but not competitively.

The foregoing represents a very brief overview of the DMA/ETL digital image processing facility. It is a research facility in which specific digital mapping functions will be investigated. To date, the DMA/ETL digital image processing efforts have been devoted to developing a system in support of other functions such as digital photogrammetry. However, many of the original DIMES modules will be implemented on this system and those which are more advantageous to array processing will be implemented on STARAN.

Storage and Retrieval

With respect to storage and retrieval, ETL has received from the DMA/Aerospace Center, St. Louis, Mo., several data tapes containing a data base of terrain information. This data base is currently implemented on the UNIVAC 1108 system at this Center. Extensive use will be made of the small parallel head disk (PHD), which is part of the DMA/ETL STARAN facility. The characteristics of this disk are discussed in the hardware section.

Not too much can be said of this function at this time since the entire DMA/ETL facility is currently being integrated. After this is accomplished, input and output to the disk, tape, and printer systems of the CDC 6400 will be available to the STARAN user. In the meantime, the only means of input and output are via the DEC card reader and printer currently on-line with STARAN. However, small amounts of data have already been passed between the arrays and the PHD and searches are being made in the arrays for prescribed data.

Summary

A Goodyear Aerospace STARAN AAP has been installed at the DMA/ETL facility, Fort Belvoir, Virginia. It is now being used in a stand-alone mode. This will continue until late 1975 when the command channel interface unit hardware and software interfaces are completed between STARAN and the host CDC 6400 computer. The data channel interface unit hardware and software interfaces are planned to begin next year.

In addition, work has been completed on a hardware interface between the CDC 6400 and the ETL digital image processing system. Work is under way on the software interface.

From the standpoint of applications, very specific functions have been chosen for investigation at the DMA/ETL STARAN complex. Generally speaking, these applications do not require large matrix inversions or double precision arithmetic. It was felt that if undue efforts were devoted to these problems, STARAN and array/parallel processing in general would have a difficult time in showing production benefits compared to current approaches at the DMA production centers. In most cases, then, fixed point arithmetic and short word lengths characterize the chosen applications.

The cartographic functions are ideally suited to array processing. The STARAN algorithms are radically different from those implemented on the sequential machine and STARAN may lead to an early payoff, so-to-speak, in terms of producing maps.

The digital photogrammetry application involves an even more fundamental problem than that associated with cartography. In cartography, the sequential computer was the automating device since the advent of automated cartography. On the other hand, the heart of the stereo-compilation process (namely, correlation) has always been relegated to electronic/analog devices since it was generally accepted that the sequential digital computer could not compete with these devices in a production environment. For this reason attention has been focused on parallel/array processors, and, in particular, STARAN. It offers an opportunity to be competitive with existing stereo-compilation devices in addition to offering flexibility that should follow with digital techniques.

Digital image processing is being driven somewhat by the other functions. Once the STARAN, the CDC 6400, and the digital image processing system are integrated, they will offer the researcher a tremendous amount of flexibility in carrying out his work in image processing as well as photogrammetry, cartography, and retrieval of information. We have yet to determine all of the image processing functions that are going to impact DMA production, but it will become increasingly important as time progresses.

References

[1] U.S. Army Engineer Topographic Laboratories, Single Photo Analysis of Sampled Aerial Imagery, Research Note ETL-RN-74-10, Fort Belvoir, Va. (August 1974).

[2] Goodyear Aerospace Corporation, Associative Array Processing for Topographic Data Reduction, USAETL Contract DAAK02-73-C-0336, Akron, Ohio (March 1974).

[3] K. E. Batcher, "STARAN Parallel Processor System Hardware," 1974 National Computer Conference, AFIPS Conf. Proc., Vol. 43, pp. 405-410.

[4] E. W. Davis, "STARAN Parallel Processor System Software," 1974 National Computer Conference, AFIPS Conf. Proc., Vol. 43, pp. 17-22.

[5] J. D. Feldman and L. C. Fulmer, "RADCAP - An Operational Parallel Processing Facility," 1974 National Computer Conference, AFIPS Conf. Proc., Vol. 43, pp 7-15.

[6] Goodyear Aerospace Corporation, STARAN Reference Manual, GER-15636A, Akron, Ohio (September 1974).

[7] Goodyear Aerospace Corporation, Proposal for STARAN S-1000P-CDC 6400 Custom Interface Unit, GAP-75-6868-S10, Akron, Ohio, (February 1975).

[8] Goodyear Aerospace Corporation, Proposal for ETL Associative Processing Facility, GAP-74-6868, Akron, Ohio (March 1974).

A PARALLEL APPROACH TO HIGH PRF DOPPLER RADAR SIGNAL PROCESSING

Frederick E. Schenfele
Boeing Computer Services, Inc.
Space and Military Applications Division
P.O. Box 24346
Seattle, Washington 98124

Abstract -- A conceptual software design for an Associative Processor to be used as a High Pulse Rate Doppler Radar Signal Processor is presented.

A brief discussion is provided of how ambiguous range is converted to unambiguous range by the use of the Chinese Remainder Theorem. Following this, Radar Signal Processing functions are analyzed at the block diagram level.

A conceptual design is presented of an AP Radar Signal Processor and the technique used to perform the tasks of simultaneous correlation, range resolution, beam splitting, destraddle and report generation in parallel.

Additional tasks that a radar designer can perform with the computing power and array storage capabilities of an Associative Processor are also discussed.

Introduction

High Pulse Rate Frequency (PRF) Doppler Radars have data rates between 10 and 50 million bits per second, making signal processing in realtime by conventional hardware and software difficult. Each signal processing task is currently performed by a unique piece of specially designed hardware, thus dimming hopes that future radars will be of modular construction. Additionally, in a dense air environment considerable processing time is wasted in range computations due to the undesirable generation of mathematical ghosts and subsequent deghosting.

Why then the interest in a High PRF Doppler Radar? The answers lie in its ability to simultaneously distinguish two or more moving targets in a cluttered surrounding, and to totally reject stationary objects. The higher the PRF, the greater is the velocity selectivity, and the greater is the ability to distinguish one or more moving targets.

This paper is intended to demonstrate the feasibility of using an Associative Processor (AP) to handle the extreme data rates obtainable from a high PRF radar receiver. A conceptual design will be provided that will demonstrate the unique advantage provided by the AP, and how the computing power of the AP enables realtime computations.

Background Design Information

The electromagnetic energy transmitted from a pulse radar is emitted in the form of short bursts of energy called pulses. The Pulse Rate Frequency (PRF) is the number of pulses emitted per second. It is desirable to have an echo of a pulse return to the radar before the next pulse is emitted, in order to avoid range ambiguities. For example, if the pulses are 500 microseconds apart and echoes are received 200 microseconds after each pulse, the echoes may be caused by a target 100 microseconds away or 600 microseconds away.

A radar operating at a pulse rate of 50,000 pulses per second will produce a new pulse every 20 microseconds. In 20 microseconds, a pulse will appear to propagate less than 2 miles before the next pulse is transmitted. Since range is time measured with respect to the last pulse transmitted, all targets, whether zero or five hundred miles away, will appear to be less than two miles away. Thus, all range measurements in a High PRF Radar are considered ambiguous.

However, if the target is examined at two or more different pulse rates, the range can be accurately computed by use of the Chinese Remainder Theorem (CRT). Typically, a target would be monitored at a constant pulse rate that would last 3 to 10 milliseconds. This would be followed immediately by one or more (3 to 10 millisecond) examinations at another unique PRF. These examinations at a constant PRF will henceforth be referred to as a "scan."

Figure 1 depicts a transmitter/receiver at A and a target at B. The distance AB is measured at PRF1 and again at PRF2 (3 to 10 milliseconds later). The returning echoes will indicate the following equation:

$$AB = 10(D1) + \Delta D1 = 8(D2) + \Delta D2 \qquad (1)$$

If the distance of the target were unknown in the above example, the numbers 10 and 8 could be replaced by integers J1 and J2. This would result in the following equation:

$$AB = J1(D1) + \Delta D1 = J2(D2) + \Delta D2 \qquad (2)$$

By use of the Chinese Remainder Theorem (CRT), the distance AB can be computed in as few as four multiplies and one divide. The actual values of J1 and J2 need never be computed. Equation 2 forms the basis for the development of the CRT. Reference [4] provides a background for CRT derivation. To use the CRT, D1 and D2 must be in units of measure, where D1 might be equal to 33 range gates in time, and one range gate might be 0.61 microseconds wide. A time of 0.61 microseconds represents 300 ft. of wave propagation. In this example, D1 would therefore represent a distance of $\cong$1.6 nautical miles (33 x 300 ft.) of wave propagation.

Additionally, a new pulse would be transmitted every 33 x 0.61 microseconds at this PRF. D2 might equal 38. Knowing D1 and D2, and then by measuring ΔD1 and ΔD2, the range can be computed by the use of the CRT as follows:

$$\frac{C1(D2)\Delta D1+C2(D1)\Delta D2}{D1xD2} = \frac{\text{Quotient and}}{\text{Remainder}} \qquad (3)$$

Where: C1 and C2 are constants dependent on the selection of D1 and D2. See Reference [1].

Quotient is discarded.

Remainder is the answer in units of range gates.

For a 3 scan PRF or 3 PRF system, the CRT formula becomes:

$$\frac{C1(D2xD3)\Delta D1+C2(D1xD3)\Delta D2+C3(D1xD2)\Delta D3}{D1xD2xD3} \qquad (4)$$

Ghosts

If two targets are observed at the same time in each PRF, there may be no way to identify which of the two echoes came from which target. In this case, there will be four CRT computations resulting in two true ranges and two ghost ranges. This is depicted in Figure 2, where at Pulse Rate 1, target B was observed at range gate number ΔD1 and target B' was observed at another range gate number ΔD1'.

It is frequently possible to observe one target in two adjacent range gates at the same time. This double observation is due to the size of the target or its position with respect to the range gate timing. This is called Range Gate Straddle. Straddle will produce as many ghosts as would any two correlated targets.

A pulse radar system cannot transmit and receive at the same time. A returning echo is lost and the target eclipsed if the echo coincided with a transmit pulse. In a three PRF system (operating with D1 equal to 33), on the average 3 out of every 33 targets will be eclipsed, resulting in an occasional degradation to a two PRF system.

There are two methods used to reduce ghost generation: 1) To correlate by elevation and range rate (velocity) windows prior to selecting a candidate for CRT computation, and 2) To limit the range of the radar. The following explains how and why to limit the range:

Using one PRF at a pulse rate of 50,000 pulses per second, the unambiguous range is less than 2 miles. By the use of the CRT and 2 PRF's, the unambiguous range is extended to 120 miles; 3 PRF - 6,000 miles, 4 PRF - 300,000 miles, and 5 PRF - 15,000,000 miles.

In a 3 PRF system, the maximum range is 6,000 miles. By discarding all CRT answers with a range greater than 300 miles, 19 of every 20 ghosts generated are discarded. In a four PRF system, the maximum range is 300,000 miles. By discarding all answers greater than 300 miles, 999 of 1,000 ghosts are eliminated.

In summary, the minimum acceptable number of PRF's required to compute range is three. The greater the number of target observations at various PRF, the easier is the task of ghost elimination. Probability of detection increases when four or more PRF's are used.

The Signal Processing Task

This paper demonstrates parallel correlation, parallel use of the CRT, and, in effect providing both a simultaneous parallel Azimuth to Azimuth Correlation and Range Gate Straddle.

To perform these tasks, the eight most recent contiguous scans of data under consideration will be correlated simultaneously with respect to the oldest of these eight scans.

Figure 3 illustrates ten contiguous scans of data. Each black box represents a target report ready for correlation. Each scan was observed at a slightly different azimuth due to antenna rotation. The antenna beam pattern is wide enough to permit more than three and less than nine observations of a target at these different azimuths or scans. The azimuth rotation is identified by scan numbers n-9 through n, where scan n-9 is the oldest. For clarity, let the range marked A through T be unambiguous.

As the range in Figure 3 is unambiguous, each cluster of black boxes represents one true target. The problem to be solved by correlation and range association, described later in this paper, is the identification of each black box or target report as belonging to one target. Correlation will identify each cluster of reports as having similar range rate and elevation, and the CRT range computation will identify each cluster as one target due to their range and azimuth proximity. However, as the returning range is in reality ambiguous and must be computed by a CRT, there can be no range identification unless 3 or more target reports are produced at exactly the same range.

In Figure 3, based on the proximity of the range and azimuth of each target report, there are four identifiable targets. Three of these targets are first observed in scan n-9. As n-9 is the earliest scan, correlation of the eight latest scans (to include scan n-2), is with respect to B(n-9), O(n-9) and R(n-9). Following correlation, the CRT is performed on selected reports. For target 1, the CRT will identify 4 reports as producing range B, and 3 reports as producing range C. As C is one range gate away from B, these reports are considered part of the target at range B due to range gate straddle. The reports from range D will be lost as only two reports exist at this range. Report B(n-1) is ineligible for consideration due to an azimuth extension limit of eight.

The resulting identified targets for scan n-9 is comprised of target reports as follows:

TARGET 1 = B(n-9),B(n-8),B(n-7),B(n-6), C(n-8),C(n-7),C(n-6) (5)

TARGET 2 = O(n-9),O(n-8),O(n-7),O(n-6), O(n-5),O(n-4),O(n-3),O(n-2) (6)

TARGET 3 = R(n-9),R(n-8),R(n-7),S(n-7), S(n-6),S(n-5),T(n-6),T(n-4), T(n-3) (7)

After scan n-9 has been processed, scan n-8 will be the oldest. Correlation will now be with respect to scan n-8. Most of the data at ranges B,C,O,R,S,T have been identified as belonging to targets 1, 2, and 3. No attempt is made to re-use the target data in eq (5), (6), and (7) as only redundant answers will result.

All correlations are now with respect to I(n-8) and J(n-8). These reports are correlated with all I, J, K, L and B(n-1) and O(n-1). The resulting target consists of:

TARGET 4 = I(n-8),I(n-7),I(n-4) (8)

Each chosen target report contains a string of information (input):

$$\left.\begin{array}{l}\text{1. Signal Strength of Filter}\\ \text{2. Filter Number (Range Rate)}\\ \text{3. Elevation X Signal Strength}\\ \text{4. Azimuth}\\ \text{5. Ambiguous Range (Range Gate Number)}\end{array}\right\} \quad (9)$$

When all the individual pieces of a target observed at different PRF's have been identified as in (5)-(8), they must be brought together and centroided by the following computations (output):

$$\left.\begin{array}{l}\text{1. Signal Strength} = \Sigma\ \text{Signal Strength}\\ \text{2. Range Rate} = \Sigma\ \dfrac{\text{Filter X Signal Strength}}{\Sigma\ \text{Signal Strength}}\\ \text{3. Elevation} = \Sigma\ \dfrac{\text{Elevation X Signal Strength}}{\Sigma\ \text{Signal Strength}}\\ \text{4. Azimuth} = \Sigma\ \dfrac{\text{Azimuth X Signal Strength}}{\Sigma\ \text{Signal Strength}}\\ \text{5. Quality Index} = \text{Number of Observations}\\ \text{6. Computed Range (Reference equation (4))}\end{array}\right\} \quad (10)$$

Chinese Remainder Theorem

The Chinese Remainder Theorem is used to compute true range from ambiguous range. The adaptation of this technique to a parallel algorithm is described below.

In a "3 of 8" PRF system, there are 120 possible valid PRF combinations with which to compute the range. This assumes combinations of 7 items taken at a time with the eighth item fixed and always chosen. Examples of four valid PRF combinations that can be used to compute range are: 8,7,6,5,4,3,2,1 or 8,6,5,2,1 or 8,7, 2,1 or 8,3,1 (where 8 is the oldest PRF of interest and 1 is the youngest PRF of interest). For example, the PRF prime number D1 for PRF8 might equal 33, for PRF7, D2 might equal 38.

If one considers that the correlation technique to be presented permits a maximum of 4 matched correlations per PRF, there can be 312, 384 total CRT combinations for an 8 PRF system. By modifying the approach slightly, only the "3 of 8" correlations need be produced in order to compute all the possible target ranges. Four examples of "3 of 8" are: 8,7,6 or 8,5,3 or 8,6,5 or 8,2,1. If three identical ranges are produced after all possible "3 of 8" CRT are computed, this corresponds to a "4 of 8" target return. If six identical ranges are found, this corresponds to "5 of 8." As an example, if the range produced by PRF 8,7,6 = 8,7,4 = 8,6,4, this is the same as "4 of 8" target returns - 8,7,6,4. A "4 of 8" range computation after deghosting has only one chance in a thousand of being a ghost. Looking for any "3 of 8" correlation requires only 21 possibilities. As the correlation routine permits a maximum of 4 matched correlations per file, there are in total 1,344 possible ways to look at any "3 of 8."

To insure that all range gate straddle possibilities are also computed along with the "3 of 8," every "3 of 7," "3 of 6," "3 of 5," "3 of 4" and "3 of 3" possibilities with identical range rate and elevation, must also be computed.

Examples of "3 of 7" are: 7,6,5 or 7,4,3 or 7,2,1.

Examples of "3 of 6" are: 6,5,4 or 6,3,2 or 6,2,1.

Examples of "3 of 5" are: 5,4,3 or 5,3,1 or 5,2,1.

All the examples of "3 of 4" are: 4,3,2 or 4,3,1 or 4,2,1.

The example of "3 of 3" is: 3,2,1.

After all the "3 of 8" through "3 of 3" possibilities have been computed, all are tested for a range in excess of 300 miles. All the answers that exceed 300 miles are considered ghosts and discarded. Then each surviving "3 of 8" is tested for a match with every other "3 of 8," through "3 of 3," for a window of ± 1 range gates. If two or more are found (excluding redundant answers), a "4 of 8" possibility has been identified.

Assumptions

The following system design assumptions are made for the conceptual design:

1. The antenna beam width and rotational rate permit a maximum target observation of 8 scans.

2. A maximum of 4 new targets per scan are observable.

3. Four scans of raw digitized input data must be processed into a final report in less than 20 milliseconds of CPU time.

4. There are eight or more unique and reasonable PRF's in which a target can be observed.

5. Correlation criteria prior to the CRT requires a minimum of 3 target observations out of the possible 8 PRF's.

To satisfy the above assumptions 1, 2, and 3, the following hardware requirements are made:

6. There is special purpose inter/intra array data transfer hardware similar to the STARANS PIO.

7. The processor contains four or more active 4096 x 256 bit associative arrays.

8. The processor will contain high speed storage sufficient to contain four or more arrays of data.

Any relaxation in assumptions presented in 1, 2 or 3 will reduce the number of arrays required and the size of the array required.

The concept that will be presented on the following pages will be designed to the assumptions just presented.

Overview of Parallel Radar Signal Processor

Figure 4 depicts the parallel adaptation of a Doppler Signal Processor. Each scan of raw digitized input data is stored into a 4096 word x 256 bit array or other storage device as the returning data is gathered. At the completion of the fourth scan, processing of these data is begun in unison in all four arrays.

Reference [2] and [3] discuss the feasibility of parallel implementation of FFT and CFAR respectively.

The results of the FFT, CFAR and centroid is to reduce or compress each scan of digitized input data of 0.1 to 0.5 million bits to one scan file which is composed of 64 words or less; each word is less than 100 bits long and contain the attribute listed under target eq (9).

Correlation of the 4 most recent scan files to their respective 7 preceding scan files is then performed. The correlation by similar range rate and elevation is used to select up to four subsets of the input data (based on the assumption of a maximum of 4 targets). Each selected subset is placed in a 4096 x 256 bit array. This data is arranged in such a manner that after four multiplies and one divide, all the possible combinations of range (ghost and true) are produced including straddle. In effect, the preceding correlation reduces the selection of target reports for CRT calculation to a reasonable subset and then every possible answer is computed.

More than one true target may be computed in each array as two targets with identical range rate and elevation may have been flying in formation.

The criteria for target identification is four target observations at one range, or two sets of three target observations at adjacent ranges. This limitation reduces the probability of a ghost report to one in a thousand.

Component words of File n that produce the acceptable target range from the eight scan files are collected in another array for eventual tree summing into one final output report via eq (10).

Detailed Description of Design

The following is a detailed description of correlation through tree sum of final target report as outlined in the lower half of Figure 4.

Correlation Technique

Following Filter/Elevation Centroiding, file n through n-3 in array 1, 2, 3 and 4 is arranged with older files n-10 to n-4 as shown in Figure 5. A closer inspection of array 4 is provided in Figure 6.

A tolerance to the elevation and range rate data of the 64 word file n-10 is used to establish the correlation window. These windows are denoted in Figure 6 under the STANDARD + Δ and $-\Delta$. The 64 word file is then replicated 32 times down the field marked by the STANDARD. Files n-10 through n-3 which are to be correlated with file n-10 are in the comparison field.

There are four copies of each file and each is rotated by 16 in each subsequent copy. Files other than the STANDARD contain all the data found in (9). In addition, a pointer designating the file number and words within the file is included for data management.

Correlation begins with simultaneous field to field comparisons between the STANDARD (file n-10) and all other files. Results of successful comparisons, those that fall within both a range rate and elevation window, are stored in the array Storage Area. This technique permits a maximum of four successful correlations for each word of file n-10.

After a comparison is made and successful comparisons stored in the Storage Area, files n-10 through n-3 are rotated 1, mod 64 and the comparison is made again to the stationary STANDARD. Successful comparisons will again be stored in the array Storage Area. In total, 16 rotations and 8 comparisons per rotation are required for complete correlations. In addition, 16 field to field moves are required to store the correlated files in the Storage Area. These comparisons are occuring simultaneously in arrays 1, 2, 3, 4, not only array 4.

When a successful field to field comparison occurs twice to the same STANDARD, it will lock out possible future candidates from occupying that same Storage Area. To prevent loss of a valid correlation, the second successful comparison of the same word will be bumped down 64 words and stored in the next Storage Area within that same file. If the Storage Area is again already taken, it will bump down 64 more and try again. After the fourth unsuccessful try, the data is discarded by this algorithm.

Following correlation, the correct PRF prime number, azimuth and the product (azimuth x amplitude), are added to successful comparisons in the Storage Area for each file. Adding this data last saves time while correlating as the added 45 bits need not be rotated.

Selection Of A Candidate For CRT

Following simultaneous correlation in four arrays, the array with the oldest data, array 4, is selected. Data in arrays 1, 2, 3 are saved. The Storage Area in array 4 contains the results of all successful correlations. These successful correlations are marked by a bit slice.

A simple algorithm employs the bit slice to select words with similar range rate and elevation. As each unique set of range rate and elevation is selected (maximum of four), the input data listed below is shipped into its own array 1, 2, 3, or 4.

1. Range Gate Number (Ambiguous Range)
2. Range Rate (Filter Number)
3. Signal Strength
4. Azimuth
5. Azimuth X Signal Strength
6. Elevation
7. Data Management Pointers
8. PRF Prime Number

(11)

Figure 10 shows the data movement for up to four unique range rate and elevation combinations from array 4 to the arrays that will perform the CRT.

CRT Computations In The Array

After the data has been shipped to array 1,2,3 and 4, these four arrays will rearrange their data in preparation for a CRT. The task will be to compute range by the formula (4). In Figure 8, the oldest scan file (n-10) will be considered PRF8; n-9 will be PRF7; etc.

Figure 8 shows an example of the parallel data arrangement in the array.

Following the range computation and deghosting, a serial search is performed in each array to pick the first "3 of 8" survivor. The first survivors range is subtracted from all of the other survivors. A test is made on the resulting delta range for a window of ± 1. If there are none, discard and pick the next "3 of 8" survivor and repeat the subtraction and ± 1 range comparison. When one or more targets are located within a window of ± 1, identify all the responders as the same target and mark this range closed to further comparisons. When all the "3 of 8" have been accounted for by this serial search in all four arrays, begin the data movement to array four in preparation for a tree sum.

Movement Of Data After CRT

Following CRT computations and identification of one or more valid answers (see Figure 9), the first valid answer (range A) is moved to array 4 starting at word 0. Each valid answer is the result of three raw words such as 8,7,4, (where 8 is a word from file n-10 containing the attributes listed in eq (11); 7 is a word from file n-9, etc.). Words 0 through 2 are then used to store these three file words. If there is another range computation within a ± 1 range gate window, the three words that created this range are moved to array 4 directly behind the previous three words starting at word 3. In all, 64 words are set aside to collect all possible permutations of the same target.

Following array 4 interrogation, array 1 is then interrogated for target reports. In Figure 9, two targets were identified in array 1, one with range B and the other with range C. Both targets had similar range rates and elevation. The first range discovered, range B, is moved to start at word 64. Any three part permutations are then moved behind it as described before. The second target at range C is then moved to start at word 128.

After arrays 2 and 3 have been interrogated for their range contributions, this task is complete.

Data Management

Figure 10 exemplifies storage on a mod 64 boundary of two unique targets in array 4. Starting at word 0, a target was observed at range 1234 and 1235 (300 ft. away). The arrangement of the data suggests that the data at range 1234 and 1235 are the same target and each answer was produced in the same array. Also notice the file number and word number duplications in word 0 and 1 with word 3 and 4. These duplications must eventually be identified, and all duplications marked deleted by the use of a bit slice called SAVE.

In Figure 10 at word 64, the data again start with another unique range. As the range rate and elevation of word 64 are different from words 0 through 8, it can be assumed that the ranges following word 64 were not created in the same array as words 0 through 8.

When all the data has been placed into array 4, correlation, azimuth, beamsplitting and range gate destraddle are complete for file n-10. The data in array 4 is available for tree summing. However, tree summing will not be performed until the remaining correlated files (correlated to file n-9, n-8 and n-7, Reference Figure 5, arrays 1,2, and 3) have also gone through the CRT calculations.

The file correlated to n-9 is next to undergo CRT computation. Correlated n-9 must be restored into an array and the data to be tree summed saved before CRT processing can begin. The restored n-9 cannot be used until components of accepted valid ranges have been deleted. For example, (see Figure 10) file n-4, word 24 is now known to be part of a target at range 1234. It therefore cannot be part of any other target and should not be considered in any future CRT calculation. Figure 11 shows that these pointers are used to mark deletions in the unprocessed correlated files. Failure to delete these components would tend to generate a future multitude of recurring answers that would quickly exceed the capacity of the system.

When all the pointers have been used to mark deletions in the remaining correlations, CRT processing can again begin for the next oldest file n-9. This will again require development of a new mask and processing defined under Selection Of A Candidate For CRT.

After processing correlated file n-9, then n-8, then n-7, the CRT computations are complete for this computer cycle. Copies of file n-6 thru n must be saved for the next computer cycle when they will then become file n-10 to n-3.

Files n-6 through n have been continually updated with deletions to its 64 word file after every satisfactory CRT range selection. Upon final storage, file n-6 will contain the greatest number of deletions and file n the least number of deletions.

The last step in this computer cycle will be to tree sum array 4, the collector of all the correlated target data.

Tree Sum For Final Report

All the data is now set up on a mod 64 basis in array 4 (see Figure 11). This data is further arranged for maximum parallelism in preparation for a tree sum of 64.

A tree sum of 64 quantities requires only 6 adds. In 6 adds, the following will be tree summed for as many as 16 unique targets, where a unique target can contain up to four more targets at similar range rate and elevation -

$$\left.\begin{aligned} A &= \Sigma \text{ Signal Strength} \\ F &= \Sigma \text{ Filter X Signal Strength} \\ E &= \Sigma \text{ Elevation X Signal Strength} \\ Z &= \Sigma \text{ Azimuth X Signal Strength} \\ \text{Quality} &= \text{Number of Observations} \end{aligned}\right\} \begin{array}{l} \text{6} \\ \text{ADDS} \end{array} \qquad (12)$$

Following the six adds, one divide will be performed.

$$\left.\begin{aligned} F \div A \\ E \div A \\ Z \div A \end{aligned}\right\} \text{ One DIVIDE} \qquad (13)$$

The result of performing (12) and (13) will be a final target report (output) for each target as defined in eq (10).

In all, the preceding correlation required only 17 multiplies and 5 divides for four scans of data. One multiply per four scans computed the product of (Azimuth * Signal Strength); four multiplies and one divide per scan computed the range; and one divide per four scans computed the final output report.

In summary, the correlation merely selects four subsets of all possible targets and places them into four arrays. Each subset has its own unique elevation and range rate. The data is arranged in such a way that all possible CRT calculations are performed in parallel on each subset at the same time. Within each subset, if two or more answers agree exactly, or within a window, it is assumed to be a valid target.

The input to each subset is limited, and only four valid answers can be produced per subset. As there are four subsets per scan, it is conceivable sixteen valid answers can fall out of one scans computation. Four scans are processed in one computer cycle producing 64 possible answers per four scans. These 64 possibilities are however limited by the correlation technique's sensitivity to Range Gate Straddle Probability, and the difficulty in finding air traffic with that great a density.

Advantages Of An Array Data Base Random Scan

Teaming of an AP with a Phased Array antenna can lend a new versatility to this AP Radar Signal Processor Design. Consider the array as a radar data storage area that need not be processed until sufficient data is available from other scans to begin processing. Consider the antenna capable of transmitting in any reasonable direction.

Then given the sufficient array space, there is no requirement to scan the horizon in a sequential pattern. A scan of data can be stored in each array until a suitable sector of the sky is available for correlation. This would in effect permit a Random Scan Search Radar at any PRF.

Conclusions

1. During the design of these algorithms, an interesting observation was made for the performance of an AP compared to a serial machine at both a high and a low PRF. If the pulse rate is doubled for a constant number of targets, then:

Serial Machine

a. Doubles the time required to centroid filters, FFT, elevation, etc.

b. Reduces time to correlate (not by a factor of 2).

Parallel Machine

a. Halves the capacity of the algorithm for centroided filters, FFT, elevation, etc. with little effect on time.

b. Doubles the capacity for correlation.

In effect, an AP would have different array specification requirements for a different PRF, while a serial processor would require more CPU's to meet higher PRF requirements. The addition of arrays may be expensive but has little effect on the original software. The addition of a special purpose CPU may not be as costly as an array, but the serial software changes would be more extensive and costly than the AP software change.

This observation suggests that the AP as a radar signal processor is more modular, and due to its versatility (capable of CFAR, FFT, Beamsplitting), perhaps a better candidate for future "plug-in modular" radars than is a special purpose serial processor.

2. A reduction of the timing assumptions or azimuth extension assumptions will reduce system costs. Reduction of these assumptions will also permit test bed implementation of the concept on the four array STARAN computer at Rome Air Development Center (RADC). It is hoped that this implementation will provide further incentive to improve these presented concepts by identifying the risks, cost and system weaknesses.

The work reported in this paper was supported by RADC Contract F-30602-75-C-0112.

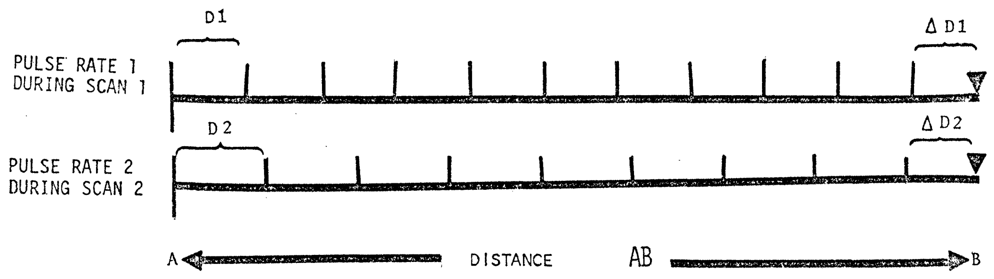
References

1. McGraw Hill, Skolnik, Radar Handbook, 1970.

2. Goodyear Aerospace Corporation, Application Of STARAN To Fast Fourier Transform, GER-16109, May 31, 1974.

3. Couranz, Gerhardt, Young, Programmable Radar Signal Processing Using The RAP, proceedings of 1974 Sagamore Computer Conference.

4. Harvey Cohn, John Wiley and Sons, A Second Course in Number Theory, 1962.



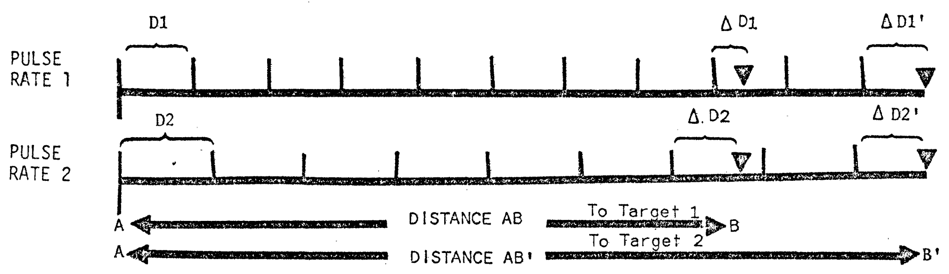
ΔD1, ΔD2 MEASURED IN UNITS OF 300 FT.

D1, D2 DISTANCE OF PULSE PROPAGATION BETWEEN PULSES

A TRANSMITTER/RECEIVER ANTENNA

B TARGET

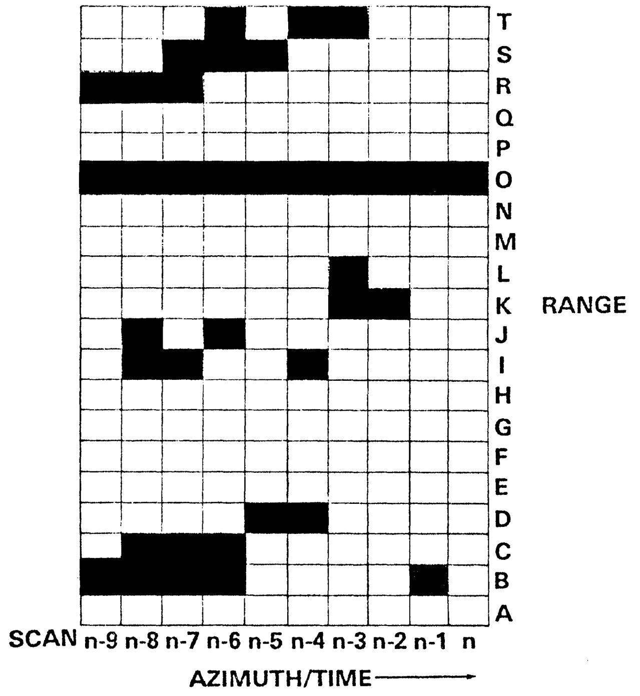
Figure 1 HIGH PRF RANGE MEASUREMENTS



TRUE RANGE OF TARGET 1 = $\frac{C1(D2)\Delta D1 + C2(D1)\Delta D2}{D1 \times D2}$ TRUE RANGE OF TARGET 2 = $\frac{C1(D2)\Delta D1' + C2(D1)\Delta D2'}{D1 \times D2}$

RANGE OF GHOST TARGET 1 = $\frac{C1(D2)\Delta D1' + C2(D1)\Delta D2}{D1 \times D2}$ RANGE OF GHOST TARGET 2 = $\frac{C1(D2)\Delta D1 + C2(D1)\Delta D2'}{D1 \times D2}$

Figure 2 GHOST PROBLEMS IN PULSE DOPPLER RADARS



Each box represents one target report.

FIGURE 3. AZIMUTH TO AZIMUTH CORRELATION AND RANGE GATE STRADDLE

ARRAY 1		ARRAY 2		ARRAY 3		ARRAY 4	
N-7	N	N-8	N-1	N-9	N-2	N-10	N-3
N-7	N-1	N-8	N-2	N-9	N-3	N-10	N-4
N-7	N-2	N-8	N-3	N-9	N-4	N-10	N-5
N-7	N-3	N-8	N-4	N-9	N-5	N-10	N-6
N-7	N-4	N-8	N-5	N-9	N-6	N-10	N-7
N-7	N-5	N-8	N-6	N-9	N-7	N-10	N-8
N-7	N-6	N-8	N-7	N-9	N-8	N-10	N-9
N-7	N-7	N-8	N-8	N-9	N-9	N-10	N-10

FIGURE 5 ARRAY SETUP PRIOR TO CORRELATION

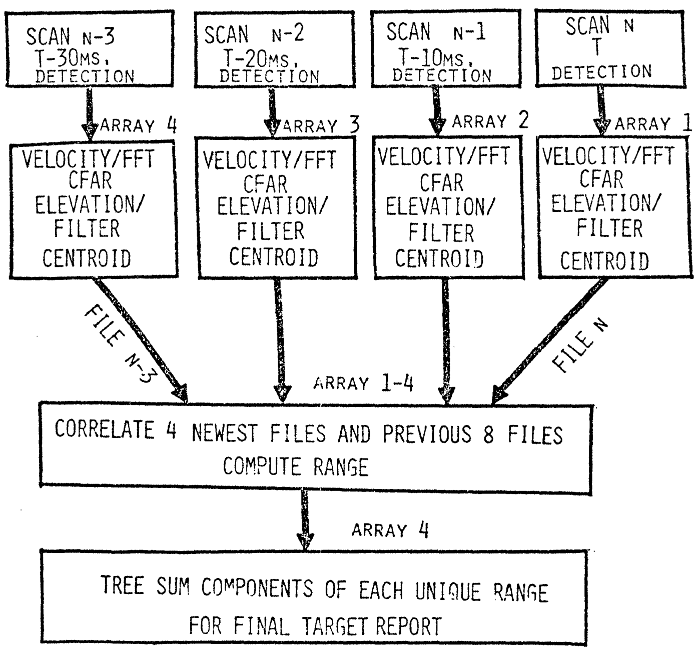

SCAN N-3 T-30MS. DETECTION
SCAN N-2 T-20MS. DETECTION
SCAN N-1 T-10MS. DETECTION
SCAN N T DETECTION
ARRAY 4
ARRAY 3
ARRAY 2
ARRAY 1
VELOCITY/FFT CFAR ELEVATION/ FILTER CENTROID
VELOCITY/FFT CFAR ELEVATION/ FILTER CENTROID
VELOCITY/FFT CFAR ELEVATION/ FILTER CENTROID
VELOCITY/FFT CFAR ELEVATION/ FILTER CENTROID
FILE N-3
ARRAY 1-4
FILE N
CORRELATE 4 NEWEST FILES AND PREVIOUS 8 FILES COMPUTE RANGE
ARRAY 4
TREE SUM COMPONENTS OF EACH UNIQUE RANGE FOR FINAL TARGET REPORT

FIGURE 4 PARALLEL RADAR SIGNAL PROCESSOR

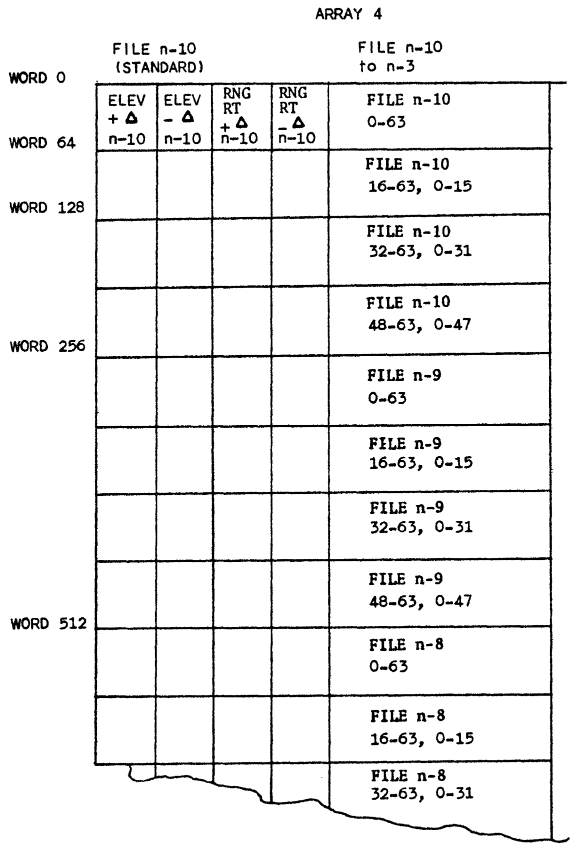


Figure 6. CORRELATION TECHNIQUE

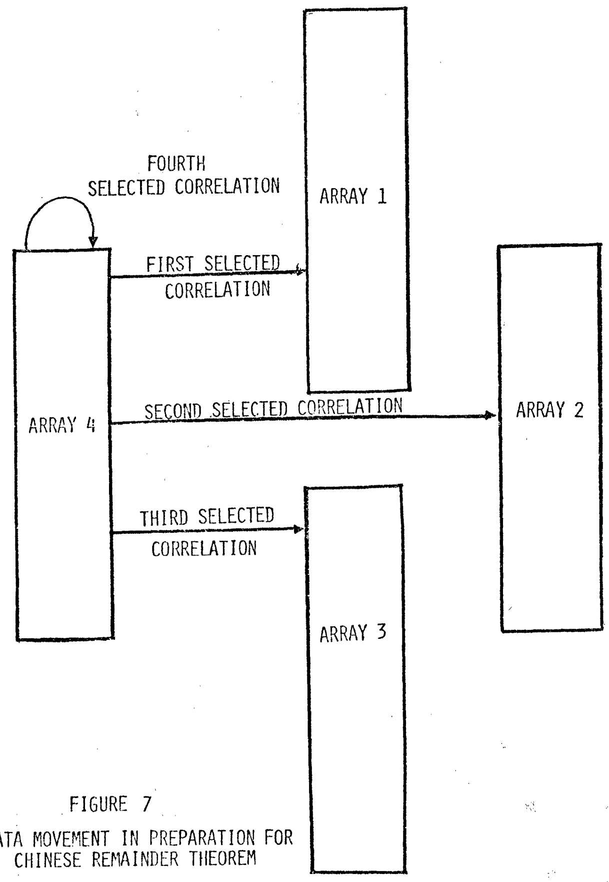

FOURTH SELECTED CORRELATION
ARRAY 1
FIRST SELECTED CORRELATION
ARRAY 4
SECOND SELECTED CORRELATION
ARRAY 2
THIRD SELECTED CORRELATION
ARRAY 3

FIGURE 7
DATA MOVEMENT IN PREPARATION FOR CHINESE REMAINDER THEOREM

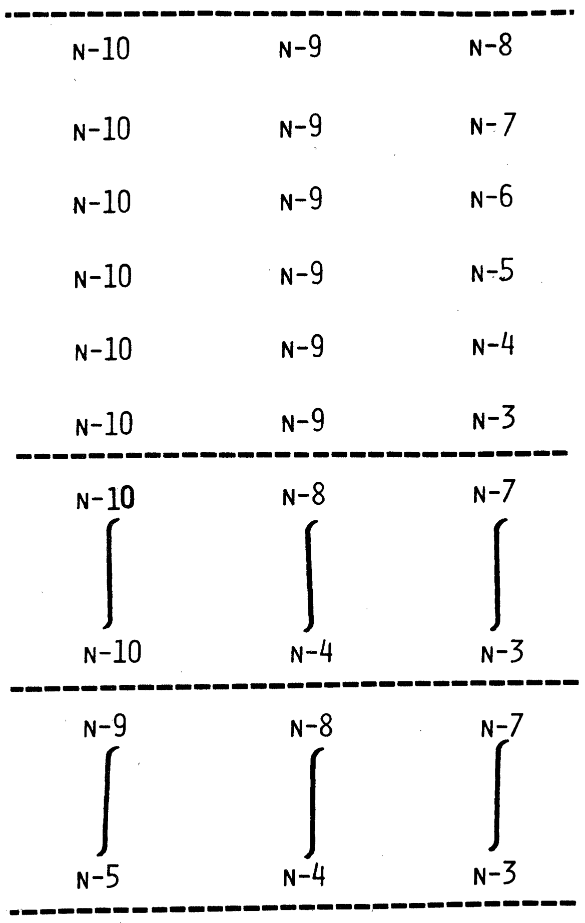


FIGURE 8
ARRANGEMENT OF FILES N-10 THRU N-3
PRIOR TO RANGE COMPUTATION

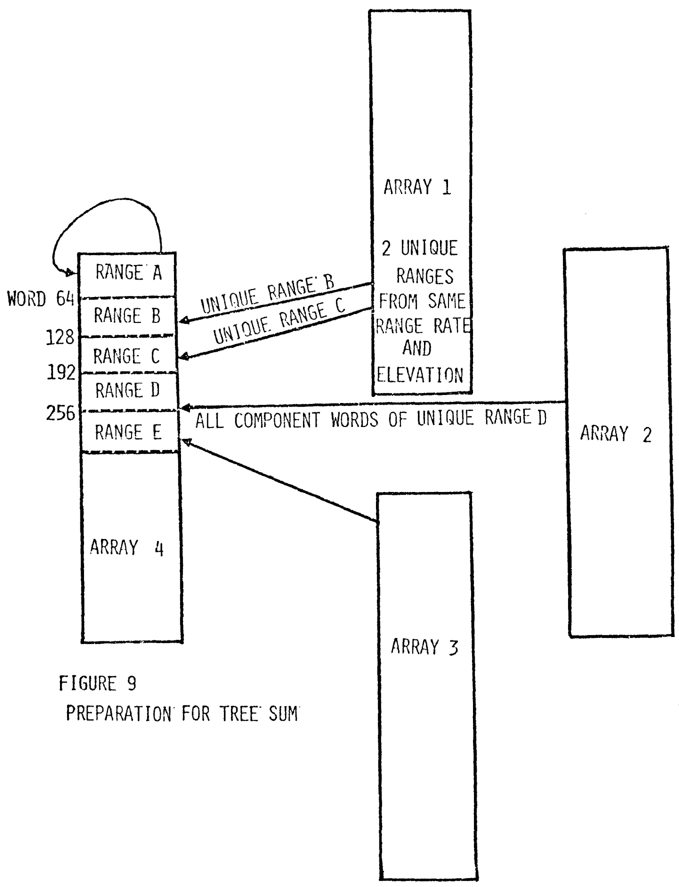


FIGURE 9
PREPARATION FOR TREE SUM

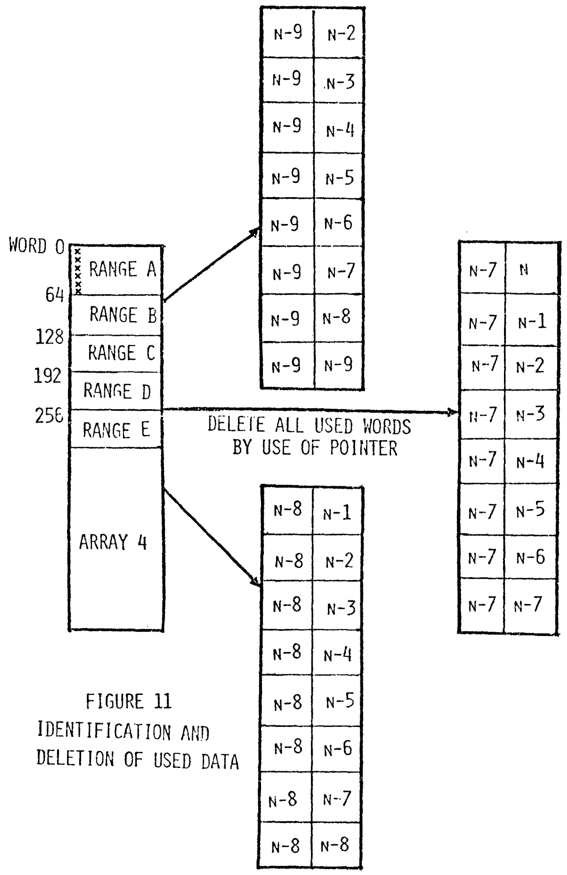


FIGURE 11
IDENTIFICATION AND
DELETION OF USED DATA

WORD 0	RANGE	AZ	EL	RNG RT	AMP	S/N	FILE NO.	WORD NO.	SAVE
	1234	100	3	17	16	13	n-8	7	1
	1234	99	3	17.2	22	15	n-7	52	1
	1234	97	3	17	30	15	n-5	9	1
	1234	100	3	17	16	13	n-8	7	0
	1234	99	3	17.2	22	15	n-7	52	0
	1234	96	3	17	28	14	n-4	24	1
	1235	100	3	17	15	13	n-8	29	1
	1235	99	3	16.8	20	15	n-7	33	1
	1235	98	3	16.8	28	15	n-6	16	1
WORD 64	2387	100	7	42	19	15	n-8	15	1
	2387	99	7	42	28	14	n-7	19	1
	2387	98	7	42	40	15	n-6	37	1
	2386	100	7	42	20	15	n-8	21	1
	2386	99	7	42	30	15	n-7	32	1
	2386	98	7	42	42	16	n-6	5	1

Figure 10 DELETION OF REPEATS

AN OPTIMAL SYNCHRONOUS REFRAMING TECHNIQUE USING AN ASSOCIATIVE PROCESSOR

Terrence L. Saxton and Cheng-chi Huang
Systems and Research Center
Honeywell, Inc.
Minneapolis, Minnesota 55413

Abstract -- Data loss resulting from the time to regain frame synchronization following an out-of-frame condition on a synchronous time division multiplexed bus or line, such as the Bell System T1 digital line, can be substantial. Most current schemes require many frame times to separate with high probability the true frame pattern from identical patterns occurring temporarily in random data. A novel technique based on an associative processor was conceived which can reduce the reframe time to its minimum value by eliminating the time spent dwelling at false frame positions.

Introduction

Synchronous high speed digital transmission terminals typically arrange serial bit streams into periodic line formats called frames. Receiving terminals require at least two levels of synchronization to permit proper decoding of the transmitted information:

1. Bit synchronization, to bring the receiver clock into phase with the transmitted bit stream for sampling of the bit values, and

2. Frame synchronization, to bring the receiver into phase with the transmitted line format or frame so that the bits can be properly arranged for decoding and/or demultiplexing.

Our concern in this paper is only with frame synchronization, and in particular with a novel technique to regain frame synchronization in the minimum possible time after a loss of frame synchronization caused by errors due to transmission path noise or equipment faults. The technique to be described can be implemented with a special purpose associative processor realizable with currently available off-the-shelf components.

Framing Strategies

At the transmitter a unique bit pattern is added to the line format. The periodicity of this framing pattern defines the frame length. For a frame length of N bits, there are N different phases the receiver can assume, only one of which corresponds to an "in-frame" condition or mode. The framing strategy consists of a procedure for searching through all N possible phases until the one containing the frame pattern is found and verified, at which time the receiver is said to be in-frame.

The receiver framing circuitry has two modes of operation. When the receiver is in-frame, a single frame pattern bit error will not initiate reframe searches. It takes repeated violation of the expected frame pattern, such as four errors in seven frames, to cause the receiver to enter the out-of-frame mode. After entering this mode, the receiver will initiate reframe searches in order to get back to the in-frame mode. In the reframe searches, a mismatch between the bits under examination at a particular phase position and the expected frame pattern bits will cause a shift to the next phase in the frame. When the expected frame pattern bits are found in a particular phase of the frame, the same phase of the next frame will be searched. The receiver returns to the in-frame mode when the frame pattern persists at that phase for a given number of times, k.

Reframe strategies are evaluated by their effects on system performance as measured primarily by reframe time and by the complexity of their circuit implementations. Reframe time is the length of time spent in an out-of-frame mode, during which all transmitted information is lost. A long reframe time may be required because random bits in a non-frame synchronization phase may happen to match the frame pattern, resulting in a false frame position until a mismatch occurs.

The expected time spent at a false frame position, $E(t_{ff})$, can be shown to be [1], [2]:

$$E(t_{ff}) = \sum_{n=1}^{\infty} \frac{nT}{(2^m)^n} = \frac{2^{-m}T}{(1-2^{-m})^2} = \frac{2^{-m}Nt}{(1-2^{-m})^2}, \qquad (1)$$

where
- m = the number of frame bits
- T = Nt, period of the frame
- N = number of bits/frame
- t = time duration of one bit

Searching over all possible positions requires N such tests. If the pattern is further required to persist for a minimum of k periods to achieve an acceptably low probability of false-frame, then the maximum average reframe time, $[\bar{t}_{rf}]_{max}$, is given by:

$$[\bar{t}_{rf}]_{max} = (N-1)E(t_{ff}) + kT. \qquad (2)$$

As an example, consider the Bell System T1 digital transmission facility operating at 1.544 megabits per second with a frame length of 193 bits and a three bit frame pattern. On the average, a receiving terminal will dwell at a false frame position for .16 periods during a search. For T = 125 microseconds and k = 4, the maximum average reframe time is 4.4 milliseconds.

Reframing Using an Associative Processor

Approach

Using associative processing techniques for reframing, reframe time may be significantly reduced. This is accomplished by performing an associative equality search operation for the frame pattern simultaneously (i.e., in parallel) over all possible phases of the frame. The reframe time attributed to dwelling at a false frame position is thus eliminated. The reframe time, t_{rf}, is now given by

$$t_{rf} = kT. \qquad (3)$$

The value of t_{rf} has thus been reduced to its minimum possible value for a given value of k, and in this sense is now optimal. For the example given above, the value of t_{rf} is reduced from 4.4 milliseconds to 0.5 milliseconds.

An associative processor implementation of this reframing procedure for the previous example with the three bit frame pattern contiguously grouped at the beginning of a frame is briefly described below.

An Associative Processor Implementation

The data paths for a simple associative processor are shown in block diagram form in Figure 1. The incoming data stream is serially shifted through the 1-bit wide X register from bottom to top. An Associative Memory, AM, has 193 words (one for each phase), each of which has a frame pattern field (for the three frame bits F_0, F_1, and F_2) and a counter field to record the number of matches. At appropriate times the X register is written as a bit slice into the frame pattern field of the AM. A Y register is used to record matches following equality searches with the search register, S. A special multiple match logic circuit is connected to each bit position in the Y register to indicate zero, one or more than one matches following a search operation.

System Operation

In each period during the out-of-frame mode, the bit stream $(b_0, b_1, \ldots, b_{192})$ is loaded into the X register, immediately following the last bit (b'_{192}) of the previous frame. The stream is shifted up until, for example, the pattern shown in the X register of Figure 1 is present, at which time it is loaded into Column 1 of the array. Likewise, after each of the next 2 shifts, Columns 2 and 3 are loaded from the X register. The 3-bit frame pattern field of the 193 words in the array thus contain all possible 3-bit windows to be searched. All windows are then compared by an equality search operation to the frame header pattern in the S register, and any matches are indicated in the Y register. The match counters (in the Match Counter Field) of matched windows are incremented by one.

This procedure is repeated for k successive periods. At the end of k periods, an equality search for k is performed over the Match Counter Field and the Multiple Match Logic is consulted to determine if there is zero, one, or more than one responder. One responder will result in a return to the in-frame mode. Zero or more than one will require additional searches until a single responder is left with a satisfactory number of matches. The probability of having only the correct frame pattern window with k in the Match Counter Field after k periods is $[1-(1/2^m)^k]^{192}$, which is 0.954 for m=3 and k=4. In any case, the frame header position is indicated by the location of the word which satisfies the match criteria.

This technique can easily be extended to any number of frame pattern bits, including one, and to distributed as well as clustered patterns within a frame. Whereas for conventional reframing schemes, exemplified by the D1 terminal for the Bell System T1 digital lines, the reframe time goes up rapidly as the number of frame pattern bits go down, the reframe time with an associative processor is only dependent on the value of k. In fact, for one bit patterns (i.e., alternating ones and zeros), the improvement in reframe time is the most dramatic. To achieve the same probability (e.g., 0.954) of correctly identifying the frame pattern window used in the previous example using a one bit frame pattern (i.e., m=1) requires k be equal to 12. The reframe time improvement in this case is from 49.5 milliseconds for the conventional approach to 1.5 milliseconds for the associative processor approach.

Conclusion

A reframing strategy based on an associative processor implementation has been described which can reduce reframe time to its minimum value. The technique eliminates the time spent dwelling at false frame positions by simultaneously searching all possible bit positions for the frame pattern. For T1 digital terminals with frame lengths of 193 bits, a period of 125 microseconds, and a one bit frame pattern, the reframe time can be reduced from 49.5 milliseconds to 1.5 milliseconds. The loss in information is thus reduced from 396 frames to 12 frames, providing a significant improvement in system performance.

References

[1] Members of the Technical Staff, Bell Telephone Labs, Transmission Systems for Communications, Fourth Edition, Western Electric Co., Inc., Technical Publications, Winston-Salem, North Carolina, 1970, 759 pp.

[2] L.B.W. Jolley, Summation of Series, Dover Publications, Inc., New York, 1961.

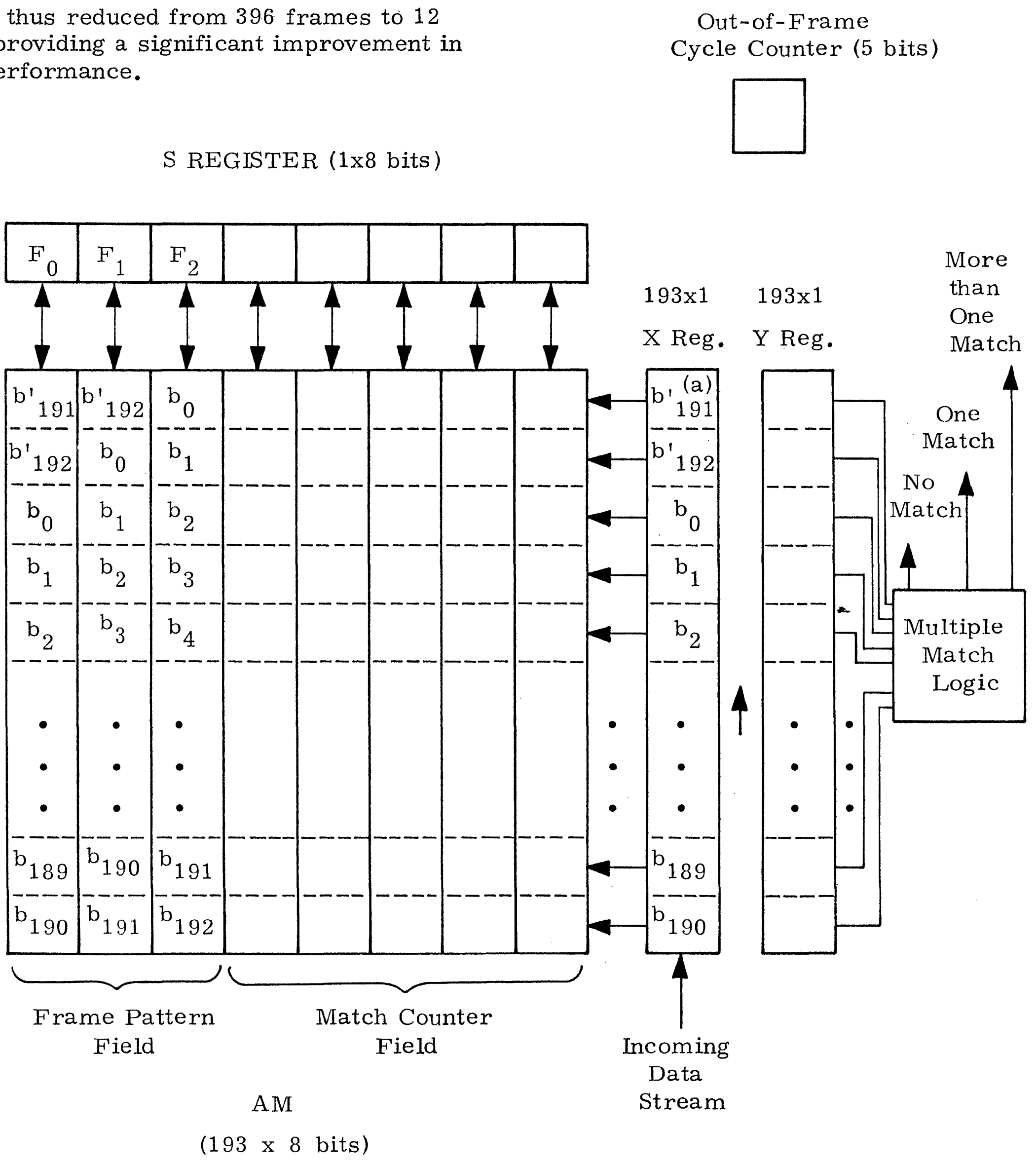


Figure 1. A Simple Associative Processor for Reframing

(a) b'_i are bits from the preceeding frame.

ORGANIZATION OF SEMICONDUCTOR MEMORIES FOR PARALLEL-PIPELINED PROCESSORS[a]

Faye A. Briggs and Edward S. Davidson
Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

Abstract -- A multimodule semiconductor memory organization for parallel-pipelined processors is developed using a Markov model. Buffered - and non buffered - request schemes were simulated for various memory configurations. Performance is evaluated as a function of configuration.

I. Introduction

This paper describes a method for exploiting the characteristics of semiconductor memories to obtain a multimodule memory organization for parallel-pipelined multiinstruction-stream processors.

In simple terms, semiconductor memories require an address to be held on the address lines for a_o seconds (an address cycle) and can perform a complete memory operation in c_o seconds (a memory cycle)[b]. In general $a_o \leq c_o$. We assume that the address is typically held on the address lines at least as long as data is held on the data lines. Hence the data lines do not pose a limiting constraint and are not considered explicitly.

A parallel-pipelined processor of order $(\underline{s},\underline{p})$ [1] is modeled here as a set of $\underline{p}$ independent but synchronized processors each of which is a pipelined processor of degree $\underline{s}$ consisting of $\underline{s}$ segments (physical resources). Each segment can simultaneously be processing a distinct step of a distinct instruction. It is assumed that these $\underline{s}$ instructions come from distinct instruction streams as in [2], hence the degree of multiprogramming is also $\underline{s}$. Each processor segment takes 1 segment time unit (STU = τ seconds) to complete its execution step. A pipelined processor can issue one memory request per STU. Hence a parallel-pipelined processor of order $(\underline{s},\underline{p})$ can issue $\underline{p}$ requests each STU and execute $\underline{s}\cdot\underline{p}$ distinct instruction streams concurrently. There is no execution overlap between instructions from the same stream.

[a]This research was supported by the National Science Foundation under Grants GJ-35584X and GJ-40584 and by the Joint Services Electronics Program under Contract DAAB-07-72-C-0259.

[b]In actuality, we have $(\underline{a}_r,\underline{c}_r)/(\underline{a}_w,\underline{c}_w)$ which are address and memory cycles for read and write respectively.

II. Model Description

The memory and processor operations described above suggest that parallel-pipelined processing can be implemented in the memory if the memory modules are organized in a matrix form with line segment $\underline{i}$ and module segment $\underline{j}$ on line $\underline{i}$ referred to as $L_{\underline{i}}$ and $M_{\underline{i},\underline{j}}$ respectively. This organization, referred to as L-M memory organization, is shown in Figure 1.

A reservation table [3], used to illustrate the flow of a computation through the segments of a pipeline, is shown in Table 1 for a straight-through pipelined processor of order (7,1) having access to the $L_{\underline{i}}$ and $M_{\underline{i},\underline{j}}$ of the memory system whose memory characteristics are $(\underline{a},\underline{c}) = (2,4)$, where $\underline{a}$ and $\underline{c}$ are the address and memory cycles expressed as an integer number of STUs, namely $\underline{a} = \left\lceil \frac{a_o}{\tau} \right\rceil$, $\underline{c} = \left\lceil \frac{c_o}{\tau} \right\rceil$. Following initiation at time instant $\underline{t}$, an x in cell $(\underline{u},\underline{v})$ indicates that a task requires the segment associated with row $\underline{u}$ for time interval $\langle \underline{t}+\underline{v}, \underline{t}+\underline{v}+1\rangle$. Note that operations in the line and module segments cannot be preempted.

A memory collision is said to occur when a memory request attempts to access a busy line or module segment, or when two simultaneous requests attempt to access the same line segment. Hence following an initiation of a memory operation at time $\underline{t}$ on line segment $L_{\underline{i}}$ and module segment $M_{\underline{i},\underline{j}}$, $L_{\underline{i}}$ will remain busy throughout $\langle \underline{t}, \underline{t}+\underline{a}\rangle$. During this period, all requests (arriving at $\underline{t}+1, \underline{t}+2, \ldots, \underline{t}+\underline{a}-1$) will find $L_{\underline{i}}$ busy. Similarly, all requests for $M_{\underline{i},\underline{j}}$ throughout $\langle \underline{t}, \underline{t}+\underline{c}\rangle$ will result in a memory collision.

The memory organization consists of $\underline{N}$ $(=2^{\underline{n}})$ identical modules arranged such that there are $\underline{\ell}$-lines and $\underline{m}$-modules per line, where $\underline{\ell} = 2^{\underline{b}}$ and $\underline{m} = 2^{\underline{n}-\underline{b}}$ such that $0 \leq \underline{b} \leq \underline{n}$. The modules are interleaved on the low order $\underline{n}$ bits and the lines on the low order $\underline{b}$ bits of the address. This scheme tries to maximize the probability that $\underline{a}$ successive requests are to distinct lines and $\underline{c}$ successive requests are to distinct modules.

Two multiaccess content addressable memories (CAMs) of size $\underline{p}\cdot(\underline{a}-1)$ and $\underline{p}\cdot(\underline{c}-1)$ are required to store the addresses busy lines and modules respectively. On the arrival of a memory request at $\underline{t}$, CAM probes are made at $\underline{t}$ to search for the requestor's corresponding $L_{\underline{i}}$ and $M_{\underline{i},\underline{j}}$ addresses. If the line or the module is busy, the request is rejected, otherwise it is accepted and its $L_{\underline{i}}$ and $M_{\underline{i},\underline{j}}$ addresses are stored in the CAMs for exactly $\underline{a}$-1 and $\underline{c}$-1 STUs respectively.

We assume for analytical purposes, that the addresses of the $\underline{p}$ requests issued per STU are independent and uniformly distributed among the $2^{\underline{n}}$ memory modules. This assumption is conservative. In real systems, requests are often vectoral. Hence higher performance should be expected than presented here.

III. Results of Markov Analysis

Discrete Markov models were developed to aid in the performance analysis of the memory organization. By way of example, Figure 2 shows a graph of a line decomposition for $(\underline{a},\underline{c}) = (2,4)$ with $\underline{p}=1$. Labels on arcs indicate transition probabilities. The node marked "*" indicates that a request is made to the line and is accepted. A steady state solution gives the probability of acceptance, $P_A(\underline{a},\underline{c},\underline{p})$ of the $(\underline{\ell},\underline{m})$ memory configuration as

$$P_A(2,4,1) = \frac{(\ell m)^2}{(\ell m)^2 + \ell m^2 + 2\ell m - m + 1} = \begin{array}{l}\ell \text{ times the}\\ \text{probability of}\\ \text{being in state 1}\end{array}$$

Bandwidth, the expected number of accepted memory requests in a memory cycle, is an indication of performance and is given by

$$B = pcP_A(a,c,p).$$

We now investigate the general case $\underline{p} \geq 1$.

Lemma 1. The probability of a request being rejected due to a line collision with one or more of the $\underline{p}$-1 other simultaneous requests is

$$P_1 = 1 - [1 - (\frac{\ell-1}{\ell})^p]\,\frac{\ell}{p}\,. \qquad \square$$

Theorem 1. For $\underline{a}=1$, $\underline{c} \geq 1$,

$$P_A(1,\underline{c},\underline{p}) = \frac{1-P_1}{1+(1-P_1)k}\,, \text{ where } k = \frac{p(c-1)}{N}\,. \qquad \square$$

Theorem 2. For $\underline{a} \leq \underline{c} \leq 2\underline{a}$, $\underline{a} > 1$,

$$P_A(\underline{a},\underline{c},\underline{p}) = \frac{1-P_1}{1+(1-P_1)k_1 + \frac{p(1-P_1)}{m-1}[1-(1-\alpha)^{c-a}]}\,,$$

where $\alpha = \frac{m-1}{N} > 0$ and $k_1 = \frac{p(a-1)}{\ell}$. Also

$$P_A(\underline{a},\underline{c},\underline{p}) = \frac{1-P_1}{1+(1-P_1)k}\,, \text{ for } \alpha = 0. \qquad \square$$

Theorem 3. For any $1 < \underline{a} < \underline{c}$,

$$P_A(\underline{a},\underline{c},\underline{p}) \geq \frac{1-P_1}{\Delta}\,, \text{ where}$$

$$\Delta = 1 + (1-P_1)(k_1+k_2),\quad k_1 = \frac{p^{(a-1)}}{\ell} \text{ and } k_2 = \frac{p^{(c-a)}}{N}. \qquad \square$$

Theorem 4. For any $(\underline{a},\underline{c},\underline{p})$, $P_A \leq \min(1, \frac{\ell}{a \cdot p})$. $\square$

IV. Discussion of Simulation Results

Several $(\underline{\ell},\underline{m})$ memory configurations have been simulated. In the Markov model, rejected requests are simply discarded, whereas in the nonbuffered simulation case, they result in a blocked process and are resubmitted $\underline{s}$ STUs later. A buffered case has also been simulated. In this case, each of the $\underline{p}$ processors has an associated FIFO queue which accepts incoming requests. Each queue is scanned each STU for the first acceptable request. A process is blocked only while it has a queued read request. For the nonbuffered case, simulation results were within 5% of the respective analytic results. Thus the graphical plots appear to be identical.

The illustrations of Figures 3 to 8 illustrate a typical set of module characteristics, whose read and write address cycles are $\underline{a}_r = 2$ and $\underline{a}_w = 2$, respectively, read and write memory cycles are $\underline{c}_r = 4$ and $\underline{c}_w = 6$, respectively. Figures 3 and 4 show the probability of acceptance, P_A, versus the number of parallel requests, $\underline{p}$, for various $(\underline{\ell},\underline{m})$ memory configurations with $\underline{N}=64$ and $\underline{N}=1024$ respectively. For any configuration, the buffered case produces as good or better P_A than the nonbuffered case.

If we consider the size, $\underline{z}$, of the memory module fixed, then the number of memory modules, $\underline{N}$, represents the total size of the memory. Thus increasing $\underline{N}$ implies increasing the size of memory. However, if we fix the total size of the memory, an increase in $\underline{N}$ implies a decrease in $\underline{z}$. Thus, $\underline{N}$ could be interpreted in two ways.

In Figures 5 and 6, a point of inflection occurs at $\underline{\ell}=\underline{p}$. For $\underline{\ell}<\underline{p}$, the lines are saturated, resulting in excessive blocking of processes. Hence, configurations with $\underline{\ell}<\underline{p}$ would not be desirable. Buffering tends to have its maximum effect near $\ell = \underline{a}\,\underline{p}$. For small $\underline{\ell}$ and large $\underline{N}$, the bandwidth of the nonbuffered case is close to the upper bound. Buffering has little effect. For large $\underline{\ell}$ and $\underline{N}$, P_A approximates 1 in the nonbuffered case as shown in Figure 4. Again buffering cannot improve P_A significantly. Buffering can thus be used for two purposes. One, to increase the performance of a configuration with $\underline{\ell}$ in the vicinity of $\underline{a}\ \underline{p}$. Two, when $\underline{\ell}$ exceeds $\underline{a}\ \underline{p}$, buffering may be used to maintain bandwidth while reducing $\underline{\ell}$. It is worth pointing out that a decrease in ℓ requires a smaller line address decoder, fewer line drivers and fewer bits per word of the CAM required for storage of line addresses.

We can see that the effect of p is dramatic as can be seen in the bandwidth curves of Figures 5 and 6. Note, however, that a significant increase in p would warrant an increase in N to avoid excessive page faulting which is not modeled in this paper.

Figures 7 and 8 show the expected wait times, in number of instruction cycles (= s STUs), versus ℓ for various p. Only the interesting cases ($\ell \geq p$) were shown. As expected, the buffering scheme exhibits a lower wait time. In the simulation results of the buffered case for N = 64 and N = 1024, the expected mean (maximum) queue length was less than 15(20) and 10(15), respectively, for all configurations with $\ell \geq p$.

The module characteristics (a,c) also affect the performance. An increase in a reduces P_A most for small values of ℓ, while an increase in c reduces P_A most for large values of ℓ.

It has been seen that any configuration with $\ell < p$ will result in very poor performance. Reasonable performance will be obtained for $\ell \geq a p$. Further increase in ℓ and/or use of buffering depends on cost and performance factors.

References

[1] D. L. Weller and E. S. Davidson, "Optimal Searching Algorithms for Parallel-Pipelined Computers," Parallel Processing, Proceedings of the Sagamore Computer Conference, (August, 1974), pp. 291-305.

[2] L. E. Shar and E. S. Davidson, "A Multi-miniprocessor System Implemented through Pipelining," Computer, IEEE, (February, 1974), pp. 42-51.

[3] E. S. Davidson, et al., "Effective Control for Pipelined Computers," Proc. Compcon Spring 1975, (February, 1975), pp. 181-184.

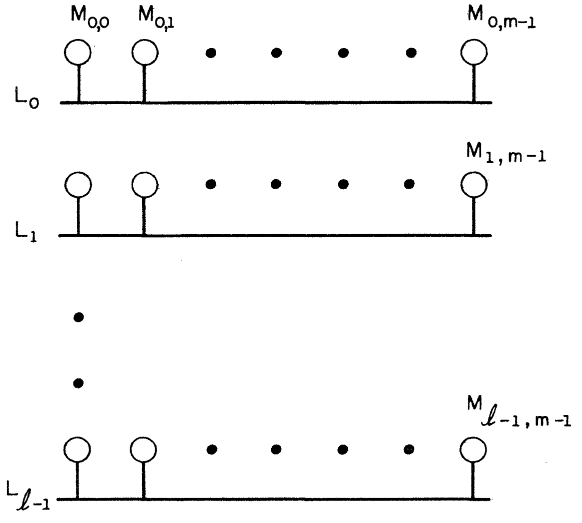
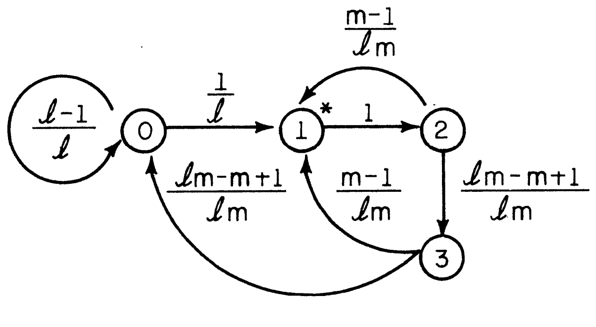


Fig. 1. L-M organization

		0	1	2	3	4	5	6	
Processor Segments	S_0	X							
	S_1		X						
	S_2			X					
	S_3				X				
	S_4					X			
	S_5						X		
	S_6							X	
Line Segment	L_i		X	X					
Module Segment	$M_{i,j}$		X	X	X	X			
Time		t	t+1	t+2	t+3	t+4	t+5	t+6	t+7

Table 1. A reservation table.



$$\begin{tikzcd}
0 \arrow[loop left, "\frac{\ell-1}{\ell}"] \arrow[r, "\frac{1}{\ell}"] & 1^{*} \arrow[r, "1"] & 2 \arrow[l, bend right, "\frac{m-1}{\ell m}"'] \arrow[d, "\frac{\ell m-m+1}{\ell m}"] \\
 & & 3 \arrow[ul, "\frac{m-1}{\ell m}"] \arrow[ull, bend left, "\frac{\ell m-m+1}{\ell m}"]
\end{tikzcd}$$

Fig. 2. A Markov diagram.

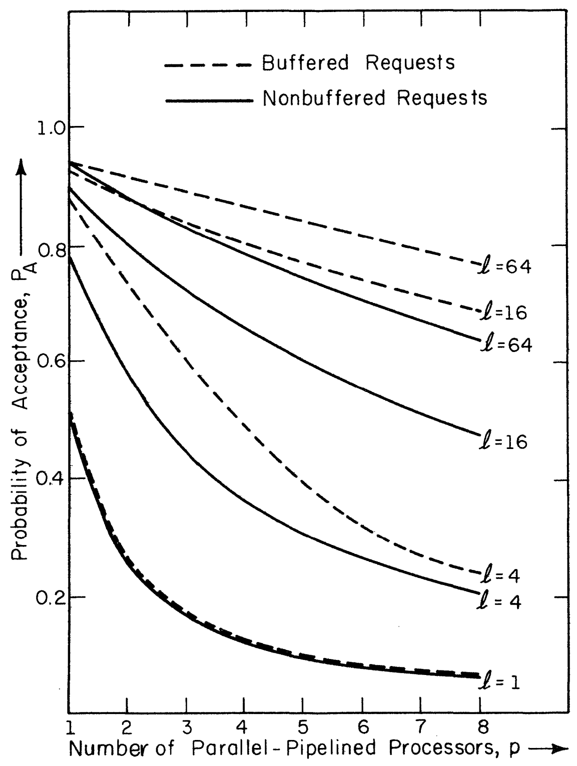


Fig. 3. P_A for N = 64.

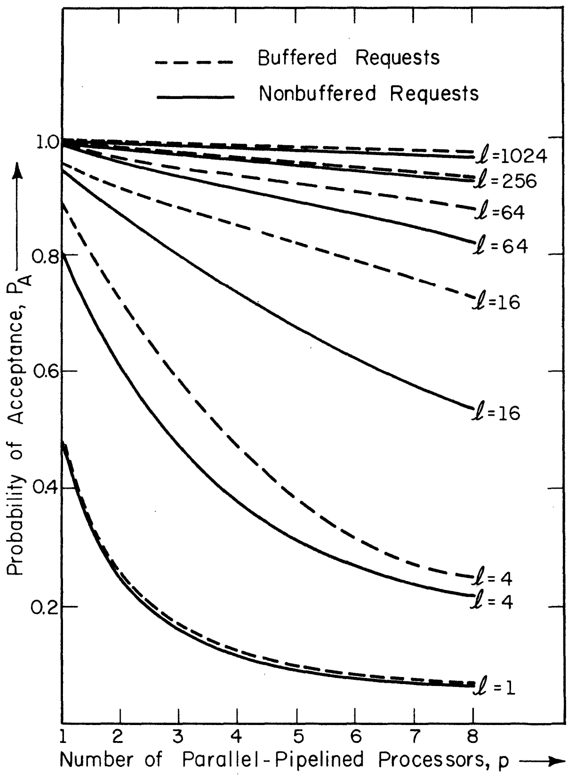


Fig. 4. P_A for N = 1024.

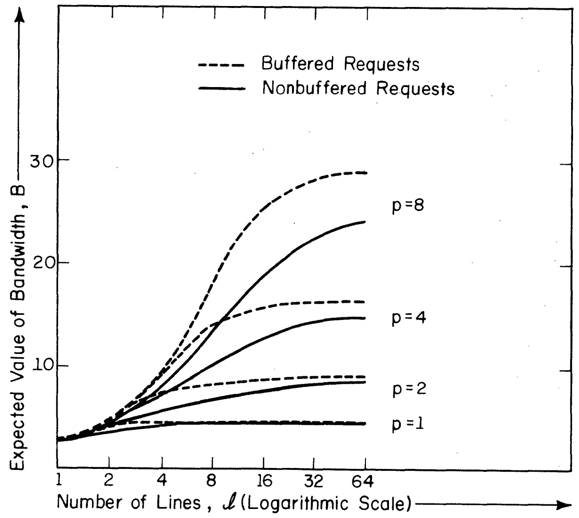


Fig. 5. B for N = 64.

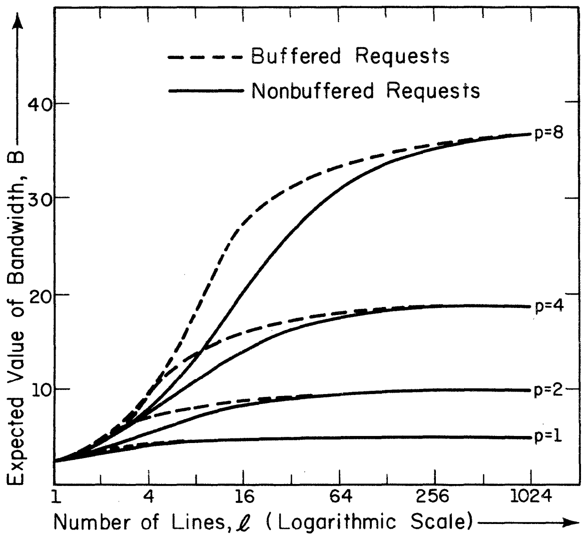


Fig. 6. B for N = 1024.

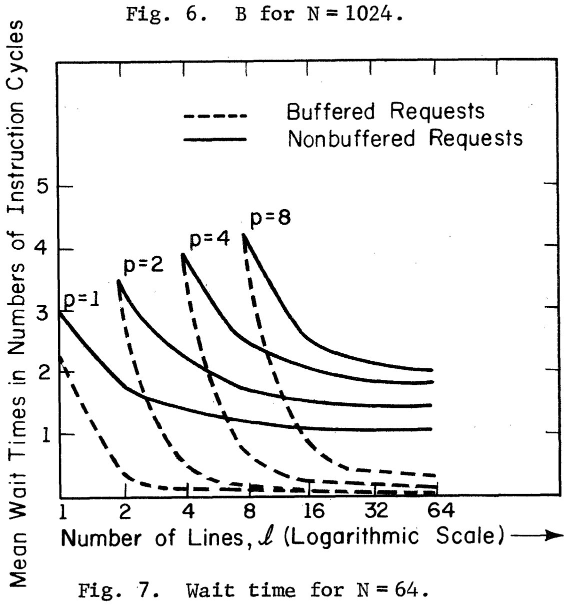


Fig. 7. Wait time for N = 64.

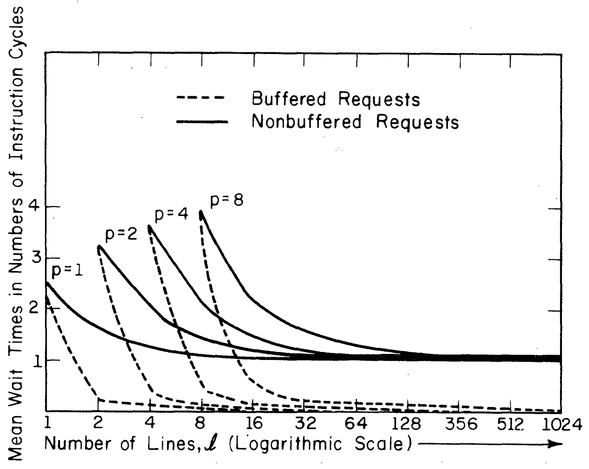


Fig. 8. Wait time for N = 1024.

STORAGE SCHEMES IN PARALLEL MEMORIES[a]

Henry D. Shapiro
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract -- Various classes of skewing schemes for storing data in independent memory modules in multi-processing computing systems are considered. Results concerning the simultaneous demand for 100% memory utilization and conflict free access to common matrix subparts are presented. Under certain conditions it is shown that both these demands cannot be met. Under other circumstances it is shown that linear skewing suffices and non-linear skewing gives no additional power.

I. Introduction

In this paper we consider the following model of a parallel computer. Besides a "control processor," we have N independent arithmetic processors and M independent memory modules. The arithmetic processors and the memories are connected by a processor-memory interconnection network (see Figure 1). We assume the interconnection network is such that in one "memory cycle" any processor can access any memory module, but two processors cannot access the same memory module. An attempt by different processors to access the same memory is called a memory conflict. If memory utilization is defined to be the ratio of the number of memories being used in one memory cycle to the total number of memories, then our goal becomes to choose N and M so that the utilization is near one when certain algorithms are executed. The algorithms we are primarily concerned with arise from numerical analysis computations on large matrices. Without discussing the algorithms in detail, it suffices to know that our data are organized as a $P \times P$ matrix, $P \gg N$ and we wish to extract any row, column, forward diagonal, or backward diagonal (see Figure 2), N consecutive elements, one per processor, at a time. A desire to keep all processors busy requires, because of our previous restrictions, $M \geq N$.

In earlier works, Lawrie [4] and Budnik and Kuck [1] investigated similar problems, imposing various restrictions on N,M and the way in which data elements can be assigned to individual memory modules. Lawrie [4] deals primarily with the case $N = 2^q$ for some q, and $M = 2N$. He gives a method for assigning data to memory modules which permits conflict free access to rows, columns, forward diagonals, backward diagonals and $\sqrt{N} \times \sqrt{N}$ blocks (when N is a perfect square). It should be noted that $M = 2N$ implies that memory utilization is $\leq 50\%$. If 100% memory utilization is desired then we must restrict $M = N$. Throughout this paper 100% memory utilization will be a design criterion, so we assume $M = N$ and use N to stand for both the number of processors and the number of memories.

Definition 1: An instance of an (x,y)-line is $\{a_{i+vy,j+vx} \mid v=\ldots,-2,-1,0,1,2,\ldots$ and the element $a_{i+vy,j+vx}$ falls within the bounds of the $P \times P$ matrix$\}$.

Pictorially, the instance of the (x,y)-line containing matrix element $a_{i,j}$ is the set of elements formed by starting at element $a_{i,j}$ and including those elements reached by repeatedly "going over x and down y" (see Figure 3). Notice that two instances of an (x,y)-line have either empty intersection or they coincide. As an example, every row is an instance of a (1,0)-line, every column an instance of a (0,1)-line, every forward diagonal is an instance of a (1,1)-line, and every backward diagonal is an instance of a (1,-1)-line.

Definition 2: A skewing scheme for a $P \times P$ matrix is a function, φ, from $\{0,1,\ldots,P-1\} \times \{0,1,\ldots,P-1\}$ to $\{0,1,\ldots,N-1\}$, where $\varphi(i,j) = m$ means matrix element $a_{i,j}$ is stored in memory m.

Defintion 3: A skewing scheme is said to be valid for a set of (x,y)-lines, $\{(x_i,y_i)\text{-lines} \mid i=1,2,\ldots,I\}$, if and only if every N consecutive elements of any instance of any (x_i,y_i)-line, $i=1,2,\ldots,I$, are stored in different memories, i.e. $\varphi(p,q)$, $\varphi(p+y_i,q+x_i)$, $\ldots$, $\varphi(p+(N-1)y_i, q+(N-1)x_i)$ are all distinct for any choice of p and q and any (x_i,y_i), $i=1,2,\ldots,I$.

If a skewing scheme is valid for a collection of (x,y)-lines then any N consecutive elements of any instance of any of them can be fetched conflict free. The main focus of this paper is to define some special classes of skewing schemes and to investigate their relative abilities to provide conflict free access to different sets of (x,y)-lines.

II. Special Classes of Skewing Schemes

Before defining the special classes of skewing schemes of interest to us, it is convenient to modify slightly our definition of skewing

[a] Work supported in part by an NSF Graduate Fellowship and in part by NSF Grant GJ-41538.

scheme. As presently defined, given a collection of (x,y)-lines, the validity of a skewing scheme, depends not only on the x_i and y_i, but also on P, the size of the matrix. Since $P \gg N$, and in general cannot be determined in advance, it is reasonable to free ourselves of this dependence on P by redefining the domain of φ to be $\{0,1,2,\ldots\} \times \{0,1,2,\ldots\}$. Mathematical justification can be found for this in

Theorem 1: Given a set of (x,y)-lines, $\{(x_i,y_i)$-lines$|i=1,2,\ldots,I\}$, if for every P, there exists a valid skewing scheme for this collection, $\varphi = \varphi(P)$, then there exist a a skewing scheme, $\varphi:\{0,1,2,\ldots\} \times \{0,1,2,\ldots\} \rightarrow \{0,1,\ldots,N-1\}$ which is also valid for the collection of (x,y)-lines.

The proof of this will not be presented here. It is a straightforward application of König's Infinity Lemma. The interested reader can find the details in [6]. The same technique can be used to eliminate the boundaries at row zero and column zero. That is, we may take the domain of a skewing scheme to be $\{\ldots,-1,0,1,\ldots\} \times \{\ldots,-1,0,1,\ldots\}$, without loss of generality. In practice the elements outside the bounds of the matrix would not be referenced or can be treated as zeros.

Two classes of skewing schemes of special interest are periodic skewing schemes and linear skewing schemes.

Definition 4: A skewing scheme, φ, is called periodic if and only if $\varphi(i,j) = \varphi(i \pm kN, j \pm \ell N)$ for $k,\ell=0,1,2,\ldots$. A skewing scheme, φ, is said to be linear if and only if there exists a and b such that $\varphi(i,j) = ai+bj \bmod N$.

We observe that a linear skewing scheme is a periodic skewing scheme, since $\varphi(i \pm kN, j \pm \ell N) = a(i \pm kN) + b(j \pm \ell N) \bmod N = ai+bj \bmod N = \varphi(i,j)$. Given a collection of (x,y)-lines, $\{(x_i,y_i)$-lines$|$ $i=1,2,\ldots,I\}$, Budnik and Kuck [1] give a necessary and sufficient condition for the existence of a valid linear skewing scheme. We recall their results.

Theorem 2: Given a collection of (x,y)-lines, $\{(x_i,y_i)|i=1,2,\ldots,I\}$, there exists a valid linear skewing scheme for this collection if and only if there exists a and b such that $(ay_i+bx_i,N)^{(b)} = 1$, for $i=1,2,\ldots,I$.

Proof: It sufficies to show that for a chosen pair, a and b, any N consecutive elements of any instance of an (x,y)-line, for definiteness, say the (x_r,y_r)-line, can be accessed conflict free if and only if $(ay_r+bx_r,N) = 1$, since the linear skewing scheme must work for each (x,y)-line independently.

Suppose first that a and b are such that the linear skewing scheme, $\varphi(i,j) = ai+bj \bmod N$ is valid for the (x_r,y_r)-line. Then $ai+bj \not\equiv_N^{(c)} a(i+\nu y_r) + b(j+\nu x_r)$, for $\nu=1,2,\ldots,N-1$. This implies that $\nu(ay_r+bx_r) \not\equiv_N 0$, for $\nu=1,2,\ldots,N-1$. This occurs if and only if $(ay_r+bx_r,N) = 1$.

Conversely, suppose $(ay_r+bx_r,N) = 1$. This implies that $a(i+\nu y_r) + b(j+\nu x_r) \not\equiv_N a(i+\nu' y_r) + b(j+\nu' x_r)$, for $\nu,\nu'=0,1,\ldots,N-1$ and $\nu \neq \nu'$. This, however, is just a mathematical way of stating that no two elements amongst N consecutive elements of an arbitrary instance of the (x_r,y_r)-line are assigned to the same memory module. ■

Conflict free access to rows ((1,0)-lines) is, therefore, equivalent to the existence of a b so that $(b,N) = 1$. Similarly conflict free access to columns, forward diagonals and backward diagonals is equivalent to the existence of an a and b so that $(a,N) = 1$, $(a+b,N) = 1$ and $(-a+b,N) = 1$ respectively. From these results Budnik and Kuck [1] point out that if 100% memory utilization is desired, at the same time conflict free access to rows, columns and forward diagonals is assured, and a linear skewing scheme is employed, then $2 \nmid N$,$^{(d)}$ for one of a,b and a+b is divisible by 2. If backward diagonals are added to the collection of matrix subparts to which conflict free access is to be assured, then we must also have $3 \nmid N$. They make two further observations. Since $N = 2^q$ is precluded, by their results, for the types of algorithms they consider, the division process in performing the modular arithmetic is slowed, and no reasonably fast, but not too expensive, interconnection network is known which can sort the permutations introduced by the skewing scheme.

It is the purpose of the next two sections to prove

Theorem 3: Given a collection of three (x,y)-lines, there is a valid linear skewing scheme for this collection if and only if there is a valid periodic skewing scheme.

The only if direction is trivial. The truth of the if direction is rather surprising, since the class of periodic skewing schemes is far larger than the class of linear skewing schemes. It is not hard to construct examples of collections of three (x,y)-lines for which there are valid periodic skewing schemes which are not linear. (The theorem only claims the existence of a valid linear skewing scheme, not that all valid periodic schemes are linear.)

(b) $(x,y) = 1$ means the greatest common factor of x and y is 1. They are relatively prime.

(c) $\not\equiv_N$ means not congruent modulo N.

(d) $c|d$ means c divides d (evenly). $c \nmid d$ means c does not divide d.

III. The Existence and Non-Existence of Linear Skewing Schemes

In this section we will completely characterize those situations for which a valid linear skewing scheme exists for a collection of three (x,y)-lines. We note that Theorem 2 requires $(x_i,y_i,N) = 1$, for i=1,2,3.

Lemma 1: Given a collection of (x,y)-lines, $\{(x_i,y_i)\text{-lines}|i=1,2,\ldots,I\}$, and given N,N', a, b, a', and b' such that $(ay_i+bx_i,N) = (a'y_i+b'x_i,N') = 1$ for $i=1,2,\ldots,I$, then there exists a" and b" so that $(a''y_i+b''x_i,NN') = 1$ for $i=1,2,\ldots,I$.

Proof: We define three numbers. Let π equal the product of those primes which divide N, but do not divide N'. Let π' equal the product of those prime which divide N', but do not divide N. Finally, let ρ equal the product of those primes which divide both N and N'. In each of π,π', and ρ include a prime factor only once, even if N and/or N' include it several times. Also, let any of π, π', and ρ be equal to one if there are no prime factors out of which to constitute the product.

Let $a'' = a\pi'\rho+a'\pi$ and $b'' = b\pi'\rho+b'\pi$. We claim that $(a''y_i+b''x_i,NN') = 1$, for $i=1,2,\ldots,I$. To see this let p be a prime factor of NN'. Then p divides exactly one of π, π', and ρ. Assume $p|\pi$. The other cases are handled similarly. If $p|a''y_i+b''x_i$ for some i, then, since $p|\pi$ implies p divides $a'\pi y_i$ and $b'\pi x_i$, it also divides $a\pi'\rho y_i+b\pi'\rho x_i$. Since $p\nmid\pi'$ and $p\nmid\rho$ we have $p|ay_i+bx_i$. But $p|\pi$ implies $p|N$, so $(ay_i+bx_i,N) \neq 1$, contrary to the hypothesis of the theorem. Thus $p\nmid a''y_i+b''x_i$ for any i. But (after similar treatment for factors of π' and ρ) this implies that no prime factor of NN' is a factor of $a''y_i+b''x_i$ for any i, i.e. $(a''y_i+b''x_i,NN') = 1$ for $i=1,2,\ldots,I$. ∎

Lemma 2: Given a collection of (x,y)-lines, $\{(x_i,y_i)\text{-lines}|i=1,2,\ldots,I\}$, and given N prime, if $(x_i,y_i,N) = 1$, for $i=1,2,\ldots,I$ and $I \leq N$ then there is a valid linear skewing scheme for this collection of (x,y)-lines.

Proof: Before proceeding with the details of the proof, note that $(x_i,y_i,N) = 1$, for all i, unless there is an i for which both x_i and y_i are congruent to zero modulo N, since N is prime.

The method of proof is constructive--we claim that one of a=0,b=1, a=1,b=1, a=2,b=1, ..., a=N-1,b=1, or a=1,b=0 will have the property that $(ay_i+bx_i,N) = 1$, for $i=1,2,\ldots,I$. Consider the following table:

$$\begin{array}{ccccc}
x_1 & y_1+x_1 & 2y_1+x_1 & \cdots & (N-1)y_1+x_1 \\
x_2 & y_2+x_2 & 2y_2+x_2 & \cdots & (N-1)y_2+x_2 \\
 & & \vdots & & \\
x_I & y_I+x_I & 2y_I+x_I & \cdots & (N-1)y_I+x_I
\end{array} \qquad (1)$$

First observe that N can only divide at most one element at each row. For suppose $N|jy_i+x_i$ and $N|ky_i+x_i$. Then, assuming $k > j$, N divides their difference, $N|(k-j)y_i$. But $k-j \leq N-1$, so $N|y_i$. (Recall that by hypothesis N is a prime number. This is critical in the justification of this step of the argument.) However, if $N|y_i$, we also have $N|jy_i$ and from before $N|jy_i+x_i$, and, hence, we have $N|x_i$. But this implies $(x_i,y_i,N) \neq 1$, contrary to hypothesis. This contradiction shows that N can divide at most one element in each row.

There are N columns and only I rows. Thus, unless I = N, there will have to be a column, say the column whose elements are of the form αy_i+x_i, for which $N\nmid\alpha y_i+x_i$, for $i=1,2,\ldots,I$. In this case take $a=\alpha,b=1$. Even if I = N, such a column exists, except when each row does have an element in it that is divisible by N, and then only when they are appropriately distributed. In this case, take a=1,b=0. This choice will suffice. For suppose $(ay_i+bx_i,N) = (y_i,N) \neq 1$ for some i. Then $N|y_i$, but also N divides some element in the row of form βy_i+x_i, $\beta=0,1,\ldots,N-1$, say $N|jy_i+x_i$. These imply $N|x_i$, again contradicting the hypothesis.

Thus we see that an a and b exist such that $(ay_i+bx_i,N) = 1$, for $i=1,2,\ldots,I$. ∎

Corollary: Given N and a collection of two (x,y)-lines, $\{(x_1,y_1)\text{-line}, (x_2,y_2)\text{-line}\}$, there is a valid linear skewing scheme if and only if $(x_i,y_i,N) = 1$, for i=1,2.

Proof: Lemma 1 shows that we need concern ourselves only with the existence of valid linear skewing schemes when N is replaced by its prime factors. Since the collection has only two elements, Lemma 2 guarantees that a valid linear skewing scheme exists for this collection for any prime. ∎

We now can completely characterize the situation for which a collection of three (x,y)-lines has a valid linear skewing scheme.

Theorem 4: Given N and a collection of three (x,y)-lines, there is a valid linear skewing scheme for the collection if and only if $(x_i,y_i,N) = 1$, for i=1,2,3 and it is not the case that $2|N$ and, after suitable renumbering of the subscripts, x_1 is odd, y_1 is even, x_2 is even, y_2 is odd, x_3 is odd, y_3 is odd.

Proof: The need for the condition $(x_i,y_i,N) = 1$, for $i=1,2,3$, was mentioned at the start of this section. As in the proof of the immediately preceding corollary, Lemma 1 implies we need only concern ourselves with the prime factors of N and Lemma 2 implies that only the factor 2 (if indeed 2 is a factor of N) can cause any difficulty. Thus the whole problem comes down to finding an a and b for which $(ay_i+bx_i,2) = 1$, for $i=1,2,3$. If all the y_i are odd we may choose $a=1,b=0$. If all the x_i are odd we may choose $a=0,b=1$. Thus we have accounted for all cases except the following: After suitably renumbering the subscripts, we have y_1 even and, hence, x_1 odd (since $(x_i,y_i,N) = 1$ implies $(x_i,y_i,2) = 1$, y_1 even implies, therefore, x_1 is odd), x_2 even and y_2 odd, and the nature of x_3 and y_3 is yet to be determined. Since x_3 and y_3 cannot both be even, we have either (i) x_3 odd, y_3 even, (ii) x_3 even, y_3 odd or (iii) x_3 odd, y_3 odd. In case (i) and (ii) we may choose $a=1,b=1$. If case (iii) is applicable there is no choice of a and b that is acceptable, for one of ay_i+bx_i for $i=1,2,3$, will be even, and hence, not relatively prime to 2. This is precisely the case, specified in the theorem, when a valid linear skewing scheme cannot be obtained. ■

IV. Ineffectiveness of More General Skewing Schemes

In this section we will show that for a collection of three (x,y)-lines, if linear skewing schemes are precluded, then so are periodic skewing schemes. This will complete the proof of Theorem 3. Our first observation is that the condition $(x_i,y_i,N) = 1$, for all i, must remain. Suppose that $(x_i,y_i,N) = s > 1$. Take any matrix element, say $a_{c,d}$. Then $a_{c+\frac{N}{s}y_i,\ d+\frac{N}{s}x_i}$ and $a_{c,d}$ are in the same block of N consecutive elements of an instance of an (x_i,y_i)-line. But $s|y_i$ and $s|x_i$, so $a_{c+\frac{N}{s}y_i,\ d+\frac{N}{s}x_i}$ is just $a_{c+Nj,d+Nk}$. If the skewing scheme is periodic then $\varphi(c,d) = \varphi(c+Nj,d+Nk)$, so these elements are stored in the same memory module, and, therefore, φ is not a valid skewing scheme.

In order to prove our assertion about the ineffectiveness of periodic skewing schemes, it is convenient to give an (x,y)-line a new geometric interpretation. The defining property of periodic skewing schemes means that every $N \times N$ submatrix (with left and upper edge on a multiple of N) has exactly the same memory map. We are thus able to restrict our attention to one $N \times N$ submatrix, and in considering N consecutive elements of an instance of an (x,y)-line, when the operation of "going over x, and down y" takes us outside the bounds of the $N \times N$ submatrix, we just "wrap around" to the opposite edge. Figure 4 gives an example of this. Throughout this section "(x,y)-line" will mean an "(x,y)-line under this wrap around interpretation." Notice that the condition $(x,y,N) = 1$ partitions the $N \times N$ array into N distinct instances of the (x,y)-line, each with N elements.

Lemma: Given an (x,y)-line, with $(x,y,N) = 1$, each of the N instances of the (x,y)-line can be characterized by an integer in $\{0,1,\ldots,N-1\}$ in the following manner: If $a_{w,z}$ is an element of an instance of the (x,y)-line, characterize this instance by $xw-yz \bmod N$.

Proof: Our characterization is a function from instances of the (x,y)-line to $\{0,1,\ldots,N-1\}$. We must show that this definition is independent of the choice of the representative element. Suppose that $a_{w,z}$ and $a_{w',z'}$ lie on the same instance of the (x,y)-line. Then $w' \equiv_N w+vy$ and $z' \equiv_N z+vx$, the modular N arithmetic resulting from wrapping around. Then $xw'-yz' \equiv x(w+vy)-y(z+vx) \equiv xw+xvy-yz-yvx \equiv xw-yz$. Thus the function is independent of the choice of representative.

Furthermore, the function is onto, i.e. different instances of the (x,y)-line are characterized by different integers. To see this it suffices to show that for any i there exists a w and a z for which $xw-yz \equiv i$, for then the instance of the (x,y)-line containing $a_{w,z}$ will be characterized by i. It is a well-known result of elementary number theory that given x and y, there exists c and d such that $xc-yd = (x,y)$. Since $((x,y),N) = (x,y,N) = 1$ and the residue classes of numbers relatively prime to N form a group under multiplication, there exists g such that $(x,y)\cdot g \equiv 1$. Hence $xcg-ydg \equiv 1$ and thus $xcgi-ydgi \equiv i$. Letting $w = cgi \bmod N$ and $z = dgi \bmod N$ gives us the needed $a_{w,z}$. ■

The proof of the following theorem uses a generalization of a technique used by Polya [5] in solving the recreational mathematics puzzle of placing N "super queens" on an $N \times N$ chessboard.

Theorem 5: Given three (x,y)-lines, with $(x_i,y_i,N) = 1$, for $i=1,2,3$, if $2|N$ and, after suitably renumbering the subscripts, x_1 is odd, y_1 is even, x_2 is even, y_2 is odd, x_3 is odd, and y_3 is odd, then there is no valid periodic skewing scheme for this collection of (x,y)-lines.

Proof: The proof is by contradiction. Suppose a valid skewing scheme exists. Then exactly N of the

N^2 elements in the $N \times N$ array must be stored in memory zero, for one and only one element from each instance of the (x_1,y_1)-line must be stored in memory zero if we are to avoid a memory conflict. Call these elements (u_i,v_i) for $i=0,1,\ldots,N-1$. Furthermore if conflict free access to (x_2,y_2)-lines and (x_3,y_3)-lines is to be assured each of these N elements must also lie on a different instance of these two (x,y)-lines. Thus the elements (u_i,v_i) can be used as representatives to characterize the instances of the (x,y)-lines. The preceding lemma, therefore, tells us that $\{x_ju_i-y_jv_i \bmod N \mid i=0,1,\ldots,N-1\} = \{0,1,\ldots,N-1\}$ for $j=1,2,3$. This allows us to conclude that $\sum_{i=0}^{N-1} x_ju_i-y_jv_i \bmod N = \sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}$. Since $2|N$ we have $\frac{N(N-1)}{2} \equiv_N \frac{N}{2}$. Thus

$$\sum_{i=0}^{N-1} x_ju_i-y_jv_i \equiv_N \frac{N}{2} \quad \text{for} \quad j=1,2,3 \tag{2}$$

Consider next the set of linear equations:

$$\begin{aligned} ax_1+bx_2 &= x_3 \\ ay_1+by_2 &= y_3 \end{aligned} \tag{3}$$

this system has solution $a = \frac{x_3y_2-x_2y_3}{x_1y_2-x_2y_1}$, $b = \frac{x_1y_3-x_3y_1}{x_1y_2-x_2y_1}$. Notice that the solution is valid, since $x_1y_2-x_2y_1$ is odd, and hence not zero. Setting $c = x_3y_2-x_2y_3$, $d = x_1y_3-x_3y_1$ and $e = x_1y_2-x_2y_1$ we have

$$\begin{aligned} cx_1+dx_2 &= ex_3 \\ cy_1+dy_2 &= ey_3 \end{aligned} \tag{4}$$

Therefore

$$\sum_{i=0}^{N-1} (cx_1u_i-cy_1v_i) + \sum_{i=0}^{N-1} (dx_2u_i-dy_2v_i) \equiv$$

$$\sum_{i=0}^{N-1} (cx_1+dx_2)u_i - (cy_1+dy_2)v_i \equiv \sum_{i=0}^{N-1} ex_3u_i-ey_3v_i \equiv$$

$$e \sum_{i=0}^{N-1} x_3u_i-y_3v_i \equiv e\,\frac{N}{2},$$ this last by (2).

Also, however, $\sum_{i=0}^{N-1} (cx_1u_i-cy_1v_i) + \sum_{i=0}^{N-1} (dx_2u_i-dy_2v_i) \equiv$

$$c \sum_{i=0}^{N-1} x_1u_i-y_1v_i+d \sum_{i=0}^{N-1} x_2u_i-y_2v_i \equiv c\frac{N}{2}+d\frac{N}{2} \equiv (c+d)\frac{N}{2}$$

Combining these last two formulae, we have $e\,\frac{N}{2} \equiv (c+d)\frac{N}{2}$.

Because of the odd/even nature of the x_i and y_i, c, d, and e are all odd. This implies $e\frac{N}{2} \equiv \frac{N}{2}$ and $(c+d)\frac{N}{2} \equiv 0$. Thus $\frac{N}{2} \equiv 0$, a contradiction. This contradiction arose from assuming the existence of a valid periodic skewing scheme. Thus the theorem is proven. ■

Coupled with Theorem 4, this proves Theorem 3. Thus we see that for collections of three (x,y)-lines, periodic skewing schemes do not provide additional power over linear skewing schemes. In certain cases it is possible to go even further.

Lemma: If a valid skewing scheme exists for a collection of (x,y)-lines, and the collection includes both rows ($(1,0)$-lines) and columns ($(0,1)$-lines), then the skewing scheme is periodic.

Proof: We show how the presence of $(1,0)$-lines implies $\varphi(i,j) = \varphi(i,j+N)$. For the skewing scheme to be valid $\varphi(i,j),\varphi(i,j+1),\ldots,\varphi(i,j+(N-1))$ must all be different. Similarly, $\varphi(i,j+1)$, $\varphi(i,j+2),\ldots,\varphi(i,j+N)$ must all be different. Since N-1 of the elements are the same, and the values of φ are fixed, there is no choice but to have $\varphi(i,j) = \varphi(i,j+N)$. The remainder of the proof is obvious. ■

As pointed out earlier Budnik and Kuck [1] showed that if $2|N$ then there is no linear skewing scheme which permits conflict free access to rows, columns and forward diagonals. The results presented here imply that if $2|N$ there is no skewing scheme of any type which permits conflict free access to these common matrix subparts. Results on collections of four or more (x,y)-lines can be found in the author's Ph.D. dissertation [6]. A result of practical significance is: If $2|N$ or $3|N$, then there is no valid skewing scheme for the collection of (x,y)-lines consisting of rows, columns, forward diagonals, and backward diagonals.

V. Implication for Research on Related Problems

The practical problem of storing large matrices in N independent memory modules so that we simultaneously have 100% memory utilization and conflict free access to rows, columns, forward diagonals, and (optionally) backward diagonals requires a number of processors and memories not a power of 2. Furthermore, in selecting the number of processors and memories only linear skewing schemes need to be considered. We, therefore, need only design memory processor interconnection networks which can sort p-ordered vectors (Swanson [7]) with arbitrary shifting. The use of non-linear skewing schemes, with a resultant rise in the complexity of the vectors we need to be able to sort, does not appear to yield any benefits. The problem of sorting p-ordered vectors with arbitrary shifting under certain time and hardware restrictions has proved to be a most difficult problem. Swanson [7] has proposed the use of k-apart shifters to sort p-ordered vectors when N is prime, but without the arbitrary shifting needed to sort the vectors introduced by linear skewing schemes. Lawrie's Ω-networks are at their best when $N = 2^q$. Unfortunately, our results show that N will have 5 as its smallest prime factor. If N is prime Lawrie's Ω-network reduces to an $N \times N$ crossbar. Hopefully a network of $cN\log N$ (c small) switches, that operates in time $\log N$, and sorts p-ordered vectors with arbitrary shifting can be developed.

Acknowledgment

The author wishes to express his sincere thanks to D. H. Lawrie, D. J. Kuck and C. L. Liu for their many helpful suggestions.

References

[1] Budnik, P. and D. J. Kuck, "The Organization and Use of Parallel Memories," IEEE Transactions on Computers, Vol. 20, December 1971, pp. 1566-1569.

[2] Chandra, A. K., "Independent Permutations, as Related to a Problem of Moser and a Theorem of Polya," Journal of Combinatorial Theory, Series A, Vol. 16, 1974, pp. 111-120.

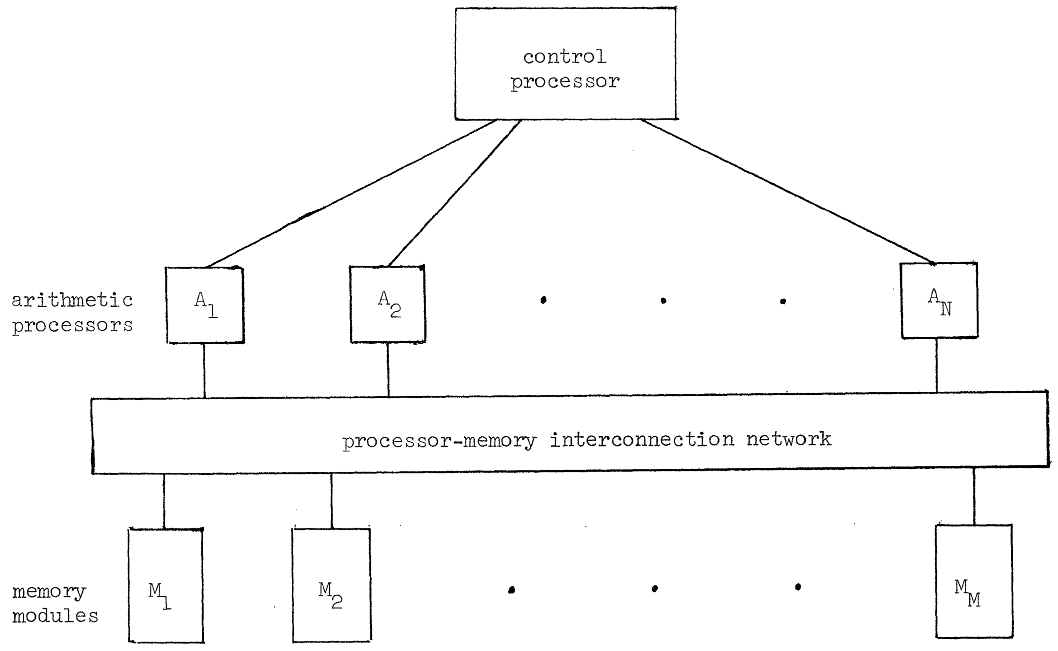
[3] Hardy, G. H. and E. M. Wright, An Introduction to the Theory of Numbers, Oxford University Press, London; 1954.

[4] Lawrie, D. H., "Memory-Processor Connection Networks," Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. UIUCDCS-R-75-557; February 1973.

[5] Polya, G., "Uber die 'doppelt-periodischen' Losungen des *n*-Damen-Problems," in W. Ahrens, Mathematische Unterhaltungen und Spiele, Teubner, Leipzig, 1918, pp. 364-371.

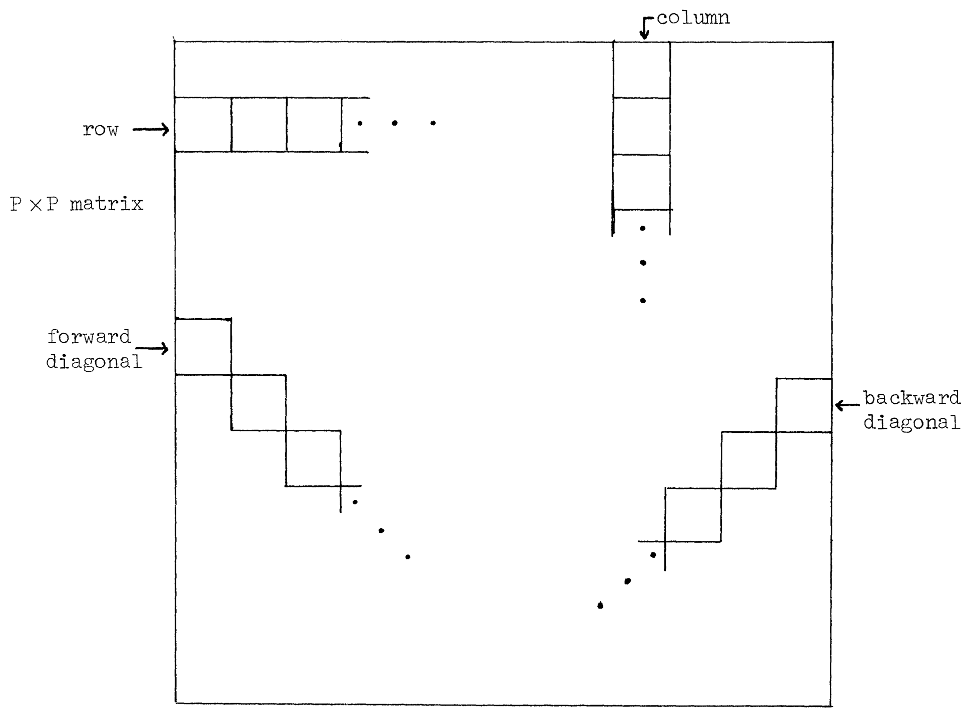
[6] Shapiro, H. D., "Theoretical Limitations on the Use of Parallel Memories," Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (to appear).

[7] Swanson, R. C., "Interconnections for Parallel Memories to Unscramble *p*-Ordered Vectors," IEEE Transactions on Computers, Vol. C-20, November 1974, pp. 1105-1115.

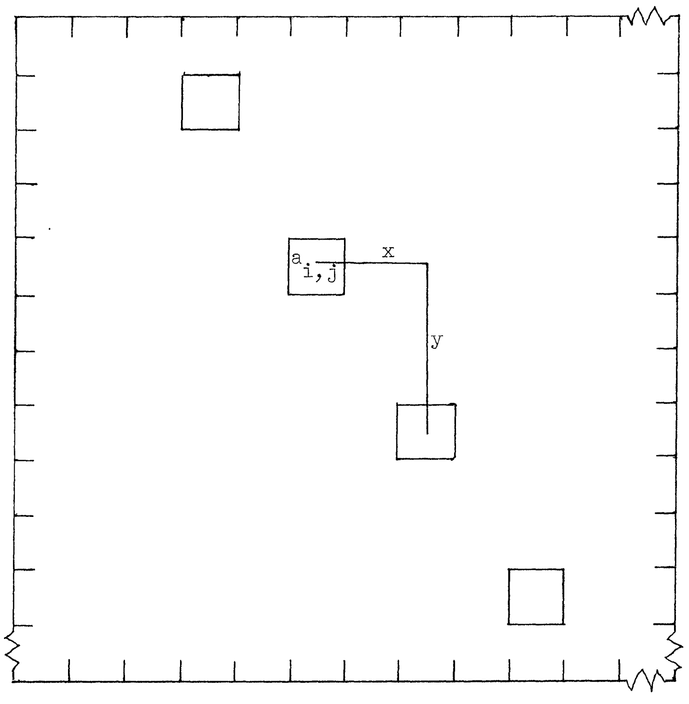


Model of a Parallel Computer

Figure 1



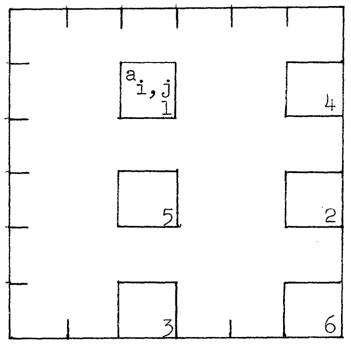
Common Matrix Subparts
Figure 2



An Example of an Instance of an (x,y)-line Containing Matrix Element $a_{i,j}$

$i = 4, \quad j = 5, \quad x = 2, \quad y = 3$

Figure 3



An Example of an Instance of an (x,y)-line Under the Wrap Around Interpretation. The Order in Which the Elements are Generated is Indicated on the Figure.

$i = 1, \quad j = 2, \quad x = 3, \quad y = 2, \quad N = 6$

Figure 4

Summary*

THE MULTI-DIMENSIONAL ACCESS MEMORY IN STARAN

By Kenneth E. Batcher

Digital Technology Department
Goodyear Aerospace Corporation
Akron, Ohio

Each array module in the STARAN+ associative array processor contains a 256 × 256 multi-dimensional access (MDA) memory (see illustration). Parallel vector arithmetic and associative search operations access memory data by bit-slices, while input, output, and scalar arithmetic operations access memory data by words. The MDA memories use standard random-access memory (RAM), integrated-circuit chips in a novel configuration. Use of standard, high-volume, low pin-count memory devices in place of custom LSI devices reduces costs significantly.

To achieve multidimensional access, data are stored in a scrambled pattern; bit B of word W is stored in bit-location B of memory chip B ⊕ W where ⊕ indicates a component-by-component exclusive-or.

Data are accessed by specifying a stencil shape with an 8-bit access mode and a stencil position with an 8-bit global address. The 256 memory bits covered by a stencil can be fetched or stored in one memory cycle.

The address bus structure of the MDA memory has 16 address lines (as opposed to 8 lines for a conventional RAM). For k = 1, 2, . . ., 8 address line x_k is fed by the kth bit of the global address, while address line y_k is fed by the exclusive-or of the kth bits of the access mode and the global address. Address pin k of memory chip $(c_1 c_2 \ldots c_8)$ is connected either to x_k if $c_k = 0$ or to y_k if $c_k = 1$.

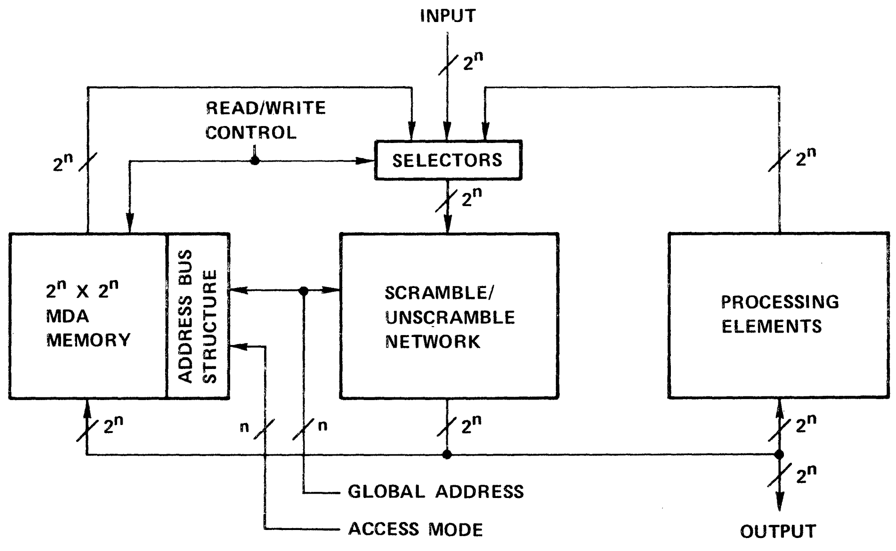
Memory data are scrambled and unscrambled by a scramble/unscramble network, which can also shift and perform other useful permutations on data fetched from memory.

When memory data are fetched or stored with access mode M and global address G, processing element P accesses bit $(\overline{M} \cdot G) \oplus (M \cdot P)$ of memory word $(M \cdot G) \oplus (\overline{M} \cdot P)$, where logical negation is indicated by "¯" and the logical product ("and") is indicated by "·".

Bit-slice access is obtained with M = (00000000) and word access is obtained with M = (11111111). Other access modes allow data to be accessed in other ways.

*This is a summary of a paper that has been submitted for publication in the IEEETC Special Issue on Parallel Processing

+TM, Goodyear Aerospace Corporation, Akron, Ohio



Block Diagram of STARAN Array Module

POLYAUTOMATON DESIGN FOR RECOGNIZING CERTAIN L SYSTEM LANGUAGES BY PARALLEL COMPUTATION

Carl F. R. Weiman
Computer Science Dept.
Courant Institute
New York University
251 Mercer Street
New York, N.Y. 10012

and

Jerome Rothstein
The Ohio State University
Department of Computer
and Information Science
2036 Neil Ave.
Columbus, Ohio 43210

Abstract -- The theory of formal languages specified by sequential rules, e.g. Chomskian grammars for generation and various automata for recognition, has contributed much to computer science. The theory of formal languages specified by parallel rules, e.g. Lindenmayer systems, is too young yet to have contributed to parallel computer design, but there are signs that it will. Languages generated by parallel rules are often of more general Chomskian type than those generated by similar sequentially applied rules. Consequently sequential recognition machinery is slow. A new polyautomaton design for recognizing certain L systems by parallel computation is described. The flexibility of its control structure suggests generalization to other kinds of parallel computation.

Introduction

The theory of formal languages specified by sequential rules, e.g. Chomskian grammars for generation and various automata for recognition, has contributed much to computer science. For example, BNF provides machine independent programming language specification and automata theory is useful in describing algorithm complexity. The theory of formal languages specified by parallel rules, e. g. Lindenmayer systems (L systems) [1,2] is too young yet to have contributed to the design of parallel computers. Among the signs that it will are that L systems constitute an abstract representation of discrete (digitized) parallel processes independent of physical realization. Parallelism in processing digitized pictures and computer modelling of many physical and biological phenomena is much more natural than serialism [3]. The translation of algorithms from serial to parallel form is not straightforward, however, and there are many implications for machine realizations from the new parallel theory that are not clearly related to results in the better known serial theory [4,5]. Most L system papers concern the generating of languages; the problem of recognition is less well represented. One difficulty with the latter is that most languages generated by parallel rules are of more general Chomskian type than those generated by similar sequentially applied rules. For example, context free parallel rules can generate languages which require context sensitive rules for sequential generation. A result is that sequential recognition machinery(LBA in this case) is slow and complex, not lending itself easily to parallelism. A new kind of polyautomaton [6] which recognizes certain L system languages is described here. Its mode of operation is highly parallel and the control structure provides enough flexibility to suggest generalization to other kinds of parallel computation.

Polyautomaton Design

In the following example, the L system whose language is to be recognized by polyautomaton is a TOL system with alphabet (0,1) and initial word 1. The tables are:

$$L1: (0 \rightarrow 0,\ 1 \rightarrow 01)\ ,\quad L2: (1 \rightarrow 1,\ 0 \rightarrow 01)$$

All productions from a single table are to be applied simultaneously to all symbols in a string. Words of the resulting language constitute a coding for the best stepwise approximations of straight line paths of arbitrary slope on grids. The symbol 0 corresponds to a unit step along a grid axis and the symbol 1 along a diagonal. Among the code's applications are digitized line generation and geometric transformation in computer graphics and straight line detection in pattern recognition, all via parallel computation. Its very interesting geometric and computational properties are described in [7] and [8] but will not be discussed here. In the Chomskian hierarchy the language is context sensitive and can be recognized by an LBA with thirteen states using four symbols. The polyautomaton recognition process to be described roughly consists of reversing the productions of the L system tables until the initial word results. However additional machinery is needed to recognize appropriate contexts for rule applications, decide which table to use, and detect accept or reject conditions. The structure of this controlling machinery is a departure from most polyautomata designs; it results in greater speed and flexibility. Details follow.

The general polyautomaton design consists of a grid of identical finite state automata. Initially, all are in state D (dead) except for those constituting the word (pattern) to be recognized. The state transition of each automaton (cell) is determined by its own and its nearest neighbors' states. There are several possible transition functions. That which prevails at any time is determined solely by the state of a finite state controlling automaton (CS) and is the same for all cells. CS communicates its state to all cells not in state D, acting as a synchronizing clock and programmer. Communication from cells to CS occurs via a "logic bus" and determines CS transitions as follows.

Each cell sends a boolean n-tuple which carries information about its own and its neighbors' states. Considering each n-tuple as indexed by the coordinates of the cell sending it, the CS computes a set of boolean functions over the set of n-tuples symmetric with respect to that index. That function in the set whose boolean value is 1 determines the state transition of CS which in turn determines cell behavior in the next time step. Intuitively, boolean symmetry over cell indices means that CS responds to combinations of messages from cells independent of the locations or populations of sets of cells sending them. This mode of operation permits interaction between cells and CS; its significance will be clarified in the detailed description which follows.

Initially the non-dead cells constitute a horizontal row of adjacent cells whose states are the symbols of the word to be recognized. Only the left nearest neighbor of any cell will be used in determining cell state transitions. Table 1 describes the two cell state transition functions corresponding to CS states X and W. The left column gives the neighbor's state (dc means don't care), the middle column gives initial and final state, and the right column shows the boolean n-tuple (a 2-tuple or pair in this case) that is sent to CS.

Left Neighbor	Cell State Transition	to CS	
0	$0 \rightarrow 0$	01	For CS state X
1 or D	$0 \rightarrow C$	00	
0	$1 \rightarrow 1$	00	
1	$1 \rightarrow 2$	10	
dc	$C \rightarrow C$	00	
dc	$1 \rightarrow 0$	10	For CS state W
dc	$2 \rightarrow 1$	01	
dc	$C \rightarrow C$	00	

Table 1. Cell Transition Functions

Table 2 describes the CS transition function. States S and F are halting states for success and failure (accept and reject) respectively. In the left column the variables x and y represent respectively the first and second digits of the boolean 2-tuples; the implicit indices are dropped but understood to constitute the range over which the boolean sums (Σ) and products (Π) are evaluated. Overbar means boolean complement and all operations are boolean. The middle column gives initial and final states. The text in the right column describes the situation which yields the value 1 for the function in the left column.

Boolean function	CS state transition	Condition of cells
$\Pi\overline{x} \cdot \Sigma y$	$X \rightarrow X$	Unerased 0's exist and no 1's are next to 1's
$\Pi\overline{x} \cdot \Pi\overline{y}$	$X \rightarrow S$	No 0's, one 1 exists
$\Sigma x \cdot \Sigma y$	$X \rightarrow F$	A 1 is next to a 1 while unerased 0's exist
$\Sigma x \cdot \Pi\overline{y}$	$X \rightarrow W$	All 0's are erased and a 1 is next to a 1.
$\Sigma x \cdot \Sigma y$	$W \rightarrow X$	1's and 2's exist
$\Pi\overline{x} \cdot \Sigma y$	$W \rightarrow F$	No 1's, only 2's exist

Table 2. CS transition function.

Consider the behavior of the polyautomaton in recognizing the string 00101. CS starts in state X, so the first cell transition yields the string C01C1 whose constituent cells send a mixture of 00 and 01 messages to CS. Cells in state C act as communication channels between their nearest neighbors. Thus, conversion to state C corresponds to erasing a symbol in the string. The result here is identical to reversing the productions of L system table L1. In the CS transition table, the only boolean function whose value is 1 is that which makes CS stay in state X, so the cells again respond to yield the string CC1C2, sending a mixture of 00 and 10 messages to CS. The latter responds by going into state W, causing the cell transition to CC0C1. The cells send the messages 00, 01 and 10 to CS which goes into state X. The last two cell transitions correspond to reversing the productions of L system table L2. Next, the cell string becomes CCCC1, which is recognized as the initial word of the L system, causing acceptance which consists of CS transition into state S.

Conclusion

More general machines which operate on two dimensional neighborhood patterns on grids have been designed [5,7,8]. Among their capabilities are the recognition of straight line paths, edges (boundaries), connected regions, and polygonal approximations to general curves. The design of polyautomata to recognize Dyck languages and languages of the type $A^nB^nC^n...Z^n$ became straightforward. Their construction has been accomplished explicitly by J. Rothstein and J. M. Moshell and is incorporated in the doctoral dissertation of the latter. These and many other languages can be recognized "immediately" by parallel computation. Properties of the straight line code suggest incorporation into a polyautomaton with parallel numerical computation capabilities such as continued fraction manipulations and linear transformations.

The general polyautomaton design is not presented here as a universal recognizer of L systems though recent work suggests that this is possible. The design treated in this paper has important practical consequences for the speed and complexity of algorithms for certain parallel computations. One of these principles is the use of conducting cells[(a)], which could have been avoided by using propagating states such as those used in von Neumann's self-reproducing automaton [9]. As longer strings of such cells arise, most of the machine's time would be wasted in the trivial propagation of information that requires only one time step using conducting cells. A more important principle is the nature of the cell-CS interaction via the logic bus. It permits a division of labor in which the cells process local information and the CS global information. This frees cells from the burden of keeping track of the states of distant neighbors and the time delays necessary for such communication which iterative automata suffer from [6] . To use a programming analogy, CS acts as a monitor rather than a user program.

The logic bus information processing by CS frees it from the exponential network and computational complexity of perceptrons and nerve nets which must somehow account for large numbers of exact configurations of cells in unbounded neighborhoods. It is precisely this exponential complexity which is absorbed into the symmetric boolean functions which determine CS transition. For k cells, the number of distinct sets of k n-tuple messages (indexed by cell location, for example) is $(2^n)^k$. The number of elements of this set distinguishable by the kind of symmetric boolean function over k messages used here is only 3^n, however (i.e., for any column of the n-tuple, only the threefold distinction between: 1)All 0's, 2)All 1's, and 3)0's and 1's, can be made). The exponent n here was 2, so the number of boolean functions is small and more important, independent of the number of cells in the pattern and their coordinates. The logic bus poses no engineering problems. It is realizable as a simple series-parallel switching network constructed by joining simple identical elements.

(a) This concept was generalized by Rothstein to that of the bus automaton. This is a cellular automaton which has, in each cell, the capability of activating conducting channels through it, whereby not only do non-adjacent neighbors communicate directly, but arbitrary connection patterns can be established between cells wherever they may be. A complete formalization and many applications will be found in Moshell's dissertation and their joint papers [5] .

References

[1] A. Salomaa, Formal Languages, Academic Press, (1973).

[2] G. T. Herman and G. Rozenberg, Developmental Systems and Languages, Am. Elsevier,(1975).

[3] C. D. Stamopoulos, "Parallel Image Processing", IEEE Trans. Comput., vol. c-24, (April, 1975), pp. 424-433.

[4] J. F. Traub (Ed.), Complexity of Sequential and Parallel Numerical Algorithms, Academic Press, (1973).

[5] J. Rothstein and J. M. Moshell, Papers in preparation,and J. M. Moshell, Parallel Recognition of Formal Languages by Cellular Automaton, Ph. D. dissertation, the Ohio State University Department of Computer and Information Science, Columbus, Ohio, (1975).

[6] A. R. Smith, "Introduction and Survey of Polyautomata Theory", introduction to german translation of von Neumann's Theory of Self-Reproducing Automata, Burks, A. W. (Editor), Rogner and Bernhard GmbH., Munich, (1975).

[7] J. Rothstein and C. F. R. Weiman, "Parallel and Sequential Specification of a Context Sensitive Language for Straight Lines on Grids", Computer Graphics and Image Processing,(to appear early 1976).

[8] C. F. R. Weiman and J. Rothstein, Pattern Recognition by Retina-Like Devices, Computer and Information Science Dept., Ohio State U., OSU-CISRC-TR-72-8 (AD 214 665/2), (1972).

[9] A. W. Burks, Essays on Cellular Automata, Univ. of Illinois Press, (1970).

SIMPARAG - Simultaneous Parallel Array Grammars *

Patrick Shen-Pei Wang and William I. Grosky
School of Information and Computer Science
GEORGIA INSTITUTE OF TECHNOLOGY
Atlanta, Georgia 30332

Summary

In recent years several research efforts have been aimed at the generalization of phrase-structure grammars[1,4] whose rewriting rules allow the replacement of a subarray of a picture with another subarray[6,8]. The first example of such a grammar is described by Kirsch[5] for generating a class of labeled 45^o right triangles. A similar formal system is developed by Dacey[3] and grammars for languages consisting of classes of polygons are exhibited. A general survey of this area of picture languages is given by Miller and Shaw[7] .

A possible modification to the definition of a derivation in a picture grammar is parallel rule application, i.e. all instances of the rule's antecedent are simultaneously replaced by the consequent(rather than just one instance). The advantage of such modification is that, for a given language, a grammar that operates in parallel may be much simpler to write than one that operates sequentially [8]. This is especially natural in two-dimensional case since local picture operations are often applied to digital pictures in parallel[8].

In this paper the effort is aimed not only at the parallelism of grammars but also at the simultaneous properties of generating rules. If in the process of derivation two or more rules can be applied simultaneously, it might speed up the derivation and hence save computation time. This is one of the motivations of this research. Another is the deterministic property which may enhance the efficiency of the derivation and hence reduce the hardware complexity in dealing with pattern processing by computers. Moreover, it may provide some fundamentals of generalized Lindenmayer Systems[9] which are more natural for biological applications.

We first establish a model AG for parallel pattern processing in arbitrary dimensions. In this model each grammar G is a quintuple $G=(V_N,V_T,P,S,\#)$ in which the set of generating rules is represented as $P=\{\ \alpha^r \rightarrow \beta^r \mid r\varepsilon \mathcal{R}\}$where $\mathcal{R}$ is the set of labels of rules and α^r and β^r are the left and right hand side arrays respectively. The maximal quadruple extension of the minimal prism(MQEMP) in terms of each rule is defined. It is shown[11] that there is a unique domain of MQEMP denoted by dom $\mathbb{P}_Q(G)$ for each grammar G and in dom $\mathbb{P}_Q(G)$ there is an unambiguous center which can be treated as the origin of the neighborhood index or template of a corresponding cellular structure - another well known parallel device, the familiarity with which is assumed[2,10].

Next, the meanings of intersection (denoted by $\cap$) and union (denoted by $\cup$) of finite array patterns are defined in terms of the overlaying domains and overlaying arrays. The concepts of 'applicable in parallel' and 'simultaneously applicable'rules are introduced. The sufficient conditions for rules to be applicable in parallel and simultaneously applicable are exhibited repectively.

We then give a formal definition of 'SIMPARAG' - simultaneous parallel array grammars and it is shown that the sufficient condition for a grammar in AG to be a simparag is that $\alpha^i \cap \beta^j = \emptyset \ \forall \ i \neq j$, $i,j \varepsilon \mathcal{R}$ and $\alpha^i \cap \beta^j = \{\alpha^i\} \forall \ i \varepsilon \mathcal{R}$. Under such conditions any intermediate array(sentential form) can be partitioned into several mutually exclusive subarrays and the derivation process is independent of the order of the rules' selection. Furthermore it is shown that for each simparag $G=(V_N,V_T,P,S,\#)$ we can find a deterministic cellular structure $Z=(Q,I^d,X, \Pi,q_o,\tilde{C}_o)$ equivalent to it.

Finally, some necessary conditions for an AG to be a simparag are discussed. It is pointed out that one of the difficulties to derive necessary conditions is that we don't know whether $\alpha^i \cup \beta^j$ will appear as a part of of any intermediate array. This is, perhaps due to the 'unpredictability' of the derivation of generating rules.

References

[1] S. Abraham,"Some Questions of Phrase Structure Grammars I",Comp. Ling. v.4(1965), 61-70
[2] A.W.Burks,Essays on Cellular Automata, U.of Ill Press, (1970)
[3] M.F.Dacey,"A 2-D Languages for a Class of Polygons",Pattern Recognition,v.3,(1971),197-208
[4] R.Y.Kain,Automata Theory:Machines and Languages Mc Graw Hill Press,(1972)
[5] R.A.Kirsch,"Computer Interpretation of English Text and Picture Patterns",IEEE Tran., v.EC-13 (Aug. 1964), 363-374
[6] D.L.Milgram and A Rosenfeld,"Array Automata and Array Grammars",Info. Proc.,(1971), 69-74
[7] W.F.Miller and A.C.Shaw,"Linguistic Methods in Picture Processing", Proc.AFIP,(1968),v33,279-290
[8] A.Rosenfeld and A.Mercer,"An Array Grammar Programming System",CACM,v.11(1973),299-305
[9] A.Salomma,Formal Languages,Academic Press(1973) §13 Lindenmayer Systems,234-252
[10] R.A.Smith,"Cellular Automata and Formal Languages",IEEE Symp.on Swit.and Auto.Th.(1971)
[11] P.S.P.Wang and W.Grosky,"The Relation Between Uniformly Structured Tessellation Automata and Parallel Array Grammars", Proceedings of ISUSAL, Tokyo, Japan,(1975), To appear

*This work is supported in part by NSF Grant GN-655.

PRELIMINARY RESULTS OF A COMPARATIVE ANALYSIS OF ILLIAC IV LANGUAGES

Dr. Robert L. Milton
Computer Resources Group
R & D Associates
Marina del Rey, California 90291

ABSTRACT

This report presents the preliminary results of comparative analysis of the three higher level languages currently available for the ILLIAC IV. Two of these languages, CFD and IVTRAN, have FORTRAN like syntaxes, while the third, GLYPNIR, has an ALGOL like syntax. For the comparison FORTRAN programs were rewritten in each of the languages. The comparison was based primarily on two factors--programming effort (measured mainly by the number of source statements) and language utilization (measured mainly by the time required to execute a standard problem). None of these languages proved to be outstanding in these results. CFD appears to give the shortest execution times, GLPYNIR appears to require the least programming effort, and IVTRAN was intermediate in both regards.

INTRODUCTION

One of the hurdles facing the prospective user of the ILLIAC IV system is the decision as to which of the currently available programming languages should be used. For the user desiring a higher level language than ASK, the ILLIAC IV assembly language, there are three languages from which to choose. These are [1] GLYPNIR, a high-level, block-structured language that resembles ALGOL; [2] CFD, a moderate-level language having some resemblance to FORTRAN; and [3] IVTRAN, a high-level language which has the basic features of FORTRAN with extensions for the ILLIAC IV.

CRITERIA FOR COMPARISON

In order to make direct and fair comparisons of the three higher-level ILLIAC IV languages, several programs originally written in FORTRAN for a serial computer were chosen for recoding as equivalent programs in each of the three languages. The FORTRAN programs chosen for the benchmark study were representative of several classes of problems that should be well suited to the ILLIAC IV. These programs included one and two dimensional hydrodynamics codes, two Fourier Transform Codes (one for transforming multiple, independent data sets; one for transforming a single, large data set), and several matrix manipulation codes. This report presents preliminary results for the two hydrodynamics programs. Work on the other codes is still in the preliminary development stage.

Two main criteria were used in the comparison and analysis of the languages--the efficiency of machine utilization and the programming effort required. Efficiency of machine utilization is more important for the ILLIAC IV than for a serial computer. The ILLIAC IV's potential execution speed can only be achieved with machine code that efficiently utilizes its architecture. In order to judge the quality of machine code generated by each of the languages, two factors were considered. First, was the execution time required to run a standard problem. Second, was the amount of machine code generated for each of the standard programs. This second factor, which will show some correlation to the execution time, was considered since the core-size of the ILLIAC IV may be a limitation for a large code.

The effort involved in converting a program to a new machine is often a more important consideration than the machine time involved. This is particularly true in the case of the ILLIAC IV in which architecture and languages are likely to be unfamiliar for a new user. Thus, we consider programming effort to be as important as machine utilization in comparing these languages.

The number of statements required to produce an equivalent version of the original FORTRAN codes was the main factor used in comparing the programming effort required for each of the languages. Features supported by the languages have an impact on the number of statements required. Particularly important features are the functions supplied and the I/O handling. One factor not taken into account in the comparison is the fact that both CFD and GLYPNIR allow insertion of ASK statements inline anywhere in a code, a potentially valuable feature not shared by IVTRAN, though it does allow subroutines written in ASK. This feature was not utilized in the comparison since we wanted to compare the languages themselves and not the programmer's ability to write good assembly language codes.

ZAP

The first program used in comparing the ILLIAC IV languages was ZAP, a one-dimensional Lagrangian hydrodynamics code. The code was used to simulate shock-wave phenomena in a spherical segment of air. The equation of state used was a simple gamma-law model.

Two versions of the program were coded in each of the languages. The first, called ZAP62, used a grid of 62 elements, not counting the boundary-condition elements at each end of the grid. With this grid size, each of the ILLIAC IV's 64 processing elements (PEs) contained the variables for a single grid element. Computations were carried out only for an "active" set of elements which numbered, at most, 62.

A second version, called ZAPN, used a grid of up to 4096 elements. This version allows computing problems of more physical significance. Thus, ZAPN is the more useful program. It also allows a determination of how well each of the languages handles vectors with lengths significantly longer than the 64 PE width of the ILLIAC IV's architecture.

Comparison of the Codes

Figure 1A shows a portion of the FORTRAN version of ZAP. These two loops update the velocity, U, and position, X, of each element in the current active set (which ranges from element 2 to element JSTAR). Two separate loops are required since updating U for a given element to the next time step requires the value of X at the current time step for the preceeding element. It should be noted that the computational sections of both ZAP62 and ZAPN are identical in FORTRAN since only a change in the dimensions of the variables is required.
Figure 1B shows the equivalent computational portion of the CFD versions of ZAP62 and ZAPN. For ZAP62, we note that the DO loops have been replaced by a MODE setting operation that turns on only those PEs containing active elements. There is no need to worry about explicitly separating the calculations for U and X since all the U's are updated simultaneously.

Major changes in coding are required between ZAP62 and ZAPN. The vector variables involved are now explicitly two dimensional, the second dimension being the row index. Explicit DO loops are now required to handle the calculation over multiple rows, and two separate DO loops are required since the vector assignment statements update only one row of values at a time.

The series of statements prior to the DO loops is needed to determine the number of rows of elements in the active set. This number, JJ, is the upper limit of the DO loops. This series of statements also determines JL, the number of active elements in the JJth row.

Two more changes are inside the DO loops. The first is the series of conditional MODE setting operations. This is required since the first and JJth rows do not have all PE's turned on as opposed to the intermediate rows which do. The second change is the presence of offset vector indices for those variables that are routed one PE from the right or left. These offset vector indices are needed to handle a routing problem with multirow vectors in CFD. This problem arises since the first index of a vector-alligned variable utilizes the wraparound feature of the ILLIAC IV's routing architecture. As an example, consider routing from one PE to the left. Using the standard CFD first index of *-1 causes all the PEs except the first to access the proper value. The first PE accesses the value in the same row in the last PE, whereas the problem requires it to access the value in the preceeding row in the last PE. The vector index OFFL has a value of -1 in the first PE and 0 in all other PEs. This difference in value gives the proper accessing. Similarly, the vector index OFFR handles accessing from one PE to the right.

It should be noted that this approach is problem dependent. For ZAPN, the first PE of the first row and the last PE of the last row are always off, since they contain only the boundary conditions. Thus, no problems are encountered when accessing from the left in the first PE in the first row or accessing from the right in the last PE of the last row. For programs where these PEs may be on, a different approach must be used.

Figure 1C shows this same portion of code for the GLYPNIR versions of ZAP62 and ZAPN. For ZAP62, the GLYPNIR equivalent of a DO loop is present. Only one loop is required since all the vector variables are updated at once. Note that the vector variables do not have subscripts since GLYPNIR does not require it. Also note the explicit references to the inline routine functions, RTL, and RTR.

Again, programming changes are needed when going from ZAP62 to ZAPN. The vector variables now have an index (K) which is equivalent to the second index (II) in the CFD version. Again, two separate loops are required. The loop control for these loops appears similar to that used in ZAP62. Actually, it is more complicated. The GLYPNIR DEFINE facility was used to produce a macro, called FORALL(J), which determines the range of the loop and handles the MODE setting operations which are explicit in the CFD version. Since there are several loops in ZAPN, this DEFINE facility allows the programmer to establish the loop control and MODE setting operations once and then to use them wherever needed by calling the defined macro.

The routine problem is handled by the use of the loop integer scalar index plus an integer vector offset index. This is similar to the approach used in the CFD version.

Figure 1D shows this same section of code in the IVTRAN versions of ZAP62 and ZAPN. These versions closely resemble the original FORTRAN code. The DO loops have now become DO FOR ALL loops, with ZAP62 again only requiring one loop while ZAPN requires two. The only other difference between the ZAP62 and ZAPN versions is the range of the control index. For ZAP62, the index is restricted to the range 2 to 62, while for ZAPN it spans the range from 2 to 4095. No special coding is required to handle either the multi-rowed variables or the routine problem. The IVTRAN compiler itself generates the necessary code to handle those factors. Thus, IVTRAN is clearly the easiest language in which to convert this section of code from ZAP62 to ZAPN.

Figure 2 shows part of another loop in ZAPN. This portion of code determines the time for a wave to cross a fraction (WW) of each element in the grid. The time step for the next calculation cycle is then set to the minimum of the times for the elements. For the FORTRAN version, a running minimum is found using the AMIN1 function.

For the CFD version, significant coding changes are again required. The calculation of the loop limit and the MODE set are again required, though they are not shown since they are similar to those in Figure 1B. The calculation of the minimum time is accomplished in two stages. The first stage calculates the times for each active element in the current row and then finds the minimum of all elements in the row, whether active or not. The times are initialized to a large value to keep the inactive elements from influencing the result. The second stage keeps a running minimum over the rows. Note that this requires an entire vector to store this minimum, since CFD requires that the floating point conditional expression contain a vector quantity and TEMP(2) is treated like a scalar.

The GLYPNIR version uses a similar approach to CFD. A scalar is set to the minimum of a row of times and a running scalar minimum is kept over the rows. Note that GLYPNIR uses a scalar, whereas CFD required an entire vector to store the minimum.

Again, the IVTRAN version looks similar to the FORTRAN version. The only difference is that the supplied row minimum function RMIN is called to set up the argument for AMIN1. IVTRAN is again the easiest language to convert this section of code.

Table 1 gives the number of source statements in the main calculational loop of ZAPN and ZAP62 for each of the languages. For IVTRAN, CFD, and FORTRAN, the number of statements is the number of lines not including continuations. For GLYPNIR, the number of statements is the number or statement terminators. For CFD and GLYPNIR, the source statement counts included setting up the vector offset indices. For GLYPNIR, the statements in the macro used for loop control are also included. For ZAP62, CFD requires more source statements than GLYPNIR or IVTRAN, which require similar numbers. For ZAPN, IVTRAN requires fewer source statements than GLYPNIR, which in turn requires fewer than CFD.

Table 1
Source Statements for ZAP

Language	# of Source Statements (ZAPN)	# of Source Statements (ZAP62)
FORTRAN	32	32
CFD	68	39
GLYPNIR	44	30
IVTRAN	32	32

Two comments should be made concerning this comparison of the various versions of ZAPN. First, the IVTRAN version was produced by putting the FORTRAN version through the paralyzer and then making minor changes to the output. The original FORTRAN codes were written with that in mind and thus a comparison of the FORTRAN and IVTRAN versions tends to underestimate the effort required in the conversion.

Second, since the original program was coded in FORTRAN, the comparison has a slight intrinsic bias against GLYPNIR. IVTRAN seems easier to use since it has the syntactic structure of FORTRAN. Had the original program been written in ALGOL, GLYPNIR would have appeared a little easier to use than it did in this comparison.

Nonetheless, IVTRAN did require the least user effort to reprogram ZAPN, primarily due to the syntactic similarity to FORTRAN. GLYPNIR was a reasonably close second because of the clarity of its syntax. The effort required with CFD was significantly greater than for the other languages since its syntactic structure conforms more directly to the architecture of the machine.

Execution of Standard Problem

Due to current hardware problems with the ILLIAC IV, the standard problem has not been successfully executed for any of the versions of ZAPN. There are results for a problem run using ZAP62. The CFD, GLYPNIR, and IVTRAN versions gave the same results as the original FORTRAN version for the problem used.

Table 2A gives the number of equivalent ASK statements for the main calculational loop of ZAP62 for each of the languages, and the time required to execute the standard problem on the ILLIAC IV.

Table 2A.
Results for ZAP62

Language	# of ASK Statements	Execution Time (Sec)
CFD	309	1.22
GLYPNIR	560	1.73
IVTRAN	330	1.36

While ZAPN has been coded, and partially debugged in all three languages, the standard problem has not been run on the ILLIAC IV with all versions. Therefore, the execution times have been estimated from the equivalent ASK generated from each version. The timing estimates are for one pass through the main body of the program, and the standard problem used was for vectors of 512 elements (i.e., 8 rows of memory) with 220 active elements (i.e., 4 rows). Since the ILLIAC IV has, in the past, been run with overlap disabled, this mode was assumed for the estimate.

Table 2B.
Results for ZAPN

Language	# of ASK Statements	Normalized Est. Execution Time
CFD	440	1.00
GLYPNIR	887	1.70
IVTRAN	425	1.19

The main reason for the relatively long estimated time for the GLYPNIR version is that GLYPNIR produced significantly more ASK statements for ZAPN than did the other languages. While some of the extra statements are due to such factors as

GLYPNIR generating the MIN function in-line (as opposed to CFD, for which ROWMIN is an external), much of the difference is due to the fact that GLYPNIR, in general, produces extra statements.

For instance, GLYPNIR produces 2 more ASK statements for each access of a multi-row vector than does CFD. It should be noted that the GLYPNIR compile was done with the BOUND option off, so as not to generate the extra code required to verify that vector row accesses were within bounds.

As was mentioned previously, neither CFD nor GLYPNIR automatically handles multi-row vectors. The programmer must explicitly include code to handle this. For ZAPN, both the CFD and GLYPNIR programmers chose to determine the number of rows in the active set and then to loop on that number. While IVTRAN automatically handles the multi-row vectors, it does so by setting the loop limit to the full number of rows contained in the vector and generating code to skip the body of the loop for those rows that do not contain active elements. Thus, while the timing estimates for each of the versions is dependent on the number of elements assumed to be active, the IVTRAN timing estimate also depends on the number of elements in the vectors. Since the normal problem run with ZAPN has half of the elements active, on the average, the length of the vectors assumed for the timing estimate was twice the active set. Note, however, that if the vectors had been assumed to be of length 4096 (i.e., 64 rows of memory), the IVTRAN estimated time would have been 3.63, while the estimated times for the CFD and GLYPNIR versions would have been unchanged.

PUKA

The second program used for comparing the ILLIAC IV languages was PUKA, a two-dimensional Eulerian hydrodynamics code. Again, a simple gamma-law gas model was used for the equation of state.

The original FORTRAN version of PUKA was written to perform calculations on a series of buffers. With the appropriate set of buffer-handling routines, the code will work on variables that are either totally core-contained or partially disk resident. The core-contained version of PUKA was used for the language comparison.

Comparison of the Codes

PUKA contains two main calculational subroutines-PH1, which update the velocities, and PH2, which calculates the mass transport between cells. These two subroutines will be discussed separately.

Comparison for PH1 Figure 3A shows a portion of the FORTRAN version of PH1. This portion determines the values of the velocity and pressure below the current row. How these values are determined depends on two conditions. If the current row is not the bottom of the grid, i.e., J is not 2, then the velocities and pressures below are the averages of the values in the current row and the row below it (the DO 60 loop). If the current row is the bottom, the values depend on the bottom boundary condition. If the bottom is not reflective, i.e., BTREF=.FALSE., then the velocity below is velocity in the current row, while the pressure is the average. If the bottom in reflective, then the velocity below is 0, and the pressure below is the pressure in the current row.

Figure 3B shows this same portion of code in the CFD version. This version looks similar to the original FORTRAN. The main difference is that an explicit mode set is used instead of the DO loop structure of FORTRAN.

Figure 3D shows this portion of code in the GLYPNIR version. While this portion appears very different from the FORTRAN version, it does offer the advantage of lucid conditional control structure that is found in GLYPNIR. The use of the IF..THEN.. ELSE structure makes explicitly clear how the variables depend on the two conditions. This clarity adds substantially to the ease of programming.

Figure 3D shows the IVTRAN version of this portion of PH1. Except for the conversion of the DO loops to DO FOR ALL loops, this version is identical to the original FORTRAN.

Table 3 gives the number of source statements in PH1 for each of the versions. For this routine, GLYPNIR required significantly fewer statements than either of the other versions because of its lucid control structure. CFD required slightly fewer statements than the original FORTRAN since explicit DO loops were not needed. The IVTRAN version required the same number of statements as the FORTRAN version.

Table 3.
Source Statements for PH1

Language	# of Source Statements
FORTRAN	69
CFD	56
GLYPNIR	33
IVTRAN	69

Even though CFD required fewer statements than IVTRAN, the programming efforts involved were similar since CFD required more changes from the original FORTRAN. While the GLYPNIR version is quite different from the FORTRAN version, it required so many fewer statements that it was the easiest language for the conversion.

Comparison for PH2 Figure 4A shows a portion of the FORTRAN version of PH2. This portion of code handles some special calculations for the top row of the grid, which has J equal to JMAXA. The DO 275 loop is primarily for determining how much mass is flowing through the top of the two (MSTTOP). For the top of the grid, this value cannot be negative since that would imply that the mass is flowing into the grid from outside. If for the top of the grid, MSTTOP is greater than a specified fraction TOZONE of the mass in the top row, a logical flag RZNTOP is set to .TRUE. to indicate that the problem should be rezoned before continuing. Then, the calculation determines how much energy is lost from the grid by the mass flowing out the top. The DO 400 loop subtracts that energy from the total so that later energy checks will not fail.

Figure 4B shows the CFD version of this same portion of code. Other than the removal of the specific DO loops, there are two major changes required. Since an entire row of values of MSTTOP is simultaneously checked to see if any values are large enough to require setting RZNTOP, the logical operator .ANY. must be used. Also note that RZNTOP is not set to .TRUE., which is a scalar value, but to ON, which is a logical vector constant. The other change occurs in subtracting the sum of the values of DTH from ETH. In CFD, this calculation requires the use of a temporary row vector since a floating point arithmetic assignment statement cannot involve only scalars.

Figure 4C shows the GLYPNIR version of this portion code. The main conditional control structure utilizes the IF..THEN..ELSE construct. The calculational statements are very similar to the original FORTRAN. The ROWSUM function is utilized, though the entire calculation requires only one statement unlike the CFD version.

Figure 4D shows this portion of code in the IVTRAN version. There are a couple of changes from the original FORTRAN. The biggest change required is for the conditional statement that sets RZNTOP. In IVTRAN, RZNTOP is set by ORing its previous value with the logical function ANY which determines if any of the values of MSTTOP are larger than the conditional value.

The DO 400 FOR ALL loop shows an idiosyncracy of the paralyzer. The original FORTRAN loop ran from 2 to IACT. The paralyzer changes this so that the loop runs from 1 to IACT-1 and compensates by replacing all references to I by I+1. For this comparison, the other loops have been manually changed to reflect the FORTRAN sequence. The IVTRAN compiler will accept both forms. Thus, the effort required to convert the programs to IVTRAN is larger than might appear from the examples shown in this report.

Table 4 gives the number of source statements in each of the versions of PH2. Again, GLYPNIR required far fewer statements than the other languages. IVTRAN and CFD required the same number of statements, though the changes required for IVTRAN were easier to implement than those required for CFD.

Table 4.
Source Statements for PH2

Language	# of Source Statements
FORTRAN	110
CFD	107
GLYPNIR	75
IVTRAN	107

Execution of Standard Problem

The standard problem for PUKA has not yet been run on the ILLIAC IV in any version due to current hardware problems. Since the various versions of PUKA are still in the development stage, it was felt unwise to estimate timings for the code produced so far.

CONCLUSION

There is currently little data for judging the efficiency of machine utilization. There are ILLIAC IV execution times only for ZAP62 and the estimated times for ZAPN. On the basis of these preliminary results, CFD appears to give the shortest execution times and GLYPNIR the longest, though the ratio is less than a factor of two. IVTRAN appears to give execution times intermediate between those of CFD and GLYPNIR, though as mentioned in the results for ZAPN, it is possible for IVTRAN to give longer execution times than GLYPNIR.

There is preliminary data for determining the user effort required for each of the languages. The comparison of the versions of ZAPN shows that IVTRAN is the easiest language for handling multi-row vectors. For PUKA, GLYPNIR required the least effort due primarily to the ease of handling conditional control statements in that language. For both codes, CFD was the most difficult language primarily because it is not as high a level as the other two.
The I/O handling capabilities of the languages have not been discussed. Only CFD presents any major difficulties in this respect since it supports only unformatted, binary I/O. Thus, post-processing of the output is required to put it into human-readable form.

One other code which has been briefly investigated deserves mention. This code is a global weather model that explicitly utilizes the wraparound feature of the ILLIAC IV's routing hardware. This wraparound is simple to utilize in either GLYPNIR or CFD, while IVTRAN requires a special function to handle it. This is as might be expected, since IVTRAN handles multi-row vectors so easily.

Several factors that influence the required programming effort were not used in comparing the languages but do deserve brief mention. Accessability of the compiler is one such factor. Currently, only GLYPNIR is available at the ILLIAC IV site. CFD is only available on the 360 at NASA Ames, and IVTRAN is only available on the TENEX system at ISI. Thus, the CFD or IVTRAN user must spend some effort in transferring files between the host site for the compiler and the ILLIAC IV site, while the GLYPNIR user does not face that problem. Another factor is the level of support for the language. Currently, GLYPNIR is without funded support, and the future support levels for both CFD and IVTRAN are in some doubt.

It should also be mentioned that each of the languages has some facilities to allow program simulations on a serial computer. For GLYPNIR, there is SSK, which simulates execution of ILLIAC IV code on the B6700. For CFD programs, there is a translator, called CFDX, which produces equivalent serial FORTRAN. For IVTRAN, there is the paralyzer that converts serial FORTRAN into IVTRAN. The use of the paralyzer may require rewriting the FORTRAN code so that efficient IVTRAN is produced, though these rewritten versions can be checked using a serial machine.

None of the three languages compared herein are best in all respects or for all programs. Each language has its own strengths and weaknesses, some of which are given in Table 5. In general, GLYPNIR's main strengths are its lucid control-structure and the in-line macro facility. GLYPNIR's main weakness is that it does not as efficiently utilize the machine as do CFD and IVTRAN. CFD's main strength is that it's syntax reflects the architecture of the machine thus allowing very efficient code to be generated. However, the syntax is fairly restrictive and, thus, more effort is required to program in CFD. IVTRAN's main advantage is that the syntax is close to standard FORTRAN; while the disadvantage is that the syntax does not reflect the architecture of the machine. Also, while IVTRAN automatically handles multi-row vectors, an advantage, the code produced is less than optimal for some problems.

I would like to acknowledge the contributions of those persons who assisted in this project: Mr. Terry Layman of R & D Associates, Dr. Charles Muntz, of Massachusetts Computer Association, and Mr. Ken Stevens of the NASA Ames Research center.

Table 5.
ILLIAC IV Languages

CFD:

Advantages
- Syntax relfects architecture of the ILLIAC IV
- Produces reasonably good machine code
- I/O reflects structure of disk transfers
- Allows inserting ASK statements in-line

Disadvantages
- BOOLEAN operations are tricky
- All variables must be assigned storage at beginning of each routine
- No formatted I/O
- No multi-statement optimization

GLYPNIR:

Advantages
- Lucid control structure
- In-line macro facility
- Allows insertion of ASK statements in-line
- Pointer constructs for handling complex data structure
- Formatted and binary I/O

Disadvantages
- Relatively poor machine utilization
- Allows only two-dimensional variables
- No multi-statement optimization

IVTRAN:

Advantages
- Syntax similar to standard FORTRAN, with extension
- Multi-statement optimization
- Formatted and binary I/O
- Multi-dimensional arrays

Disadvantages
- Syntax does not reflect machine architecture
- Routing is serial, not wraparound

Illustrations

```
     ZAP:
     DO 341 J=2,JSTAR
     U(J)=U(J)+DELT*((P(J)+Q(J)-
     1Q(J+1)-P(J+1))/(RHO(J)*(X(J)-
     2     X(J-1))+RHO(J+1)*(X(J+1)
     3-X(J))))
      IF(ABS(U(J)).LT.UMIN) U(J)=0.0
 341 CONTINUE
     DO 351 J=2,JSTAR
     X(J)=X(J)+DTC*U(J)
 351 CONTINUE
```

Figure 1A. FORTRAN

```
     ZAP62:
     MODE=OFF.TURN ON .2.TO.JSTAR
     U(*)=U(*)+DELT*((P(*)+Q(*)-Q(*+1)-
      -P(*+1))/(RHO(*)*(X(*)-X(*-1))+ -
      RHO(*+1)*(X(*+1)-X(*))))
    * IF((ABS(U(*)).LT.UMIN))U(*)=0.0
      X(*)=X(*)+DTC*U(*)
      ZAPN:
      JJ=1
      JL=JSTAR
 231*IF(JL.LE.64) GO TO 232
      JL=JL-64
      JJ=JJ+1
     *GO TO 231
 232 JN=64-JL
     *DO 341 II=1,JJ
      MODE=ON
     *IF(II.EQ.1) MODE=MODE.TURN OFF..FIRST.1
     *IF(II.EQ.JJ) MODE=MODE.TURN OFF..LAST.JN
      U(*,II)=U(*,II)+DELT*((P(*,II)+Q(*,II)-
       -Q(*+1,OFFR(*)+II)-     -
        P(*+1,OFFR(*)+II))/RHO(*,II)*-
         (X(*,II)-X(*-1,OFFL(*)+II))+    -
         RHO(*+1,OFFR(*)+II)*(X(*+1,     -
         OFFR (*)+II)-X(*,II))))
     *IF((ABS(U*,II)).LT.UMIN)) U(*,II)=0.0
  341 CONTINUE
     *DO 351 II=1,JJ
      MODE=ON
     *IF(II,EQ.1) MODE=MODE.TURN OFF.FIRST.1
     *IF(II.EQ.JJ) MODE=MODE.TURN OFF..LAST.JN
      X(*,II)=X(*,II)+DTC*U(*,II)
  351*CONTINUE
```

Figure 1B. CFD

```
ZAP62:
    IF(PEN GTR 0 AND PEN LEQ JSTAR) THEN
    BEGIN
      U:=U+(DELT*(P+Q-RTL(1,MODE,Q)-RTL(1,
      MODE,P)))/(RHO*(X-RTR(1,MODE,X))+RTL
      (1,MODE,RHO)*(RTL(1,MODE,X)-X));
    IF ABS(U) LSS UMIN THEN U:=0.0;
    X:=X+DTC*U;
    END;

ZAPN:
    FORALL (JSTAR) DO
    BEGIN
      U[K]:=U[K]+(DELT*(P[K]+Q[K]-RTL(1,,
      Q[K+OFFL])-(RTL(1,,P[K+OFFL])))/
      (RHO[K]*(X[K]-RTR(1,,X[K+OFFR]))+
      RTL(1,,RHO[K+OFFL])*(RTL(1,,X[K+OFFL])
      -X[K]));
    IF ABS(U[K]) LSS UMIN THEN U[K]:=0.0;
    END;
    FORALL (JSTAR) DO
    BEGIN
      X[K]:=X[K]+DTC*U[K];
    END;
```

Figure 1C. GLYPNIR

```
ZAP62:
    DO 351 FOR ALL (J)/[(J)/[2,3...63]:
  1   J.LE.JSTAR]
    U(J)=U(J)+DELT*((P(J)+Q(J)-Q(J+1)-P(J+1))
  1   /(RHO(J)*(X(J)-
  2   X(J-1))+RHO(J+1)*(X(J+1)-X(J))))
    IF(ABS(U(J)).LT.UMIN) U(J)=0.0
    X(J)=X(J)+DTC*U(J)
351 CONTINUE

ZAPN:
    DO 341 FOR ALL (J)/[(J)/[2,3...4095]:
  1  J.LE.JSTAR]
    U(J)=U(J)+DELT*((P(J)+Q(J)-Q(J+1)-P(J+1))/
  1  (RHO(J)*(X(J)     -
  2      X(J-1))+RHO(J+1)*(X(J+1)-X(J)))
    IF(ABS(U(J)).LT.UMIN) U(J)=0.0
341 CONTINUE
    DO 351 FOR ALL (J)/[(J)/[2,3...4095]:
  1   J.LE.JSTAR]
    X(J)=X(J)+DTC*U(J)
351 CONTINUE
```

Figure 1D. IVTRAN

```
FORTRAN:
    DO 501 J=2,JSTR1
    .
    .
    .
    DTZJM=AMIN1(DTZJM,WW*(X(J)-X(J-1))/
   1  (CRNT(J)+CS(J)+0.01))
501 CONTINUE

CFD:
   *DO 501 II=1,JJ
    .
    .
    DTZJ(*)=1.E30
    .
    .
    .
    DTZJ(*)=WW* (X(*,II)-X(*-1,OFFL(*)+II))/-
     (CRNT(*)+CS(*,II)+0.01)
    TEMP(*)=ROWMIN(DTZJ(*))
   *IF(.ANY.(DTZJM(*).GT.TEMP(2))) DTZJM(*)  -
      =TEMP(2)
501*CONTINUE

GLYPNIR:
    FORALL (JSTR1) DO
    BEGIN
    .
    .
    .
      CTEMP:-MIN(WW*(X[K]-RTR(L,,X[K+OFFR]))/
        (CRNT+CS[K]+0.01));
      IF CTEMP LSS DTZJM THEN DTZJM:=CTEMP;
    END;

IVTRAN:
    DO 501 FOR ALL (J)/[(J)/[2,3...4095]:
  1   J.LE.JSTR1]
    .
    .
    DTZJM-AMIN1(DTZJM,RMIN(WW*(X(J)-X(J-1))/
  1   (CRNT(J)+CS(J)+0.01)))
501 CONTINUE
```

Figure 2. ZAPN

```
    IF(J.NE.2) GO TO 50
    IF(.NOT.BTREF) GO TO 30
    DO 20 I=1,IACTA
    VB(I)=0.0
    PB(I)=P(I,I3)
 20 CONTINUE
    GO TO 75
 30 DO 40 I-1, IACTA
    VB(I)+V(I,I3)
    PB(I)=0.5*(P(I,I3)+O(*,I2))
 40 CONTINUE
    GO TO 75
 50 DO 60 I=1, IACTA
    VB(I)=0.5*(V(I,I3)+V(I,I2))
    PB(I)=0.5*(P(I,I3)+P(I,I2))
 60 CONTINUE
 75 CONTINUE
```

Figure 3A. FORTRAN

```
    MODE=OFF.TURN ON..FIRST.IACTA
   *IF(J.NE.2) GO TO 50
   *IF(.NOT ANY.(BTREF)) GO TO 30
    VB(*)=0.
    PB(*)=P(*,I3)
   *GO TO 75
 30 VB(*)=V(*,I3)
    PB(*)=0.5*(P(*,I3)+P(*,I2))
   *GO TO 75
 50 VB(*)=0.5*(V(*,I3)+V(*,I2))
    PB(*)=0.5*(P(*,I3)+P(*,I2))
 75*CONTINUE
```

Figure 3B. CFD

```
FORALL (1,IACTA) DO
  IF J=2 THEN
    IF BTREF THEN
    BEGIN
      VB:0.0; PB:=P[I3];
    END;
    ELSE
    BEGIN
      VB:=V[I3]; PB:=0.5*(P[I3]+P[I2]);
    END;
    ELSE BEGIN
      VB:=0.5*(V[I3]+V[I2]);
      PB:=0.5*(P[I3]+P[I2]);
    END;
```

Figure 3C. GLYPNIR

```
   IF(J.NE.2) GO TO 50
   IF(.NOT.BTREF) GO TO 30
   DO 20 FOR ALL (I)/[(I)/[1,2...64]:I.LE.IACTA]
   VB(I)=0.0
   PB(I)=P(I,I3)
20 CONTINUE
   GO TO 75
30 DO 40 FOR ALL (I)/[(I)/[1,2...64]:I.LE.IACTA]
   VB(I)=V(I,I3)
   PB(I)=0.5*(P(I,I3)+P(I,I2))
40 CONTINUE
   GO TO 75
50 DO 60 FOR ALL (I)/[(I)/[1,2...64]:I.LE.IACTA]
   VB(I)=0.5*(V(I,I3)+V(I,I2))
   PB(I)=0.5*(P(I,I3)+P(I,I2))
60 CONTINUE
75 CONTINUE
```

Figure 3D. IVTRAN

```
    IF(J.EQ.JMAXA) GO TO 250
    .
    .
    .
    GO TO 300
250 DO 275 I=2,IACT
    VTPIN(I)=V(I,I8)
    MSTTOP(I)=VTPIN(I)*DT*DMSDDY(I)
    IF(MSTTOP(I).LT.0.0) MSTTOP(I)=0.0
    IF(MSTTOP(I).GT.TOZONE*AMX(I,I2)) RZNTOP=.TRUE.
    DTH(I)=DTH(I)+MSTTOP(I)*DTPSPE(I)
275 CONTINUE
300 CONTINUE
    .
    .
    .
    DO 400 I=2,IACT
    ETH=ETH-DTH(I)
400 CONTINUE
```

Figure 4A. FORTRAN

```
   *IF(J.EQ.JMAXA) GO TO 250
    .
    .
    .
   *GO TO 300
250 VTPIN(*)=V(*,I8)
    MSSTOP(*)=VTPIN(*)*DT*DMSDDY(*)
   *IF((MSTTOP(*).LT.0.0)) MSTTOP(*)=0.0
   *IF(.ANY.((MSTTOP(*).GT.ROZONE*AMX(*,I2))))  -
      RZNTOP=ON
300*CONTINUE
    .
    .
    .
    TEMP(*)=ROWSUM(DTH(*))
    TEMP(*)=ETH-TEMP(*)
    ETH=TEMP(2)
```

Figure 4B. CFD

```
FORALL (2,IACT) DO
IF (J NEQ JMAXA) THEN
.
.
.
ELSE
BEGIN
  VTPIN:=V[I8];
 MSTTOP:=VTPIN*DT*DMSDDY;
 IF   (MSTTOP LSS 0.0) THEN MSTTOP:=0.0;
 IF   (MSTTOP GTR TOZONE*AMX[I2]) THEN
   RZNTOP:=TRUE;
 END;
 .
 .
 .
 FORALL (2,IACT) DO
   ETH:=ETH-ROWSUM(DTH);
 END;
```

Figure 4C. GLYPNIR

```
    IF(J.EQ.JMAXA) GO TO 250
    .
    .
    .
    GO TO 300
250 DO 275 FOR ALL (I)/[(I)/[2,3...63]:I.LE.IACT]
    VTPIN(I)=V(I,I8)
    MSTTOP(I)=VTPIN(I)*DT*DMSDDY(I)
    IF(MSTTOP(I).LT.0.0)MSTTOP(I)=0.0
    RZNTOP=RZNTOP.OR.ANY(MSTTOP(I).GT.TOZONE*
   1 AMX(I,I2))
    DTH(I)=DTH(I)+MSTTOP(I)*DTPSPE(I)
275 CONTINUE
300 CONTINUE
    .
    .
    .
    DO 400 FOR ALL (I)/[(I)/[1,2...63]:I.LE.IACT-1]
    ETH=ETH-RSUM(DTH(I+1))
400 CONTINUE
```

Figure 4D. IVTRAN

References

[1] Lawrie, D. H., T. Layman, D. Baer, and J. M. Randal, "GLYPNIR - A Programming Language for ILLIAC IV," Comm. ACM 18, 3 (March 1975), 157-164.

[2] CFD - A FORTRAN-based Language for ILLIAC IV, Computational Fluid Dynamics Branch, NASA/Ames Research Center.

[3] The IVTRAN Manual, Institute for Advanced Computation, Doc. DS-A-8000-010-B, February 18, 1975.

SYSTEMS DESIGN AND DOCUMENTATION USING PATH DESCRIPTIONS

Alan C. Shaw[a]
Department of Computer Science
University of Washington
Seattle, Washington 98195

Summary

Sequential and concurrent systems are often designed and understood by informally considering the execution time paths through their software and hardware components. While there has been much theoretical research in parallel processing based on control flow specification and analysis, for example using different graph models of computation [1], we are not aware of many practical schemes that are of direct assistance to the designer. This paper presents some preliminary results on the definition and application of a practical notation for systems description.

Related research includes that of Riddle, who defined operators similar to ours in his Message Transfer Expressions for specifying permissible sequences of messages and synchronization constraints among processes [2] - [3], and the Path Expression work of Campbell, Habermann, and Lauer [4] - [6]. There is also the report on the Pascal <P> compiler [7]; the code generation software is described in a top-down fashion by a sequential path notation, somewhat clearer than conventional flowcharts, which incorporates syntax diagrams and productions.

In our scheme, systems are described by sets of rules or productions where each rule contains a path description denoting the possible execution-time control paths through some components. Let $L(S)$ be the set of paths represented by the path description S. For sequential activities, the language of regular expressions is used for path descriptions; $L(S)$, S a regular expression, is the regular set corresponding to S. Extensions to this language are provided to handle concurrency. (Path descriptions can also be represented graphically, using a straightforward extension of syntax diagrams.)

The concurrent operator $\odot$ specifies the meaning of concurrent activities in terms of sequential paths through a system; the underlying idea is that the effect of the concurrent (parallel) execution of two components or routines is equivalent to the execution of one sequential path through both components, but the particular one chosen is in general unknown.
Define $L(e_1 \odot e_2) = \{e_1e_2\ ,\ e_2e_1\}$ for e_1, e_2 elementary (indivisible), and
$L(S_1 \odot S_2) = \{\text{all paths thru both } S_1 \text{ and } S_2\}$
$= \{x_1y_1 \ldots x_ky_k : x_1 \ldots x_k \in L(S_1)\ ,$
$y_1 \ldots y_k \in L(S_2)$, and either or both x_1 and y_k could be empty$\}$.

A variable degree of concurrency is denoted by the concurrent * operator, $\circledast$; i.e.

$$L(S^{\circledast}) = \bigcup_{n=0}^{\infty} L(S^{\odot n}) \quad (\text{e.g. } S^{\odot 3} = S \odot S \odot S).$$

This operator is useful when there may be execution-time control over the degree of logical or physical concurrency.
An expression is designated as indivisible (a critical section) by surrounding it with square brackets; the expression is then treated as elementary in the context of the $\odot$ operator. Different critical section locks are distinguished by labelling the bracketted descriptions; bracketted expressions are then considered elementary only when they interact concurrently with those having identical locks.
Thus $L([S_1]_k \odot [S_2]_k) = L(S_1.S_2) \cup L(S_2.S_1)$ [b]
and $L([S_1]_j \odot [S_2]_k) = L(S_1 \odot S_2)$ (provided that no j locks appear in S_2 and no k locks are in S_1).
Note that the definition of $\odot$ must now be amplified to ensure that all paths obey the locking rules.
Cyclic processes and programs are described with the ω operator; thus $S^{\omega} = S.S.S \ldots$.

For a first example, we consider the path of a batch user job through a simple spooled multiprogramming system. This path can be generated by the "syntax":

Batch_User_Paths → Input_Spool . Run . Output_Spool

Input_Spool → (Read_Input . Process_Input . Write_on_Disk)$^+$ [c]

Run → Load . Execute . Terminate

Execute → (Compute . (Input_Output ∪ Resource_Request))$^+$

Output_Spool → (Read_From_Disk . Process_Output . Write_Output)$^+$

We next describe the multiprogramming situation where a variable number of such jobs may be in execution concurrently, where each job stream is cyclic, where only one Input_Spool and Output_Spool operation can occur at a time but Input_Spool and Output_Spool can occur concurrently, where only one job can be loaded at a time, and so on:

System → Sequential_Jobs$^{\circledast}$

Sequential_Jobs → Batch_Job$^{\omega}$

[a] The author is grateful to R. Ladner and W. Riddle for several useful discussions. This work was supported in part by NSF Grant GJ-36273.

[b] The "." (dot) denotes concatenation of regular expressions.

[c] The "+" specifies one or more repetitions.

$$\text{Batch\_Job} \rightarrow [\text{Input\_Spool}]_{is} \ . \ \text{Run} \ . \ [\text{Output\_Spool}]_{os}$$

$$\text{Run} \rightarrow [\text{Load}]_{\ell} \ . \ \text{Execute} \ . \ [\text{Terminate}]_{t}$$

$$\text{Execute} \rightarrow (\text{Compute} \ . \ ([\text{Input\_Output}]_{io} \cup [\text{Resource\_Request}]_{r}))^{+}$$

Our second example is in time-sharing systems. Suppose a system potentially swaps workspaces of interactive terminal users out of main storage at the end of a time slice or on an input-output (IO) request; assume that swapping and terminal IO can occur concurrently. Then, ignoring logon and logoff, the paths of a terminal user can be described:

$$\text{Terminal\_User\_Paths} \rightarrow (\text{Compute} \ . \ ((\text{Time\_Up} \ . \ (\text{Swap} \cup \varepsilon)) \cup (\text{Terminal\_IO} \odot (\text{Time\_Up}^{*} \ . \ (\text{Swap} \cup \varepsilon)))))^{+}$$

$$\text{Swap} \rightarrow [\text{Swap\_Workspace\_Out}]_{sw} \ . \ [\text{Swap\_Workspace\_In}]_{sw}$$

Because we are interested in a scheme which is practically useful and not too cumbersome, we have not incorporated data into the notation nor have we provided for more complex interactions by, for example, including the equivalent of a Dijkstra P and V at this time. Consequently, many synchronization constraints, such as those that appear in general Reader-Writer problems, appear difficult to express. The notation is being used to describe the software in several systems currently under development. As we gain more experience, we expect our top-down, syntax-oriented, path scheme to evolve to correct any serious deficiencies.

References

[1] J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing", ACM Computing Surveys 5, 1 (March, 1973), pp. 31-80.

[2] W. E. Riddle, The Modeling and Analysis of Supervisory Systems, Ph.D. Thesis, Computer Science Dept., Stanford University, (March, 1972).

[3] W. E. Riddle, Message Transfer Expressions and Their Use in Specifying Synchronization Constraints, RSSM/1, Dept. of Computer and Communication Sciences, University of Michigan, (Sept., 1974), 14 pp.

[4] A. N. Habermann, Operations on Shared Data Controlled by Function Modules in Type Definitions, Computer Science Dept., Carnegie-Mellon University, (Sept., 1973), 10 pp.

[5] R. H. Campbell and A. N. Habermann, The Specification of Process Synchronization by Path Expressions, TR 55, Computing Laboratory, University of Newcastle-Upon-Tyne, (Jan., 1974).

[6] P. E. Lauer and R. H. Campbell, "A Description of Path Expressions by Petri Nets", Proc. 2nd ACM Symp. on Principles of Progr. Lang., Palo Alto, Calif., (Jan., 1975), pp. 95-105.

[7] K. V. Nori, U. Ammann, K. Jensen, and H. H. Nägeli, The Pascal <P> Compiler: Implementation Notes, Nr. 10, Berichte des Instituts für Informatik, Eidgenössische Technische Hochschule, Zürich, (Dec., 1974), 57 pp.

A NEW SCHEME FOR ANALYZING PARALLEL PROCESSING SYSTEMS

Ines MARGARIA (-), Angelo Raffaele MEO (°), Maddalena ZACCHI (-)

(-) Istituto di Scienza dell'Informazione,
Università di Torino, Italia.

(°) Istituto di Elettrotecnica Generale,
Politecnico di Torino, Italia

Abstract. - A new model specifically studied for analyzing multimicroprocessor systems and automatically detecting parallelism is introduced and discussed from a theoretical point of view.

The present short paper is subdivided into two sections. In the first section the new model is introduced and its general properties presented. In the second section the important problem of determinacy is discussed.

In this paper only an important subset of the class of the new schemata, referring to a family of "well-structured" programs, is analyzed. The new model is well suited for automatically detecting parallelism; related techniques will be presented in a forthcoming paper.

I. The model

The model presented in this paper is described by three sets of elements: 1) the OP-set, namely, the set of operations and predicates used by the system; 2) the graph, describing the parallel program to be executed; 3) the interpretation, which specifies the operations and the predicates.

1. The OP-set

The OP-set $\underline{S}$ [a] is the union of two subsets $\underline{O}$ and $\underline{P}$:

$$\underline{S} = \underline{O} \vee \underline{P}$$

where

$$\underline{O} = \{o_1, o_2, \dots\}$$

will be called "operation set" and

$$\underline{P} = \{p_1, p_2, \dots\}$$

will be called "predicate set".

To each $o \in \underline{O}$ are associated:

a) a finite set

$$\underline{D}(o) \subset \omega$$

(where ω is the set of natural numbers), which will be called the "domain" of o;

b) a finite set

$$\underline{R}(o) \subset \omega$$

which will be called the "range" of o.

To each $p \in \underline{P}$ is associated a finite set

$$\underline{D}(p) \subset \omega$$

which will be called the "domain" of p.

Of course, the set

$$\underline{W} = [\cup\{\underline{D}(o) \mid o \in \underline{O}\}] \cup [\cup\{\underline{R}(o) \mid o \in \underline{O}\}] \cup$$
$$\cup [\cup\{\underline{D}(p) \mid p \in \underline{P}\}]$$

represents the memory cells used by the computation system.

2. The graph

The oriented graph G is characterized by a finite set of nodes

$$\underline{N} = \{n_1, n_2, \dots\}$$

and a finite of oriented arcs

$$\underline{A} = \{a_1, a_2, \dots\}$$

The nodes represent control units of the whole system, and the arcs the operations to be performed (or their corresponding processing units).

To each node $n \in \underline{N}$ are associated:

a) a mode $M(n)$, which is a binary specification indicating whether the node is of type AND or OR;

b) a set of predicates

$$\underline{P}(n) \subseteq \underline{P}$$

which may be missing, in which case the node will be called a "free node".

To each arc $a \in \underline{A}$ is associated an operation

$$o = O(a) ;$$

besides, if a is an output arc of node n and a set of predicates $\underline{P}(n)$ has been associated to n, then a predicate

$$P(a) \in \underline{P}(n)$$

will be assigned to a. (Generally, if $\{a_1, a_2, \dots, a_v\}$ is the set of the arcs going out of n,

$$\underline{P}(n) = \{P(a_1), P(a_2), \dots, P(a_v)\} \quad)$$

Fig. 1 shows an example of graph and illustrates some conventions used in the sequel. In particular, only the nodes OR will be labelled in the graph (using a sign +; see nodes n_4 and n_{15} in Fig. 1), the remaining nodes being assumed to be of type AND. Notice also that in Fig. 1 only the arcs going out of n_8 and n_{15} are assigned predicates; this implies that only the nodes n_8 and n_{15} are associated sets of predicates, coincident (unless otherwise specified) with the sets of predicates associated to the leaving arcs.

In the sequel, the set of arcs entering a node n will be indicated as $\underline{I}(n)$, and the set of arcs leaving n as $\underline{U}(n)$. Besides, the node which the arc a is leaving will be denoted with G(a) (generator of a) and the node which a is entering with S(a) (successor of a).

A formal definition of the behaviour of the system described by the graph will be given in a forthcoming paper. However, it may be useful to anticipate at this point in loose terms that a control section of type AND begins to work after the completion of all the operations associated to the entering arcs, while a section of type OR requires the completion of only one input operation. When a control section (that is, a node) has completed its activity, all the operations associated to the leaving arcs will be initiated, unless a set of predicates has been associated to that node and those arcs, in which case only the operations

[a] The symbols of sets will be always underscored.

associated to the arcs for which the predicates are satisfied will be initiated. Thus, the activity of the system shown in Fig. 1 will begin with the activity of the input nodes n_1 and n_2. After completion of the control activity of n_1, the operations associated to a_1 and a_2 will be performed in parallel; similarly, a_3 and a_4 will be initiated after the activity of n_2. When all a_1, a_2, a_3 and a_4 are completed, n_3 and then a_5 will be activated. Since n_4 is of type OR, the completion of the activity of a_5 will be sufficient to begin the operation of n_4; then all the operations associated to a_6 - a_{11} will be performed in parallel. And so on. Notice that the activity of n_{15} will be to calculate the predicates p_4 and p_5. If p_4 is satisfied, a_{12} will be activated and the loop to n_4 will be closed; if p_5 is satisfied, a_{13} and then n_{16} will be activated.

3. <u>The interpretation</u>

An oriented graph, each node and arc of which has been assigned an element of the OP-set, will be called a "<u>specified graph</u>". Since only specified graphs are of interest for our study, in the sequel the word "graph" will always be used for denoting a "specified graph". Besides, a (specified) graph will be called an "<u>interpreted</u>" graph, when an <u>interpretation</u>, that is, a suitable set of assignments of the type below defined, has been assigned.

An interpretation of a (specified) graph is a quadruple

$$I = (\underline{\Delta}, \delta_0, \underline{F}, \underline{G})$$

where:

(i) $\underline{\Delta}$ is the universe of the values assumed by the variables of our computation system;

(ii) $\delta_0 \in \underline{\Delta}^{\underline{W}}$ (b) is the initial assignment;

(iii) for each $o \in \underline{O}$

$$F_o : \underline{\Delta}^{\underline{D}(o)} \to \underline{\Delta}^{\underline{R}(o)}$$

is a total function, and

$$\underline{F} = \{F_o \mid o \in \underline{O}\} ;$$

(iv) for each $p \in \underline{P}$

$$G_p : \underline{\Delta}^{\underline{D}(p)} \to \{\text{TRUE, FALSE}\}$$

is a total function, and

$$\underline{G} = \{G_p \mid p \in \underline{P}\}$$

It is apparent that, if $\underline{W}$ the set of the memory cells used by the systems, δ_0 specifies the initial values contained in those cells, $\underline{F}$ the functions performed by the operations and $\underline{G}$ the predicates calculated in the branch points.

In any interpretation, for each not free node $n \in \underline{N}$ at least one of the functions G_p associated to the predicates of n must take the value TRUE. In loose terms, at least one of the arcs leaving n must always be activated after n.

The functions G_p associated to $\underline{P}(n)$ are not necessarily mutually exclusive, in the sense that more than one of the arcs leaving a not free node may be activated after that node. If functions G_p associated to $\underline{P}(n)$ are mutually exclusive for all the sets of values assigned to the domains $\underline{D}(p)$, the interpretation will be called "<u>mutually exclusive</u>" (ME) on n. The restriction to the class of the interpretations ME on some nodes leads sometimes to simpler statements than the general case and it is not a serious one because it is not difficult to prove that any (interpreted) graph can always be transformed into an equivalent (interpreted) graph which is ME on all the nodes of some subset of them.

(b) $\underline{\Delta}^{\underline{W}}$ denotes the set of the functions from $\underline{W}$ into $\underline{\Delta}$.

4. <u>Bigraphs and structures</u>

In order to restrict our attention to a class of graphs which can be easily treated and which covers most cases of practical interest, the following definitions are assumed.

A "<u>complex bigraph</u>" is a subgraph recursively defined by the following statements:

A. An oriented arc (associated to a given operation and, possibly, to a given predicate)(elementary bigraph of type A) is a complex bigraph.

B. A free and loop-free subgraph containing nodes of AND mode only, connected to other subgraphs by only two arcs, an input arc leading to a node which no other arc is entering, and an output arc leaving a node from which no other arc is leaving (elementary bigraph of type B; Fig. 2), is a complex bigraph.

C. A loop-free subgraph of the type indicated in Fig. 3 (elementary bigraph of type C), which is ME on node n_1 for every interpretation, is a complex bigraph.

D. A subgraph of the type indicated in Fig. 4 which is ME on n_2 for every interpretation (elementary bigraph of type D) is a complex bigraph.

E. A subgraph obtained from a complex bigraph by substituting for any arc a complex bigrapn (oriented in the same direction) and labelling its input arc with the same predicate as the substituted arc, is a complex bigraph. The arcs connecting a given bigraph to other subgraphs will be called the "input arc" and the "output arc", and will be denoted with a_I and a_U, respectively (cpr. figs. 2, 3, 4).

Besides, a free and loop-free graph composed of nodes of the only type AND will be called an "elementary structure". A graph obtained from an elementary structure by substituting for any arc a complex bigraph (oriented in the same direction) and labelling its input with the same predicate as the original arc will be called a "complex structure". An elementary structure will be considered as a particular complex structure.

The activity in the bigraphs and the structures is described by some basic theorems, which will be presented in a forthcoming paper.

II. <u>Determinacy</u>

1. <u>Definitions</u>

Our model must permit of the analysis and the simulation of parallel computation systems working asynchronously. The following definitions lead to the notion of determinacy, an important property for all the models of parallel programs since it guarantees the invariance of the computation results even under the hypothesis of a completely asynchronous operation.

Our concept of determinacy relies on the following definitions, here given rather informally, for the sake of brevity.

<u>Definition</u>

A graph will be said to be "determined" for a given interpretation I if for each cell of memory w the hystory $H_\alpha(w)$ of the contents of w is independent of the computation α considered, namely of the durations of the operations involved.

<u>Definition</u>

A graph will be said to be determined if it is determined for any interpretation.

2. <u>Determinacy of elementary bigrapns and structures</u>

Let us first consider elementary structures. On the set $\underline{A} \vee \underline{N}$

of the graph a relation $\geq$ is defined by the following statements:

A. If for an arc a and a node n

$$a \in \underline{I}(n)$$

then

$$a \geq n$$

B. If for an arc a and a node n

$$a \in \underline{U}(n)$$

then

$$n \geq a$$

C. The relation $\geq$ is transitive.

Two elements (s_1, s_2) of $\underline{A} \vee \underline{N}$ such that neither $s_1 \geq s_2$, nor $s_2 \geq s_1$ will be called "uncomparable elements".

Theorem 1

An elementary structure is determined if, and only if, the following condition is verified.

Condition α). If two arcs a_1 and a_2 are uncomparable, then

$$[R\ (O(a_1)) \cap R(O(a_2))] \vee [R\ (O(a_1)) \cap D(O(a_2))] \vee$$
$$\vee [R\ (O(a_2)) \cap D(O(a_1))] = \emptyset$$

where, as usual, O(a) denotes the operation associated to arc a, and D(o) and R(o) denote the domain and the range of operation o, respectively.

As a particular case, Theorem 1 holds also for elementary bigraphs of type B:

Theorem 2

An elementary bigraph of type B is determined, if, and only if, the condition α of Theorem 1 is verified.

Besides, the following theorems hold.

Theorem 3

An elementary bigraph of type C is always determined.

Theorem 4

An elementary bigraph of type D is always determined.

3. Equivalence

On the set of the (specified but not interpreted) complex bigraphs and structures it is possible to define an equivalence relation which will result usefull for determinacy conditions. Such equivalence relation is defined by the following statements.

A. An elementary bigraph B of type B is equivalent to an oriented arc a defined as follows:

A1) Its direction coincides with the direction of the input arc (or the output arc) of the assigned bigraph B.

A2) The operation associated to a is a dummy operation O(a).

A3) The domain of O(a) is

$$D\ (O(a)) = \cup \{D\ (O(a))\ /\ a \in \underline{A}_B\}$$

where $\underline{A}_B$ denotes the set of the arcs of B.

A4) The range of O(a) is

$$R\ (O(a)) = \cup \{R\ (O(a))\ /\ a \in \underline{A}_B\}$$

A5) If the input arc of B is labelled with a predicate, the same predicate must label arc a.

B. An elementary bigraph B of type C is equivalent to an oriented arc a defined as follows:

B1) Its direction coincides with the direction of the input arc (or the output arc) of the assigned bigraph B.

B2) The operation associated to a is a dummy operation O(a).

B3) The domain of O(a) is

$$D\ (O(a)) = [\cup \{D\ (O(a))\ /\ a \in \underline{A}_B\}] \cup [\cup \{D(p_i)\ /\ p \in \underline{P}\}]$$

where $\underline{A}_B$ is the set $a_1, a_2, \ldots, a_f, a_l, a_u$ of the arcs of B and $\underline{P} = p_1, p_2, \ldots, p_f$ is the set of predicates associated to n_1.

B4) The range of D(a) is

$$R\ (O(a)) = \cup \{R\ (O(a))\ /\ a \in \underline{A}_B\}$$

where $\underline{A}_B$ is again the set of the arcs of B.

B5) If the input arc of B is labelled with a predicate, the same predicate must label arc a.

C. An elementary bigraph B of type D is equivalent to an oriented arc defined as follows.

C1) Its direction coincides with the direction of the input arc (or the output arc) of the assigned bigraph B.

C2) The operation associated to a is a dummy operation O(a).

C3) The domain of O(a) is

$$D\ (O(a)) = [\cup \{D\ (O(a))\ /\ a \in \underline{A}_B\}] \cup D(p_1) \cup D(p_2)$$

where $\underline{A}_B$ is again the set $\{a_l, a_u, a_r, a_f\}$ of the arcs of B.

C4) The range of O(a) is

$$R(O(a)) = \cup \{R\ (O(a))\ /\ a \in \underline{A}_B\}$$

C5) If the input arc of B is labelled with a predicate, the same predicate must label arc a.

D. The complex bigraph B_2 (or the complex structure S_2), obtained from a complex bigraph B_1 (or a complex structure S_1) by substituting in B_1 (S_1) any elementary bigraph with its equivalent arc, is equivalent to B_1 (S_1).

E. Each complex bigraph or complex structure is equivalent to itself (reflexivity).

F. If B_1 (or S_1) is equivalent to B_2 (or S_2), then B_2 (S_2) is equivalent to B_1 (S_1) (symmetry).

G. If B_1 (or S_1) is equivalent to B_2 (S_2) and B_2 (S_2) is equivalent to B_3 (S_3), then B_1 (S_1) is eqivalent to B_3 (S_3) (transitivity).

Notice that, owing to conditions E, F, G, our notion of equivalence defines a relation of equivalence in the usual sense.

4. Determinacy of complex bigraphs and structures

Determinacy of complex bigraphs and structures can be easily verified, by applying the theorems presented in this subsection. Four distinct cases are to be considered separately.

A. Complex bigraph of type B

A complex bigraph which can be thought of as an elementary bigraph of type B in which a set $\underline{B} = \{A_1, B_2, \ldots\}$ of complex bigraphs have been substituted for a set of arcs will be called "a complex bigraph of type B". By definition, by substituting again equivalent arcs for the component complex bigraphs we can obtain an elementary bigraph of type B, which will be called the "equivalent elementary bigraph of the assigned complex bigraph with respect to $\underline{B}$".

The relationship between the two equivalent bigraphs is illustrated by the following theorem.

Theorem 1

A complex bigraph of type B is determined if, and only if, its equivalent elementary bigraph with respect to the set $\underline{B}$ of component bigraphs is determined, and each element of $\underline{B}$ is determined.

B. Complex bigraph of type C

A complex bigraph which can be thought of as an elementary bigraph of type C in which a set $\underline{B} = \{B_1, B_2, \ldots, B_g\}$ of complex bigraphs have been substituted for the set $\{a_1, a_2, \ldots, a_g\}$ of internal arcs will be called "a complex bigraph of type C".

By definition, by substituting again an equivalent arc for each $B_j \in \underline{B}$ we obtain an elementary bigraph of type C, which will be called the "equivalent elementary bigraph of the assigned complex bigraph with respect to $\underline{B}$".

Theorem 2

A complex bigraph of type C is determined if, and only if, each element $B_j \in \underline{B}$ is determined.

C. Complex bigraph of type D

A complex bigraph which can be thought of as an elementary bigraph of type D in which the complex bigraph B_g and B_r have been substituted for the internal arcs a_g and a_r, will be called a "complex bigraph of type D". By definition, by substituting the equivalent arcs a_g and a_r for B_g and B_r, respectively, we obtain an elementary bigraph of type D, which will be called "the equivalent elementary bigraph" of the assigned complex bigraph with respect to $\{B_g, B_r\}$.

Theorem 3

A complex bigraph of type D is determined if, and only if, both B_g and B_r are determined.

D. Complex structure

By definition, a complex structure can be transformed into an elementary structure by substituting a set $\underline{B} = \{B_1, B_2, \ldots\}$ of complex bigraphs with the set of the corresponding equivalent arcs. Such elementary structure will be called "the equivalent elementary structure of the assigned complex structure with respect to $\underline{B}$". Theorem 1 extends to this case as follows.

Theorem 4

A complex structure is determined if, and only if, its equivalent elementary structure is determined, and each $B \in \underline{B}$ is determined.

5. An algorithm for verifying determinacy

The theorems presented in the preceding subsection suggest an algorithm which makes it possible to easily verify whether or not a given (complex) structure (or bigraph) is determined. An outline of this algorithm is presented below.

A. The assigned graph is transformed into a complex structure by simple rules. For example, if more than one arc is entering a not free node, a dummy arc (with empty domain and range) and a dummy node are introduced.

B. The complex structure is transformed into an equivalent elementary structure with respect to a suitable set of bigraphs $\underline{B}$.

C. Each complex bigraph $B \in \underline{B}$ is transformed into its equivalent elementary bigraph with respect to a new subset of complex bigraphs. The procedure is iterated until simple arcs or elementary bigraphs are obtained.

D. The preceding decompositions are used for determining the equivalent arc of each complex bigraph above determined. It is so possible to apply the theorems stated in the preceding subsection 4 for verifying the determinacy of each component bigraph and the assigned complex structure.

REFERENCES

[1] Ashcroft, E. and Z. Manna, Formalization of Properties of Parallel Programs. Machine Intelligence, vol. 6, New York (1971).

[2] Holt, A. W. e Commoner, F., Events and Conditions - Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation ACM, New York (1974), 3-52.

[3] Kahn, G., A Preliminary Theory for Parallel Programs - Laboratoire de recerche en informatique et automatique - Rapport de recherche n° 6 (1973).

[4] Karp, R. M., and Miller, R. E., Parallel Program Schemata - J. Comput. Syst. Sci. 3, 2 (1969), 147-195.

[5] Keller, R. M., Parallel Program Schemata and Maximal Parallelism, J. ACM 20, 3 (1973),514-537; 4 (1973), 696-710.

[6] Petri, C.A., Communication With Automata, Supplement 1 to Technical Report RADC-TR-65-377, Vol. 1, Griffiss Air Force Base, New York (1966).

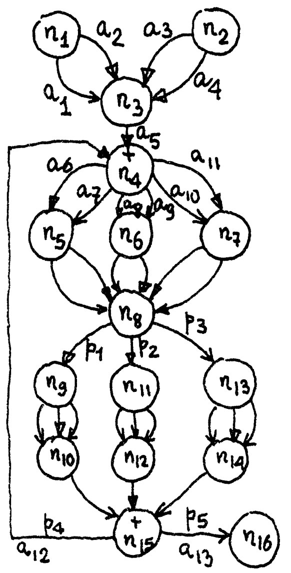


Fig. 1 - An example of graph.

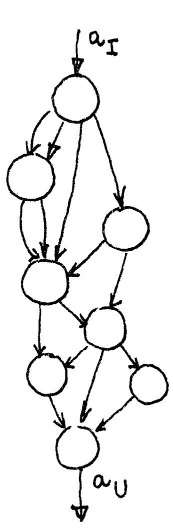
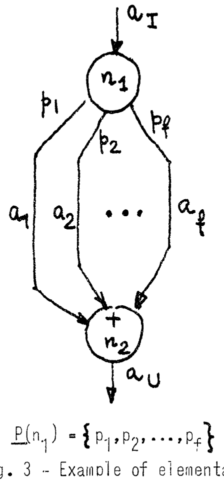
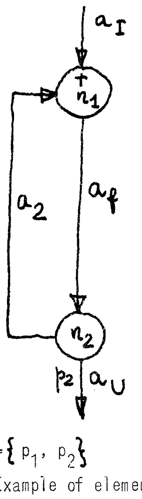


Fig. 2 - Example of elementary bigraph of type B.



$\underline{P}(n_1) = \{p_1, p_2, \ldots, p_f\}$

Fig. 3 - Example of elementary bigraph of type C.



$\underline{P}(n_2) = \{p_1, p_2\}$

Fig. 4 - Example of elementary bigraph of type D.

FORTRAN EXTENSION DESIGN CONCEPTS FOR ASSOCIATIVE PROCESSING

Edward B. Allen, Arvid G. Larson

Applied Research Group
PRC Information Sciences Company
7600 Old Springhouse Road, McLean, Virginia 22101

Summary

This paper presents some high-level language design concepts [2] - [4] and applies them to a FORTRAN extension for a generalized associative processor (AP) such as is described in [4]. Reference [1] is a more detailed presentation.

While it is not necessary that the high-level programmer have a detailed understanding of hardware operation, it is necessary that the programmer's machine concept be adequately defined. We employ the following concepts: the AP is a "slave" to a "master" sequential processor (SP). The content addressable memory (CAM) of the AP is divided into modular arrays. The CAM includes a logical "response" vector indicating which array elements have the desired content. Other AP numerical/logical operations are functionally similar to those of the SP host.

The functional requirements of a language must be defined in terms of the computational capabilities which it must support. Our FORTRAN extension exploits AP capabilities in parallel numerical calculation, manipulation of logical arrays, and searches of a data base.

Process control and functional control concepts must also be defined, the former stemming from the programmer's machine concept, the latter following from the required computation capabilities. Finally, a syntax must be defined; here the language designer should choose necessary semiotics by considering "What is natural for the programmer?" We favor AP language designs based on extension of existing languages, where "natural for the programmer" is defined as the base language programming conventions.

Extension of an existing language has several advantages (1) The programmer need learn relatively few new concepts. (2) Applications which exploit the SP and AP in the same system can utilize the same basic language in both processors with minimal programmer concern for hardware interfaces. (3) Development costs are likely to be less, since implementation can be based on existing compiler and ancilliary software.

In the area of process control, our extension design adapts FORTRAN subroutine concepts to control subprograms executing in the SP and AP; the AP subprogram is a subroutine of the SP main program. Subroutine arguments can be used to invoke the transfer of data between SP memory and CAM. Since AP computational efficiency is highly dependent on the layout of variables in CAM, the extension requires the programmer to explicitly assign variables to CAM locations.

Functional control extensions are illustrated by several new statements. A PARALLEL statement, modeled on a DO, defines a range of lines where statements with an embedded index i are executed for all values of i in parallel. Thus, parallel execution of the same statement on different array elements is explicitly controlled by the programmer. An IF statement within the range can include an embedded i in the logical condition to indicate a test performed in parallel for all values of i. The THEN clause will be executed only for .TRUE. values of i; subsequently the ELSE clause is executed only for .FALSE. values of i.

A "FIND arrayname, (logical expression)" statement sets bits of the CAM "responder" according to the value of the logical expression for each element of "arrayname", where "arrayname" is embedded in the logical expression. This in effect finds elements of "arrayname" which satisfy the logical expression. Other high-level operations are also defined, such as ".ANY. (logical expression)" and ".ALL. (logical expression)" which return a single logical value, where an "arrayname" is embedded in the logical expression, and such as "ROTATE" and "SHIFT" of arrays.

References

[1] E.B. Allen, and A.G. Larson, A FORTRAN Extension for Associative Processing, Technical Report D-1875, PRC Information Sciences Company, McLean, Va., (June 1975).

[2] C.R. DeFiore, A.A. Vito, and L. Baner, "Toward the Development of a Higher Order Language for RADCAP", Proc. Sagamore Computer Conf. (August 1972), pp. 99-112.

[3] W.W. Patterson, "Some Thoughts on Associative Processing Languages", National Computer Conference, (May 1974), pp. 23-26.

[4] H.K. Resnick, and A.G. Larson, "A COBOL Extension for Associative Array Processors", Proc. Conf. on Programming Languages and Compilers for Parallel and Vector Machines, pub. as SIGPLAN Notices 10, 3, (March 1975), pp. 54-61.

PARALLEL EXECUTION ON ARRAY AND VECTOR COMPUTERS

Leslie Lamport
Massachusetts Computer Associates, Inc.
Wakefield, Massachusetts 01880

Abstract -- The DO SIM loop was introduced as a natural way of expressing parallel execution on an array or vector computer. However, it presents implementation problems for an array computer with fewer processors than the number of elements in the index set, for vector computers, and for ordinary sequential computers. A DO SYNC loop is defined to be a special type of DO SIM loop suitable for execution on a single instruction stream array computer, on certain types of single pipe vector computers, and on sequential computers. A compiler for such a computer may have to rewrite a DO SIM loop as one or more DO SYNC loops. A condition is given for deciding if a DO SIM loop is also a DO SYNC loop. It shows that the DO SIM loops obtained from sequential DO loops by previous automatic translation techniques are actually DO SYNC loops.

The problem of implementing a DO SIM loop on more complicated machines, such as multiple pipe vector computers, is also considered. A condition is given for determining if an implementation is correct. It can be used by the compiler in generating code for a DO SIM loop.

Introduction

Several parallel array and vector computers have appeared, including the Illiac IV, the CDC Star-100, and the Texas Instruments ASC. These computers require some means of expressing parallel execution in a high level programming language. One method is with the DO SIM loop, defined below. It is essentially equivalent to several other proposed constructions. This paper discusses the general problem of executing a DO SIM loop on an array or pipelined vector computer. The results are useful in designing a compiler for such a computer.

A special type of DO SIM loop called a DO SYNC loop is introduced. For execution on a single instruction stream array computer, the compiler may have to write a DO SIM loop as several DO SYNC loops. A condition is given to determine if a DO SIM loop is also a DO SYNC loop. It is shown that the DO SIM loops produced by the coordinate method of [1, 2] are actually DO SYNC loops.

More general types of parallel computers are also considered, including multiple pipe vector computers. A general condition is given to determine if an implementation of a DO SIM loop on such a computer is correct.

Even when a program is written for a parallel computer, it will sometimes be desirable to execute it on a sequential computer. It is shown that executing a DO SIM loop on an ordinary sequential computer is essentially the same problem as executing it on a single instruction stream array computer.

The DO SIM Loop

The DO SIM loop was defined in [1-3]. Three examples are given in Figure 1.

```
(A)    DO 1 SIM FOR ALL I ε {1, ..., 256}
       A(I) = A(I) + B(I)
   1   CONTINUE
(B)    DO 1 SIM FOR ALL I ε {1, ..., 256}
       A(I) = A(I + 1) + B(I + 1)
   1   CONTINUE
(C)    DO 1 SIM FOR ALL I ε {1, ..., 256}
       A(I + 1) = A(I) + B(I)
   1   CONTINUE
```

Figure 1

(In [3], the word "SIM" is omitted, and the set is specified by a boolean array expression.) Loop C, for example, has the following effect. For each $i = 1, \ldots, 256$: the new value of A(i+1) is set equal to B(i) plus the old value of A(i) . Loop C might also be expressed in the following different notation.

$$* \leftarrow \{1, \ldots, 256\}$$
$$A(* + 1) \leftarrow A(*) + B(*)$$

In general, we consider the following loop.

(1) DO α SIM FOR ALL $(I^1, \ldots, I^k) \in \mathbb{S}$

$$\boxed{\text{loop body}}$$

α CONTINUE

where $\mathbb{S}$ is a subset of the set $\mathbb{Z}^k$ of all k-tuples of integers. Loop (1) has the following meaning. To each element $(i^1, \ldots, i^k) \in \mathbb{S}$, assign a separate processor and have it set $I^1 = i^1, \ldots, I^k = i^k$. All the processors then execute the loop body simultaneously, operating in lock step. E.g., in the loops of Figure 1, all processors first perform the addition, then they all perform the store.

In practise, the nature of the computer design will place certain restrictions on the form of $\mathbb{S}$ and on what may appear in the loop body. More is said about this below.

In [1], we also defined a DO CONC loop. It is similar to the DO SIM loop, except that the individual processors are assumed to operate asynchronously, completely independent of one another. Loop A is actually a DO CONC loop, since different executions of the loop body all reference different data. However, Loops B and C are not DO CONC loops.

The DO SIM construction is a natural

means of expressing parallelism for an array computer. It seems easiest to think in terms of this type of parallelism. Although it is called a "loop", the body is actually executed just once, simultaneously for all values of the index variables. However, it presents one major implementation problem. Suppose we want to execute the loops of Figure 1 on an array computer with only 64 separate processors. We would like to replace the DO SIM I statements with the following:

```
DO 1 J = 0, 3
DO 1 SIM FOR ALL I ∈ {1+J*64, ..., 64+J*64} .
```

This form of replacement is called strip mining. Loops A and B can be strip mined in this way, but loop C cannot. E.g., strip mining loop C would yield a loop in which the assignment of A(64) + B(64) to A(65) would use the new value of A(64) instead of the old value.

(Loop C can actually be strip mined in the reverse direction, using a "DO 1 J = 3, 0, -1" loop. However, it is easy to construct DO SIM loops which cannot be strip mined in either direction. For now, we ignore the possibility of reverse strip mining.)

Strip mining has two advantages: (1) it minimizes loop initialization costs, and (2) it minimizes the amount of temporary storage needed. For example, loop C would actually be executed as follows:

```
    DO 11 J = 0, 3
    DO 11 SIM FOR ALL I ∈ {1+64*J, ..., 64+64*J}
 11 TEMP(I) = A(I) + B(I)
    DO 1 J = 0, 3
    DO 1 SIM FOR ALL I ∈ {1+64*J, ..., 64+64*J}
  1 A(I + 1) = TEMP(I)   .
```

This involves twice the loop setup time and four times the temporary storage space as the strip mined version of loops A and B. Hence, we want to strip mine a DO SIM loop whenever possible.

It is clear that any DO CONC loop can be strip mined, since it can be executed by completely unsynchronized processors. Loop B indicates that there is a class of loops which are not DO CONC loops but which can still be strip mined. We will define these to be DO SYNC loops.

The DO SYNC Loop

Before defining the DO SYNC loop, we introduce some terminology. An occurrence of a variable in a program is said to be a generation if it causes the assignment of a new value to the variable, otherwise it is called a use. The execution of a use consists of fetching the indicated value from memory. The execution of a generation consists of storing the value into memory. Let f and g be two variable occurrences in a loop body. We say that f precedes g if for a single execution of the loop body, f may be executed before g is. (If there is a loop inside the loop body, then it is possible for f and g to each precede the other.)

A DO SYNC loop has the following form,

(2) DO α SYNC FOR ALL $(I^1, \ldots, I^k) \in \mathcal{S}$

loop body

α CONTINUE

where $\mathcal{S}$ is now assumed to be an ordered subset of $\mathbb{Z}^k$. It is defined in a similar manner to loop (1), except that the individual processors are not assumed to operate in lock step. Instead, they are merely assumed to satisfy the following synchronization condition. Let f and g be any two variable occurrences in the loop body such that f precedes g, and let $P, Q \in \mathcal{S}$ with $P \leq Q$. Then the processor assigned to P must execute f before the processor assigned to Q executes g.

Loop B can be executed as a DO SYNC loop. For example, the above restriction implies that the processor with I = 65 cannot assign a new value to A(65) before the processor with I = 64 fetches the old value to compute A(65) + B(65). (We are assuming here that we use the usual ordering on the set of integers.)

Returning to the general loop (2), suppose $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$, and for each $i = 1, \ldots, n-1$ and all $P \in \mathcal{S}_i$, $Q \in \mathcal{S}_{i+1}$ we have $P < Q$. Then the loop (2) is equivalent to the following:

DO α J = 1, N

DO α SIM FOR ALL $(I^1, \ldots, I^k) \in \mathcal{S}_J$

loop body

α CONTINUE .

In other words, a DO SYNC loop can always be strip mined.

We do not propose that the DO SYNC loop be adopted for writing parallel programs. Rather, it provides a useful semantic concept for the compiler. The programmer should write DO SIM loops. The compiler for an array computer must translate them to DO SYNC loops if strip mining is necessary. For loops A and B, the compiler can merely replace the DO SIM by a DO SYNC. Loop C must be rewritten as follows:

```
    DO 11 SYNC FOR ALL I ∈ {1, ..., 256}
11  TEMP(I) = A(I) + B(I)
    DO 1 SYNC FOR ALL I ∈ {1, ..., 256}
 1  A(I + 1) = TEMP(I) .
```

This is all quite obvious for the simple one-dimensional loops of Figure 1. However, it becomes more subtle for higher dimensional loops such as the following.

DO 1 SIM FOR ALL (I, J) $\in \mathcal{S}$

1 A(I + 1, J - 1) = A(I, J) + B(I, J)

A general rule for deciding if a DO SIM may be replaced by a DO SYNC is given below. It has an important corollary. A technique for translating sequential DO loops into DO SIM loops, called the coordinate method, was described in [1] and [2]. The general rule shows that the coordinate method actually produces DO SYNC loops.

Sequential Computers

It may be desirable to compile DO SIM loops for execution on a sequential computer. For example, an array or vector computer will usually be more expensive than a sequential computer, so it might be better to debug a program on a sequential computer before running it on a more powerful parallel computer.

A sequential computer can simply be thought of as an array computer with just one processor. Hence, the compiler must rewrite the DO SIM loop as one or more DO SYNC loops. The DO SYNC loops can then be executed sequentially like ordinary DO loops.

Vector Computers

To the user, a pipelined vector computer like the Star-100 or the ASC appears similar to an array computer. The DO SIM loop is a reasonable way of expressing vector operations for such a computer. However, implementing a DO SIM loop on a vector computer is a more general problem than implementing it on an array computer.

The execution of one of the loops of Figure 1 on a pipelined vector computer is indicated schematically in Figure 2. The buffers, arithmetic unit and result register form a <u>pipe</u>. Calculation of several different sums would actually be taking place concurrently within the pipe. However, the final results are produced one at a time as shown.

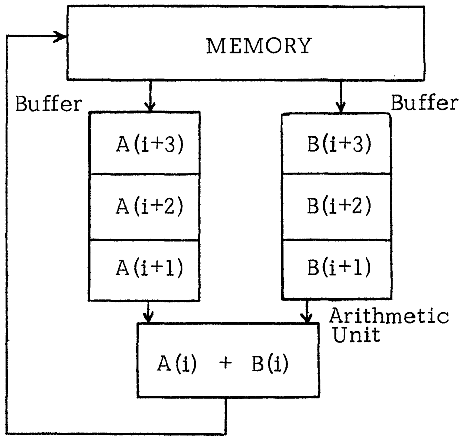


Figure 2

Loops A and B can be executed as shown in Figure 2 with no problem. Loop C can also be executed in this way if the pipe never became empty - i.e., if there is always some value in the left hand buffer or in the result register. However, if the flow of new arguments is stopped long enough for the pipe to empty itself by storing results in memory, perhaps by an external interrupt, then an error will occur.

A vector computer may contain several separate pipes. These might be synchronized with one another in some fashion, or might operate asynchronously. To see the problems which this raises, consider the following example.

```
    DO 2 SIM FOR ALL I ε {i : 1 ≤ i ≤ 256}
1   C(I) = A(I) + B(I)
2   A(I + 1) = D(I)**2
```

A two pipe computer could execute the two statements concurrently -- provided the two pipes are properly synchronized so a value generated by statement 2 is not stored in memory before its previous value is used in statement 1.

We can also envision an array computer capable of concurrent execution of several different array operations. The problem of executing a DO SIM loop on such a computer is similar to that of executing it on a multiple pipe vector computer.

We will formulate the problem of correctly implementing a DO SIM loop in a sufficiently general way to be applicable to these types of computers. A validity condition will be given which determines if an implementation is correct. It can be used by the compiler to generate the best code possible.

Assumptions

Efficient parallel execution on an array or vector computer is possible only under certain restrictions on the way operands are stored in memory. Execution of loop A with the Star-100 vector operations requires that each element A(i) be stored in the memory location adjacent to A(i+1). For the Illiac IV, A(i) and A(i+1) must be stored in the memories of adjacent processors, in some regular fashion. E.g., A(j) might be stored in memory location 2j of processor number j mod 64.

Because of these restrictions, we may make certain assumptions about the specification and execution of a DO SIM loop. Define a <u>rectangular</u> set in $\mathbb{Z}^k$ to be a set of the form $[\ell^1, u^1] \times \ldots \times [\ell^k, u^k]$, where $[\ell^i, u^i] = \{x : \ell^i \leq x \leq u^i\}$. We assume that the set $\mathcal{S}$ of loop (1) is of the form $\mathcal{S} = \{P \in \mathcal{R} : \ldots\}$, for some rectangular set $\mathcal{R}$. We further assume that the variable occurrences in the loop are executed for all the elements of $\mathcal{R}$, but the effects of the execution are inhibited for the elements not in $\mathcal{S}$. For example, consider the loop

```
  DO 1 SIM FOR ALL I ε {i ε [1,256] : C(i) ≠ 0}
1 A(I) = B(I) .
```

We assume that the use B(I) and the generation A(I) are executed 256 times, except that the value of A(i) is not actually stored in memory if C(i) = 0 .

This is an accurate description of the way execution actually takes place in an array or vector computer. For an empty index set $\mathcal{S}$, an array computer might simply skip over the loop body. Hence, some occurrence executions might actually be skipped in a strip mined execution of the loop. However, we may pretend that these executions take place.

Our problem arises from the fact that instead of being simultaneous, the different executions of an occurrence are actually done in some order. This is indicated by an ordering relation $<$ on the set $\mathcal{R}$. We assume that the ordering of $\mathcal{R}$ can be expressed with the aid of a linear transformation $\sigma : \mathbb{Z}^k \to \mathbb{Z}$ which is a 1-1 mapping from $\mathcal{R}$ onto an interval $[\ell, u]$, such that $P < Q$ if and only if $\sigma(P) < \sigma(Q)$. The execution of an occurrence for P is assumed to precede its execution for Q if and only if $P < Q$. For an array computer, in which some executions actually do occur simultaneously, we can pretend that they take place in very rapid succession in the indicated order.

For example, suppose $k = 3$ and $\mathcal{R} = [1, L] \times [1, M] \times [1, N]$. Then the usual lexicographical ordering on $\mathbb{Z}^3$ is obtained if σ is defined by $\sigma(x, y, z) = MNx + Ny + z$.

For the loops of Figure 1, we can let σ be either the identity mapping or else the mapping $\sigma(i) = -i$. They represent the two ways of going through the index set: forwards and backwards.

The Validity Condition

Let P and Q be two elements of $\mathcal{R}$. For a single occurrence f, we know that the execution of f for P precedes its execution for Q if and only if $\sigma(Q-P) > 0$. In that case, the execution for Q is the $\sigma(Q-P)$th one after the execution for P.

Now consider two distinct occurrences f and g such that f precedes g. To determine the correctness of the implementation of a DO SIM loop, we need to consider the following question: does the execution of f for P precede the execution of g for Q? Since f precedes g, the architecture of array and vector computers implies that the answer will be "yes" if $\sigma(Q-P)$ is large enough. Let us define the number $f\#g$ to be the smallest number such that <u>if</u> $\sigma(Q-P) > f\#g$ <u>then</u> the execution of f for P must precede the execution of g for Q. In other words, we have

$$f\#g = 1 + \text{maximum}\ \{\ \sigma(Q-P) : \text{execution of } g \text{ for } Q \text{ precedes the execution of } f \text{ for } P\text{, for some } P, Q \in \mathcal{R}\}\ ,$$

where the maximum of the empty set is assumed to be $-\infty$.

The value of $f\#g$ depends upon the implementation. For a single instruction stream array computer, $f\#g$ always equals zero if the loop is strip mined, and $-\infty$ if it is not. For the vector computer of Figure 2, suppose f and g are the use and generation of A, respectively. Let r be the minimum number of values which can be in a buffer and in the result register during execution of the vector operation -- neglecting the initial filling and the final emptying of the pipe. Then $f\#g = -r$.

If f and g are occurrences which are executed by completely asynchronous separate pipes in a multiple pipe vector computer, then we cannot answer the question of whether an execution of f will precede an execution of g. Hence, we must assume that $f\#g = \infty$.

The assumption about the synchronization of processors in a DO SYNC loop is equivalent to the assumption that for every pair of occurrences f, g such that f precedes g: $f\#g$ must be ≤ 0.

Let f, g be two occurrences of the same variable. As in [2], we define $\ll f, g \gg$ to be the set of all elements $X \in \mathbb{Z}^k$ for which there is an element $P \in \mathbb{Z}^k$ such that P and $P + X$ are elements of $\mathcal{S}$ and the executions of f for P and of g for $P + X$ reference the same array element. If f is the use and g the generation of A in loop C, then $\ll f, g \gg = \{-1\}$.

The set $\ll f, g \gg$ may only be known at run time. We assume that the compiler constructs a set $< f, g >$ which contains $\ll f, g \gg$. Note that for a DO SIM loop to be legal, we must have $< g, g > = (0, \ldots, 0)$ for every generation g.

For the correct execution of a DO SIM loop, it is necessary to maintain the specified ordering of references to any single array element when one of the references is by a generation. E.g., in loop C, it is necessary to guarantee that the reference to any element A(i) by the use A(I) precedes the reference to that element by the generation A(I+1).

We know that the execution of an occurrence for the element $P+X \in \mathcal{R}$ is the $\sigma(X)$th one after the execution for P. (The linearity of σ is used here.) It is easy to verify that the following condition is therefore sufficient to insure the correct execution of a DO SIM loop.

> (VC) For every pair of occurrences f, g of a single variable, at least one of which is a generation, such that f precedes g in the loop body, and for every element $X \in < f, g >$: we must have $\sigma(X) \ge f\#g$.

To apply condition (VC) to DO SYNC loops, we simply replace $f\#g$ by zero and recall that $\sigma(X) \ge 0$ if and only if $X \ge 0$. This yields the following condition for a DO SIM loop to be a DO SYNC loop.

> (VS) For every pair of occurrences f, g of a single variable, at least one of which is a generation, such that f precedes g in the loop body, we must have $X \ge 0$ for all $X \in < f, g >$.

For the DO SIM loops constructed by the coordinate method of [1], we can have $X \in < f, g >$ and $X < (0, \ldots, 0)$ if and only if g precedes f, where $>$ denotes the lexicographical ordering of k-tuples. Those DO SIM loops therefore satisfy (VS). Hence, the coordinate method of [1] produces DO SYNC loops. (Note that the set $< f, g >$ we have been using here is calculated just for the DO SIM loop. It equals $\{(i^1, \ldots, i^k) : J(X) = (0, \ldots, 0, i^1, \ldots, i^k)$ for some $X \in < f, g > \}$, where J and the latter $< f, g >$ are defined as in [1].)

The general coordinate method of [2] is somewhat more complicated, since it can create a DO SIM loop having DO loops inside its loop body. The definition of the DO SYNC loop can be generalized to include such loops, and a similar analysis shows that the general coordinate method also produces DO SYNC loops. In particular, those loops may be strip mined on an array computer. For execution on a vector computer, it is necessary to move the DO SIM loop inside any inner DO loops, splitting it into several loops if necessary. The resulting DO SIM loops will also be DO SYNC loops.

Application of the Condition

Let us now consider the application of rules (VC) and (VS) to the implementation of the DO SIM loop (1). The relation $<$ and the mapping σ will be implied by the conventions of the programming language. In IVTRAN [3], for example, σ is determined by the allocation of the boolean array which specifies $\mathcal{S}$.

There may be several possible choices for σ . On the Illiac IV, one can use both σ and $-\sigma$, representing the possibility of forward and backward strip mining. For the ASC there can be several choices of σ , some of which are more efficient than others. They represent the various ways of using the vector parameter registers to step through an array. For a sequential computer, a choice of σ represents a way of stepping through the set $\mathcal{R}$ with a nest of DO loops.

For a single instruction stream array computer (or a sequential computer), if the computer has fewer processors than there are elements in $\mathcal{R}$, then the compiler will try to write (1) as a DO SYNC loop. This can be done if (VS) is satisfied by one of the possible choices of the relation $<$. If not, then (1) must be rewritten as a sequence of DO SYNC loops, each of which satisfies (VS). Note that if f precedes g in the loop body and $X < 0$ for some $X \in \langle f, g \rangle$, then f and g may not be placed inside the same forward strip mined loop. However, if $X < 0$ for all $X \in \langle f, g \rangle$, then f and g can be placed inside the same backward strip mined loop. (I.e., we can replace $<$ by the inverse ordering.)

Designing a general algorithm to produce an optimal splitting of the loop is a non-trivial task. However, it should be possible to find a simple procedure that is adequate. For example, it is easy to do if one does not split individual statements or change the statement order.

For a pipelined vector computer, (VC) provides a legality condition which must be satisfied by the compiled code. For each pair of occurrences, f, g, (VC) determines the maximum permissable value of f#g . Making f#g small enough may require inserting some delay between the execution of f and of g . This delay may be effected by introducing temporary storage. E.g., for the one pipe computer of Figure 2, we could implement loop C as follows:

```
      DO 1 SIM FOR ALL I ε {1, ..., 256}
      TEMP(I) = A(I) + B(I)
  1   A(I + 1) = TEMP(I) .
```

Note that a violation of (VC) by f, g when g is a use can only happen in a multiple pipe computer.

Conclusion

We have discussed the problem of compiling a DO SIM loop. For a single instruction stream array computer, a sequential computer, or a vector computer with a single pipe which can be emptied by an interrupt, the compiler needs to generate DO SYNC loops. Condition (VS) provides a sufficient condition for a DO SIM loop to be a valid DO SYNC loop. It implies that the coordinate method for the parallel execution of sequential DO loops actually produces DO SYNC loops. For a more general multiple instruction stream array computer or multiple pipe vector computer, the compiler must use condition (VC) to determine if an implementation of a DO SIM loop is correct.

References

[1] L. Lamport, The Parallel Execution of DO Loops. Comm. ACM 17, 2 (February, 1974).

[2] L. Lamport, The Coordinate Method for the Parallel Execution of DO Loops. Proceedings of the 1973 Sagamore Computer Conference on Parallel Processing, Syracuse University, August 22-23, 1973, IEEE, New York, 1973, p. 1-12.

[3] R. Millstein, Control Structures in Illiac IV FORTRAN. Comm. ACM 16, 10 (October, 1973), pp. 621-627.

FORMALIZING CODE GENERATION IN THE MULTI MINI COMPUTER COMPILER[†]

C.V. Ramamoorthy and P. Jahanian
Computer Science Division
Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California at Berkeley
Berkeley, California 94720

Abstract -- This paper provides a formalism for generating the appropriate code for a number of different mini-computers, when provided with a single higher level language together with the parameters of the desired target machine. The multi-mini computer compiler isolates the generation of the object code from the syntax and semantic analyses of the source language by providing a machine independent intermediate language. With this intermediate language, the compiler can produce code in the target machine assembly language by analyzing information provided by the syntactical entity called the machine specification language. Specification language instructions, containing the target machine description, are first preprocessed by the compiler, and internal tables are set up. Then, the automatic code generator, an integral part of the compiler, can produce the object code by following the information contained in these tables.

Introduction

The basic objective in the development of the Multi Mini Computer Compiler (MMCC) is to directly translate a high level language to a number of target machines (mini computers). This objective implies that the logic involved in the code generation module of the compiler should be formalized. Formalization is done by providing the target machine description in a notation which is mathematically sound and is processable by the computer. The code generation module (called the Meta Assembler) processes the formal description of the target machine and then adapts itself to the new environment for the generation of object code.

Formalization of code generation is assisted by two specification languages:

1. an intermediate language which is target machine independent and describes the operations of the high level language,
2. a machine specification language which describes the target machine to the MMCC.

The intermediate language (called the Meta Assembly Language (MAL)) is an intermediate assembly language between the high level language and the machine language. This language is machine independent but "closer" to the machine language than the high level language.

The machine specification language is used by the Target Machine Specifier to clearly define those features of his machine which are common and can be formalized. Those target machine characteristics which represent the operational idiosyncracies must be handled by specially written routines.

In the following sections, a brief overview of the MMCC is given. Emphasis will be placed on the discussion of the specification language and its use in describing mini computers. Finally, an explanation is given in how code can be generated from the description provided by the specification language.

Design Technique for the MMCC-Universal Compiler

The basic conceptual model of MMCC is shown in Fig. 1. Inputs are of two types. One is the source code input and the other, the parametric input. The source input is software for some application written in a high level language; the parametric input is the set of target machine characteristics. The object output is the corresponding assembly code for the proper target machine. In accordance with the parametric description, MMCC behaves as a compiler whose function is to translate source programs written in a high level language into the corresponding assembly code for a specific target machine.

The above model may be implemented by the universal compiler approach. This is illustrated in Fig. 2. The universal compiler accepts source programs as input and translates them into the target machine assembly language code using tables extracted from the parametric input. The overall implementation scheme of a universal compiler is depicted in Fig. 3. The two major internal components of the universal compiler are a table-driven compiler and a meta assembler.

The table-driven compiler is responsible for converting the input source language program into language dependent constructs as indicated by the formal language description. From these language constructs, the semantic processor generates corresponding code in a lower level language, the Meta Assembly Language.

The output of this table-driven compiler, the meta assembly program, is the input to the next component, the meta assembler, which generates the target machine assembly program. A compiler-compiler [1] could also be optionally employed to help automatic generation of the table-driven compiler, which is a step toward high level language independence. The above techniques are fairly well known and will not be pursued any further.

[†]Research supported by National Science Foundation Grant DCR74-21248.

Meta Assembly Language

The purpose of MAL is to obtain effective adaptation for a variety of machines. The translation from source language to the Meta Assembly Language eliminates many of the past ad hoc approaches used in the direct translation process into different machine assembly languages. Consequently, efforts required to achieve machine independence can be minimized.

The Meta Assembly Language is not a universal intermediate language. Rather, it is designed to include only the features found in various minicomputers while maintaining generality with respect to machine assembly languages. Its design has been influenced by the features found in today's most popular minicomputers.

On the Uniform Features of Minicomputers Supported by MMCC

The non-universality of the intermediate language brings up the question of what features the minicomputers should have in order to qualify as "target machines" for MMCC. In this section, the findings from a study of uniform characteristics of several of the most popular minicomputers [4,5,6] will be summarized. The purpose of this study is the following:

1. To classify common minicomputer features which the Meta Assembler will support as object or target machines.
2. To single out those idiosyncrasies which need special attention at the source or Meta Assembly language level.

A qualified target machine is one that essentially fits the mold of traditional second and third (and many of the fourth) generation minicomputer designs, i.e. serial processing of instructions that reference registers, memory and immediate operands. Capabilities of the target machine should include multiple registers; direct, indirect, indexed, self-relative addressing; and the full complement of arithmetic, logical, comparison, test and branch instructions.

Each minicomputer, in addition to the above general features, should possess certain capabilities in order to overcome restrictions imposed on it by hardware limitations. These include page addressing, implicit base addressing, relocatable addressing, etc. Such features are made transparent to the user of the machine through a powerful resident assembler which performs the necessary transformation during assembly time. MMCC ignores these peculiar features and deals only with the symbolic assembly language, generating the appropriate symbolic forms of opcode and operands which are then recognized and processed by the target machine's local assembler.

Meta Assembly Language Abstract Machine

In this section the abstract machine for the Meta Assembly Language will be described. The abstract architecture reflects the common features of those minicomputers which qualify as target machines. During code generation, the modules of the abstract machine are mapped one by one onto the minicomputer's corresponding hardware units according to the description provided by the Target Machine Specifier.

Organization

A basic block diagram of Meta Assembly Language machines is shown in Fig. 4. The program refers to two types of memories, temporary (T) and main (M) memory. Temporary memory is used to hold intermediate computational results. Operations normally take place on the variables after they are transferred to (T). During code generation, (T) will in effect be mapped onto the target machine's registers as well as on primary memory. Storage allocation (registers and primary memory) may be minimized by implementation of a suitable algorithm based on global flow analysis of the program.

Main memory (M) is used to hold program declared variables. The length of this memory is potentially unlimited and is directly addressable using the variable's symbolic representation. There is a direct mutual path between the memories and the CPU, implying that the contents of main storage are directly alterable by CPU instructions without secondary reference to temporary storage. This latter capability is included so that those minicomputers which allow direct manipulation of the information stored in their main storage may be simulated.

CPU

The central processing unit in the MAL abstract machine is capable of performing those operations common to most minicomputers. Operations are performed on the data stored in temporary or main storage. Instructions have been selected after careful consideration of the instruction set of the three most common minicomputers, e.g. HP 2100, PDP-11 and TI960A [4,5,6]. These instructions are anticipated to be general enough and representative of a large class of minicomputers. Instructions have one of the following formats:

```
a) single-operand: [<LABEL>=]<opcode><operand>

      example:  INCR  T1
                ALS   T2  /* arithmetic left
                             shift */

b) double-operand: [<LABEL>=]<opcode><operand1>,
                    <operand2>

      example:  ADI   T1,A  /* integer addition
                               A+T1→T1 */
                MOVE  T2,B  /* transfer operation
                               B→T2 */

c) triple-operand: [<LABEL>=]<opcode><operand1>.
                    <operand2>,<label>

      example:  BEQ   B1,B2,L  /* if B1 = B2 then
                                  goto L */
```

Because of space limitations a more detailed description of the language is omitted. The operations include:

1. program control
2. conditional testing
3. arithmetic operations
4. logical operations
5. data transfer operations
6. input/output handling
7. subroutine linkage and their invocations
8. optional utilization of machine independent features

Target Machine Description

The target machine description includes both information concerning the hardware capabilities and the details for interfacing with the appropriate resident programs (e.g. operating system monitors, user's defined semantic routines, etc.). Thus, the description includes both the processing units and the system configuration to which the CPU belongs. In other words, the description should be sufficient to allow the meta assembler to set up a mapping from meta assembly language to the target machine assembly language. The target machine description contains the following features:

1. Programmable hardware resources, including registers, memory, etc.
2. Relevant operational characteristics, including word-length, data format, addressing mode, etc.
3. Instruction set with appropriate classification.

The target machine description is input to the meta assembler by means of a formalism called Machine Specification Language (MSL). Describing hardware units by a description language has been used by others, among other things, for testing the logic circuit and microprogram design [2]. These languages have been used mainly to describe system components and the interconnections between them at a lower level than our intended application. On the other hand MSL is basically a declarative, non-procedural language designed to describe hardware modules at the level of an assembly language. It provides sufficient flexibility for defining the target machines belonging to a large class of architectures. The MSL syntax is simple and corresponds to the normal sequence needed to define the structure of a machine. In the following paragraphs, those features of the language pertaining to memory structure and CPU description are explained. A complete BNF syntax of the language for the definition of these two components is included in Appendix A.

I. Target Machine Memory Structure Definition

The Target Machine Specifier defines the structure of his machine's memory by providing the following information:

1. program addressable storage units and their size;
2. length (precision) of data if different from size of storage units;
3. off-set of data from storage unit boundary, if applicable.

Example: (memory structure definition for HP 2100A)

```
BSTRUCT
      WORD[0..15];
     DWORD[0..31]
ESTRUCT
  BMAP
        INT: WORD;
        INT: DWORD;
       CHAR: WORD[8..15];
    ADDRESS: WORD;
    LOGICAL: WORD
  EMAP
```

In the above example addressable storage units are 'WORD' (16 bits) and 'DWORD' (32 bits). Variables of type integer (INT) can occupy either a single word or double words, depending on precision requirements. A character occupies the right most eight bits of a word (bits 8 to 15). BSTRUCT/ESTRUCT and BMAP/EMAP separate the definition sequences into blocks with different identities, thus, make them suitable for later processing.

II. Target Machine CPU Description

The description of the CPU includes register definition, pathways between registers and primary memory, and control.

1. Register Definition: The Target Machine Specifier defines the program addressable registers of his machine by specifying their symbolic address, length and the sets to which these registers belong. The definition of registers is an important part of target machine descriptions and is closely related to the semantic description of the instruction set.

Example: (description of registers for HP 2100A)

```
BREG
     REG = ('A':16,'B':16,'E':1)
   BCLASS
        GPR = ('A','B');
         R1 = ('A');
         R2 = ('A');
         R3 = ('E')
   ECLASS
      BREL R2 = [<'A':'B'>](FREE)  EREL
      BRPATH
        GPR → R1('LD' R1,GPR);
        GPR → R2('LD' R2,GPR)
      ERPATH
EREG
```

In the above example, register set 'REG' is defined to include the symbolic name of all program addressable registers and their size. In the next part, these registers are grouped under different register sets, thus organizing the class set {GPR,R1,R2,R3}. Each of these register sets has some semantic information attached to it. Some of this information is implicit and emerges during the definition of the instruction set (see description below) while other information is explicitly attached to each register set in the BREL/EREL part of the definition. In the above example, semantic entity 'FREE' has been attached to register set R2. This attachment implies that any reference to an element of R2 (in this case only register A) will 'FREE' its corresponding register; the one indicated by the relation R2.

A register set R may carry with it the following semantic information, implicitly or explicitly.

a. Only certain operations are allowed to perform on these registers.
b. Only operands with specified precisions can occupy these registers.
c. Some arbitrary semantic information, defined by the target machine specifier, are attached to these registers.

Since the Target Machine Specifier has complete freedom in classifying the registers according to different semantic entities, he also has to provide the semantic routines. Each routine is invoked whenever a register belonging to its corresponding register set is referenced.

The Target Machine Specifier should also define the transfer paths between different register sets, if applicable. There exists a transfer path from a register set R_i to R_j ($R_i \rightarrow R_j$) if and only if there exists $r_i \in R_i$ such that $r_i \notin R_j$ and this transfer is semantically correct. In the above example BRPATH/ERPATH defines the transfer paths between GPR and each of R1 and R2. Note that 'B' $\in$ GPR but 'B' $\notin$ R1,R2. Transfer paths between R3 and other register sets are not meaningful and therefore they are not specified here. The formalism provided for register definition allows for modularized development of semantic routines with well defined interface specification with other modules of the Meta Assembler.

2. Pathways to Primary Memory: In the previous section different register sets were defined. In this part, the target machine specifier should describe the permitted transfer path between addressable storage units and registers of his machine. This is a necessity since for LOAD operation, the selected register and storage unit should match in size and type requirements.

Example: (HP 2100)

```
BMREG
   WORD(INT,LOGICAL,CHAR,ADDRESS) → GPR;
   DWORD(INT) → R2
EMREG
BPATH
   GPR: ('LD' GPR,OP)('ST' GPR,OP);
    R1: ('LD' R1,OP)('ST' R1,OP);
    R2: ('DLD' OP)('DST' OP)
EPATH
```

In the above example, any operand with word precision would be loaded into a single register ('A' or 'B') while a double precision integer would eventually occupy both registers. Note that register sets 'GPR' and 'R2' have already been declared. The BPATH/EPATH pair define the LOAD/STORE instructions necessary for transferring the operands from primary memory to the elements of the register sets previously defined. For example, operands with double word precision can be specified to occupy the double length register formed by combining A and B. This process is implicit in the semantic entity 'FREE' attached to R2.

3. Control: The flow of information (data and instruction) between various submodules of the computer system is controlled by the machine's assembly instructions. Description of the target machine's instruction set consists of two parts:

a. Definition of the target machine's instruction sequencing;
b. Definition of the target machine's addressing modes.

For each Meta Assembly Instruction, the Target Machine Specifier should define the equivalent instruction(s) for his machine. This process can be formalized using the state transition technique discussed in [3].

At each instant of instruction execution, an operand might be residing in a primary storage location, a register, a sub-field of the machine's instruction (immediate operand) or the operand might be a temporary with no storage allocated to it at all. These cases are indicated by letters 'S', 'R', 'O' and 'N', respectively. Before object code can be generated, the Meta Assemblers should be in one of the permitted states. This state corresponds to the required run time physical locations of the operands in the target machine's assembly instruction. Therefore, the Target Machine Specifier should first specify the sequence of state transitions needed to reach one of the target machine's permitted states from any initial input state. Then he should define the instruction(s) of his machine corresponding to the Meta Assembly instruction.

Example: (HP 2100A)

```
'AND' ⇒ SO → RO(LOAD R1, OP1);
        SS → RS(LOAD R1, OP1);
        SR → RR|RS(LOAD R1, OP1);
        RR → RS(STORE OP2);
        RO → NIL(TRAN R1, OP1) ('AND' OP2)
        RS → NIL(TRAN R1, OP1) ('AND' OP2)
```

In the above example, which is the definition of an 'AND' operation for HP 2100, the input states are (SO), (SS), (SR), (RR), (RO) and (RS) while the permitted states are (RO) and (RS) only. Note for example that to reach (RS) from (SR) the sequence of transitions would be SR → RR → RS or SR → RS (in the case where loading of the first operand might cause storing the second one back to main memory). Function (TRAN R1, OP1) would cause the first operand (OP1) to be loaded in register of class R1 (in this case 'A').

A similar formalism has been developed for defining addressing modes for the target machine. This uniform formalism for describing both the instruction sequencing and the address mode makes the code generation process simpler since the same module can be used for both address translation and code generation.

Example: (TI960A - definition for indexing)

```
$X ⇒ SS → SR(LOAD R1, OP2);
     RS → SS(STORE OP1);
     RR → SR(STORE OP1);
     SR → NIL(CONCAT OP1, OP2, ',I')
```

Indexed operands in MAL are represented by $X(A, IND) where A is the array name and IND is the indexing operand. This operand is translated into 'A, r, I' for the TI machine, where r is a register name and I is the index designator. A similar state transition formalism is used to define direct, indirect and immediate addressing.

Automatic Code Generation from Target Machine Description

The Meta Assembler has an Automatic Code Generator (ACG) built into it. This module accepts the description of a particular machine and translates the Meta Assembly Language instructions, line by line, into a sequence of target machine's symbolic assembly language instructions. The following steps summarize the process of object code generation.

I. Each MAL instruction is scanned and its label (if any), opcode and operand(s) are extracted.

II. The address mode of each operand is determined. Then using the description of the target machine's address modes, this operand is modified (symbolically) to conform to the target machine's conventions. Moreover, additional intermediate code might be generated in the cases where mere symbolic transformation is not sufficient. In any case, any required operation is clearly indicated by the definition provided by the target machine specifier with the aid of the formalism discussed before.

III. The state of the operand is determined in this stage. To do this, the symbol table and the register status table are searched and, according to the different modes, one of the states S, R, O or N is assigned to the operand (as discussed previously).

IV. Once the input state is determined, the MAL instruction opcode is used to initiate the particular state transition process. Normally, the input state does not match the target machine's permitted state. Therefore the code generation is further broken down into two steps: 1) state transition; to reach the target state from the input state (which might result in generation of load or store instructions) and 2) the generation of the target code to perform the operations specified by the Meta Assembly instruction's opcode.

These four steps are repeated for each new line of a Meta Assembly instruction. A block diagram for the Meta Assembler's different phases of operations is shown in Fig. 5.

Conclusion

In this paper recent studies in formalizing the code generation process have been reported. Formalization is partially achieved due to our ability to describe the program addressable modules to MMCC in the notation of the Machine Specification Language. Efforts at describing input, output routines and other idiosyncrasies such as interrupts is still under way. Also the problem of subroutine calls and their invocations has not been dealt with yet. Since these aspects of compilation are highly machine dependent it is anticipated that a description would only consist of providing high level primitives in Meta Assembly Language and requiring the target machine specifier to provide the code sequences in the formalism already discussed.

Using the MSL, we have defined the CPU and the memory structure of three machines which enjoy uniform characteristics. These machines are the TI960A, HP 2100 and PDP-11/45. A description of the TI960A machine [4] is provided in Appendix B for reference.

Acknowledgment

The authors would like to thank Mr. Alex Conn for his careful reading of the draft of this paper and making useful suggestions for its improvement.

References

[1] R.A. Brooken et al., "The Compiler-Compiler," Annual Review in Automatic Programming, Vol.3, (1963).

[2] Special Issue of COMPUTER on Hardware Description Language (Dec. 1974).

[3] P.L. Miller, Automatic Creation of a Code Generator from a Machine Description, TR-85, Project MAC, MIT (1971).

[4] Programmers Reference Manual for the Model 960A Computers, Texas Instrument Corporation, Houston, Texas.

[5] HP 2100 Reference Manual, Hewlett Packard Company, Cupertino, California.

[6] PDP 11 Processor Handbook, Digital Equipment Corporation.

Appendix A

Syntax description for Machine Specification Language (MSL)

```
 <COMPUTER> ::= BEGIN<MEMORY><CPU>END
   <MEMORY> ::= BMEM<MM>EMEM
       <MM> ::= <ACSU><MAPPING>
     <ACSU> ::= BSTRUCT<STI>ESTRUCT
      <STI> ::= <DT>|<STI>';'<DT>
       <DT> ::= <STRGE>'['<LOWER>'..'<UPPER>']'
    <LOWER> ::= $INTEGER
    <UPPER> ::= $INTEGER
    <STRGE> ::= WORD|DWORD|BYTE|HWORD
  <MAPPING> ::= BMAP<MAPS>EMAP
     <MAPS> ::= <MAP>|<MAPS>';'<MAP>
      <MAP> ::= <DATA-TYPE>':'<STRGE>
                |<DATA-TYPE>':'<STRGE>'['<LOWER>
                '..'<UPPER>']'
      <CPU> ::= BCPU<REGISTERS><MEMORY-REG>
                <CONTROL>ECPU
<REGISTERS> ::= BREG<NAMES><CLSFS><RELATIONS>
                <CLSPATHS>EREG
    <NAMES> ::= 'REG =' '(' <ARGS> ')'
     <ARGS> ::= <ARG>|<ARGS>','<ARG>
```

```
        <ARG> ::= $RN ':' <LENGTH>
     <LENGTH> ::= $INTEGER
      <CLSFS> ::= BCLASS<CLASS>ECLASS
      <CLASS> ::= <CL>|<CLASS>';'<CL>
         <CL> ::= $CLSNM '=' '(' <RGNM> ')'
       <RGNM> ::= $RN|<RGNM>','$RN
  <RELATIONS> ::= BREL<RLTN>EREL|Λ
       <RLTN> ::= <RLT>|<RLTN>';'<RLT>
        <RLT> ::= $CLSNM '=' '[' <CPLS> ']'
                  '(' <OPSEQ> ')'
      <OPSEQ> ::= $OPTNS|$OPTNS','<OPSEQ>
       <CPLS> ::= <CPL>|<CPLS>','<CPL>
        <CPL> ::= '<' $RN '>'|'<' $RN<RNSEQ> '>'
      <RNSEQ> ::= ':'$RN|<RNSEQ>':'$RN
   <CLSPATHS> ::= BRPATH<CPATHS>ERPATH|Λ
     <CPATHS> ::= <CPATH>|<CPATHS>';'<CPATH>
      <CPATH> ::= $CLSNM '→' $CLSNM<CSEQ>
 <MEMORY-REG> ::= <M-REG><MPATHS>
      <M-REG> ::= BMREG<SQNC>EMREG
       <SQNC> ::= <SEQ>|<SQNC>';'<SEQ>
        <SEQ> ::= <STRGE> '→' $CLSNM|<STRGE>
                  '(' <DATA> ')' '→' $CLSNM
       <DATA> ::= <DATA-TYPE>|<DATA>','<DATA-TYPE>
  <DATA-TYPE> ::= LOGICAL|BOOL|INT|ADDRESS|CHAR
                  |REAL
     <MPATHS> ::= BPATH<PATHS>EPATH|Λ
      <PATHS> ::= <PATH>|<PATHS>';'<PATH>
       <PATH> ::= $CLSNM ':' <CSEQ>
    <CONTROL> ::= BINST<INSTR>EINST
      <INSTR> ::= <OPCODE><OPERAND>
     <OPCODE> ::= BCODE<MAC>ECODE
        <MAC> ::= <MAL>|<MAC> next<MAL>
        <MAL> ::= <OPCOD> '⇒' <SEQUENCING>
      <OPCOD> ::= CL|CM|···|$M|$D|$R|$O|$H|$C|$X|$I
 <SEQUENCING> ::= <STS>';'<CGEN>|<CGEN>
        <STS> ::= <TREM>|<STS>';'<TRFM>
       <TRFM> ::= <PS> '→' <NS> '(' $OPTNS ')'
                  |<PS> '→' <NS> '|' <ALT>
                  '(' $OPTNS ')'
         <PS> ::= R|S|SO|RO|NS|NR|RR|RS|SR|SS
                  |$TARGET
         <NS> ::= R|S|SO|RO|RR|RS|SR|SS
        <ALT> ::= <NS>
       <CGEN> ::= <GEN>|<CGEN>';'<GEN>
        <GEN> ::= <PS> '→' NIL<CSEQ>
       <CSEQ> ::= '(' $OPTNS ')'|<CSEQ>'('$OPTNS')'
    <OPERAND> ::= BOP<MAC>EOP
```

Description of the terminals:

$RN	is register's symbolic name
$CLSNM	is register set name
$OPTNS	is Target Machine Specifier's defined semantic routine
$TARGET	is symbolic name for Target Machine Specifier's defined state

Appendix B

Partial description of TI960 mini computer using MSL.

```
BEGIN
  BMEM
    BSTRUCT
        WORD[0..15];
      DWORD[0..31]
    ESTRUCT
    BMAP
             INT: WORD;
             INT: DWORD;
            CHAR: WORD[8..15];
         ADDRESS: WORD;
         LOGICAL: WORD;
            BOOL: WORD
    EMAP
  EMEM
  BCPU
    BREG
             REG = (0:16,1:16,2:16,3:16,4:16,5:16,
                    6:16,7:16)
      BCLASS
             GPR = ('0','1','2',...,'6');
              R1 = ('0','1','2',...,'6');
              R2 = ('0','1','2',...,'6');
              R3 = ('0','1','2',...,'6')
      ECLASS
      BREL
              R1 = [<0:1>,<1:2>,<2:3>,<3:4>,<4:5>,
                    <5:6>,<6:7>](FREE)*;
              R2 = [<0:1>,<1:2>,<2:3>,<3:4>,<4:5>,
                    <5:6>,<6:7>](NORM)*;
              R3 = [<0:1>,<1:2>,<2:3>,<3:4>,<4:5>,
                    <5:6>,<6:7>](SWITCH)*
      EREL
    EREG
    BMREG
             WORD(INT,BOOL,LOGICAL,CHAR,ADDRESS)
                → GPR
    EMREG
    BPATH
        'GPR': ('L' GPR,OP)('ST' GPR,OP);
           R1: ('L' R1,OP)('ST' R1,OP);
           R2: ('L' R2,OP)('ST' R2,OP);
           R3: ('L' R3,OP)('ST' R3,OP)
    EPATH
    BINST
      BCODE
         'INCR' ⇒ S → R(LOAD OP1);
                  R → NIL('AA' OP1, 1) next
         'DECR' ⇒ S → R(LOAD OP1);
                  R → NIL('SA' OP1, 1) next
           'CL' ⇒ S → R(LOAD OP1);
                  R → NIL('LA' OP1, 0) next
           'CM' ⇒ S → R(LOAD OP1);
                  R → NIL('XORA' OP1, X'FFFF')
                      next
          'SET' ⇒ S → R(LOAD OP1);
                  R → NIL('LA' OP1, X'FFFF')
                      next
           'JP' ⇒   → NIL('B' OP1, 0) next
```

*FREE, NORM and SWITCH are semantic routines related to AND, DIVIDE, MULTIPLY and other operations whose definitions have been omitted for brevity.

```
             'ADI' ⇒ SS → RS(LOAD OP1);
                     SR → SS(STORE OP2);
                     RR → RS(STORE OP2);
                     RS → NIL('A' OP1, OP2);
                     SO → NIL('AMI' OP1, OP2);
                     RO → NIL('AA' OP1, OP2) next
          :
        ECODE
        BOP
           $M ⇒     → NIL next
           $D ⇒     → NIL next
           $R ⇒     → NIL (ERROR) next
           $O ⇒     → NIL (ERROR) next
           $H ⇒     → NIL (CONCAT 'X', OP1) next
           $C ⇒     → NIL (CONCAT 'C', OP1) next
           $X ⇒ SS  → SR(LOAD OP2);
                RS  → SS(STORE OP1);
                RR  → SR(STORE OP1);
                SR  → NIL (CONCAT† OP1, OP2, ',I')
                      next
           $I ⇒  R  → S(STORE OP1);
                 S  → NIL (CONCAT '*', OP1)
        EOP
      EINST
    ECPU
  END
```

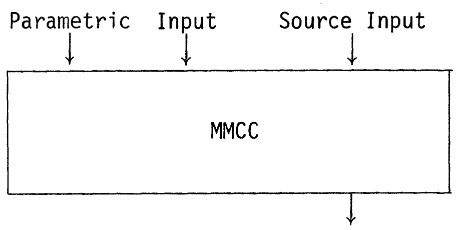


Fig. 1 Basic Model for MMCC

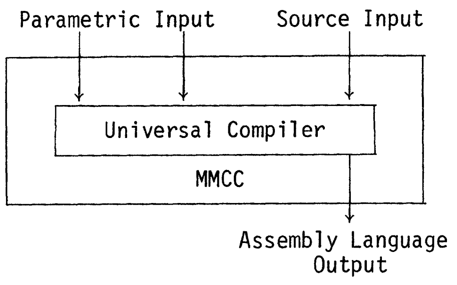


Fig. 2 Model for MMCC, Employing a Universal Compiler

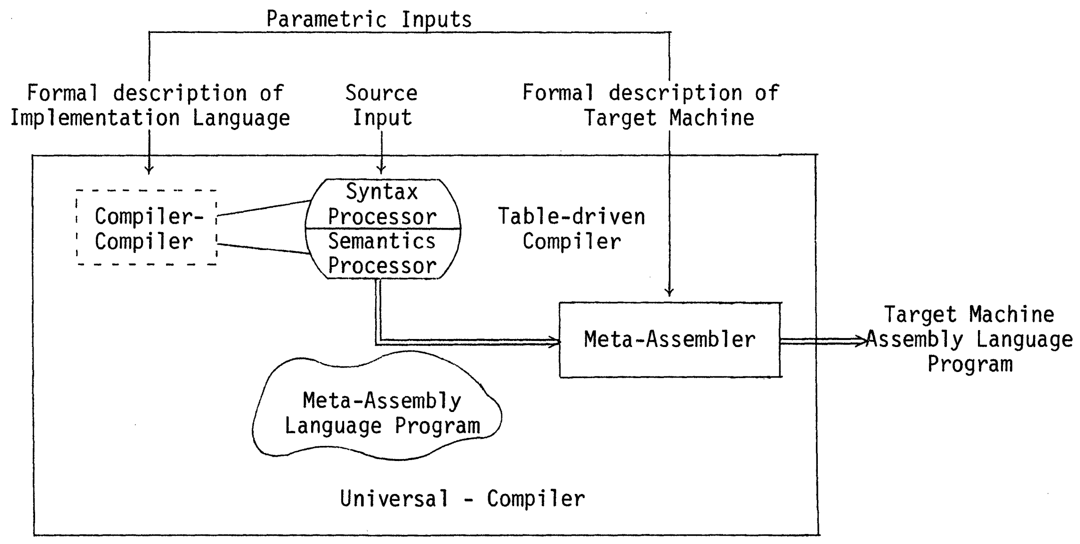

Parametric Inputs
Formal description of Implementation Language
Source Input
Formal description of Target Machine
Compiler-Compiler
Syntax Processor
Semantics Processor
Table-driven Compiler
Meta-Assembler
Target Machine Assembly Language Program
Meta-Assembly Language Program
Universal - Compiler

Fig. 3 Implementation Scheme of a Universal Compiler

†CONCAT is a routine which concatenates its arguments and returns the result as the symbolic address of the operand.

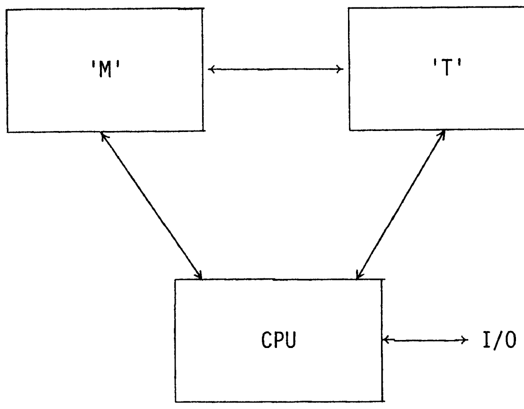


Fig. 4 Simple Abstract Machine of the Meta Assembly Language

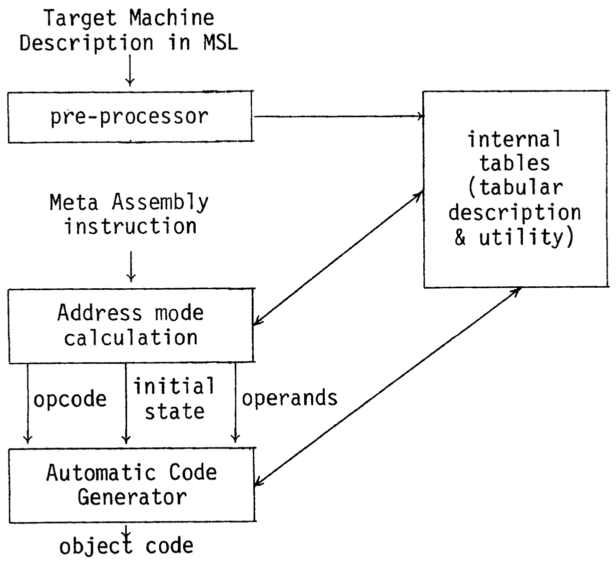

Target Machine Description in MSL
pre-processor
internal tables (tabular description & utility)
Meta Assembly instruction
Address mode calculation
opcode
initial state
operands
Automatic Code Generator
object code

Fig. 5 Implementation Scheme for the Meta Assembler

EVALUATION OF A POLISH FORM EXPRESSION ON A FI-FO QUEUE: A NEW APPROACH TOWARD THE REALIZATION OF A HIGH LEVEL "PIPE-LINE" COMPUTER

Gérard G.BAILLE and Jean P.SCHOELLKOPF
Computer Architecture Group
ENSIMAG-BP 53 38041 GRENOBLE(FRANCE)

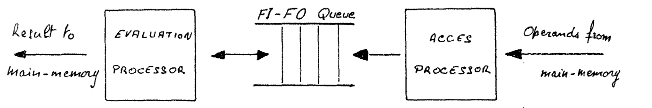
Summary

Let the polish form expression where O_i^j represents the jth operand of the group number i and x_i^k the kth operator of the same group:

$$\underbrace{O_1^1 \ldots O_1^{\alpha_1}\ x_1^1 \ldots x_1^{\beta_1}}_{\text{group } 1}\ \ldots\ \underbrace{O_p^1 \ldots O_p^{\alpha_p}\ x_p^1 \ldots x_p^{\beta_p}}_{\text{group } p}$$

In classical machines using such a post fixed language, the expressions are valued on a push-down stack. This envolves a monoprocessor architecture for which access to the operands and computation of operators cannot run at the same time.

Parallel processing is made possible if the push-down stack is replaced by a FI-FO Queue in the following manner:



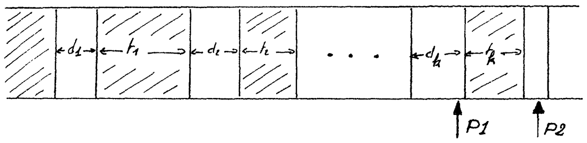
It is necessary to define two pointers P1 and P2 which give respectively the location of the first and the second operand on the queue.

A queue must be cleared out by the bottom:therefore the result of the operator x_i^j is stored in position given by P2(second operand) and a new free location is created in position given by P1(first operand).

Assume that a finite sequence

$$(d_1,t_1)\ ,\ (d_2,t_2),\ \ldots,\ (d_k,t_k)$$

defines the state of the queue during evaluation process $(t_1,t_2,\ldots,t_k)$ gives the length of the free location sequences, $(d_1,d_2,\ldots,d_k)$ gives the length of the sequences of location which hold the operands waiting for a further operation.



In order to have P1 pointing to a not free location, before storing of result of operator x_i^j, let us decrement P1 by one. In the same time, let us modify the last element of the sequence (d_k,t_k):

$$d_k := d_k - 1$$
$$t_k := t_k - 1$$

Either d_k becomes zero, in which case P1 must be decremented again by t_{k-1}. Further the last element (d_k,t_k) disappears and the state of the queue is modified

$$t_{k-1} := t_{k-1} + t_k \quad ; \quad k := k-1 \ ;$$

If a $x_i^{\beta_i}$ is the last executed operator, the next to be executed is operator x_{i+1}^1. P2 must give the position of the second operand which is $O_{i+1}^{\alpha_{i+1}}$, therefore it must be incremented by α_{i+1}. P1 must receive a value equal to the value of P2 minus 1.

In the same time, a new element is created for the sequence (d_k,t_k) in order to have a new state of the queue:

$$k := k+1 \ ;\ d_k := \alpha_{i+1} \ ;\ t_k := 0$$

Therefore we must generate three kinds of orders for the evaluation processor:

- execute a dyadic operator,
- increment P2 by a value n and initialize P1(INCR)
- decrement P1 by a value n (DECR).

If there are N operands divided in p groups in the expression to be evaluated, we have:

N orders for Access Processor,
N-1 orders for Evaluation Processor,
p extra orders INCR.

The maximum of DECR orders will be given by:

N-1-p if $p+1 \leq N \leq 2p$
p-1 if $N \geq 2p$

The analysis of the expressions is performed by a third processor called Control Processor which fetches instructions from main memory, and generates orders for the other two processors.

The above evaluation mechanism permits the realization of a new pipe-line architecture.

A COMPARISON OF APPROXIMATE SCHEDULING ALGORITHMS

Klaus Ecker
Gesellschaft für Mathematik
und Datenverarbeitung
5205 St. Augustin, Germany West

Summary

We consider a simplified version of the scheduling problem where a task-graph is given, and the tasks have to be performed without pre-emptions on an array of m identical processors. Each task has to be assigned to one of the processors, but at any time not more than one task to each processor. We assume that all task execution times are equal. The problem is to construct algorithms which produce time-minimal schedules. Several efficient algorithms producing time-minimal schedules under special conditions are known, but in most cases little is known about the quality of the results of these algorithms in general. In this paper the schedules obtained by these algorithms, applied under general conditions, are compared with the best possible schedules.

If A is a scheduling algorithm, $t_{A,m}(G)$ denotes the length (in time) of the schedule generated by A for the task-graph G when m processors are available. $t_{O,m}(G)$ denotes the length of a time-optimal schedule. The ratio $r_m(A,G) = t_{A,m}(G)/t_{O,m}(G)$ is a measure for the applicability of A with respect to G. The value $R_m(A,n) = \max\{r_m(A,G) \mid G \text{ contains at most } n \text{ tasks}\}$ is suitable for a comparison of A with time-optimal algorithms [4].

Turning to special algorithms, the measure r_m for the level-algorithm defined by HU in [3] can become arbitrarily close to $(2m-1)/m$ which is the worst possible ratio for list scheduling [1], [2]. The same is true for an algorithm which assigns a higher priority to tasks with a higher number of immediate successors, and for combinations of this algorithm with HU's level-algorithm. Consequently, due to the measure R_m those algorithms cannot be considered as algorithms generating good schedules in general.

Therefore we ask for efficient algorithms which are better in the above sense. An algorithm A is called <u>k-depth-bounded</u> iff there is a $k \in \mathbb{N}$ ($\mathbb{N}=\{1,2,\dots\}$) such that at every point of time A considers only the subgraph defined by all chains of length k starting from the actual starting nodes. A is called <u>k-optimal</u> iff A produces time-minimal schedules for all graphs of height $\leq k$.

<u>Theorem 1</u>. Let A_k ($k \geq 2$) be k-optimal and k-depth-bounded algorithms. Then
$\forall m \in \mathbb{N}, \forall \varepsilon \in (0,1], \exists n \in \mathbb{N}: R_m(A_k,n) \in [\frac{2m-1}{m}-\varepsilon, \frac{2m-1}{m})$.

In the sequel we consider the case $k=2$. It is possible to construct A_2 in such a way that its its time complexity is $O(n^{m-1})$. Using A_2 we define a new algorithm H_2:

```
(1) $G' := G$.
(2) Let $V_s$ be the set of starting nodes of $G'$.
    If $V_s = \emptyset$, go to (5).
(3) If $|V_s| \leq m$, assign the elements of $V_s$ to
    the processors. Set $G' := G' - V_s$. Go to (2).
(4) Let $i \in \mathbb{N}$ be the maximal integer with
    $|\{a \mid a \in V_s, \lambda(a) \geq i\}| \geq m$ ($\lambda(a)$ = level of
    task a). Let $\tilde{G}$ be the subgraph of $G'$ which
    consists exactly of all chains of $G'$ star-
    ting in $\{a \mid a \in V_s, \lambda(a) \geq i\}$. Determine $m$
    tasks $a_1, \dots, a_m$ which are to be assigned to
    the $m$ processors by applying $A_2$ to $\tilde{G}$.
    Set $G' := G' - \{a_1, \dots, a_m\}$. Go to (2).
(5) Stop.
```

The time complexity of H_2 is $O(n^m)$.

<u>Theorem 2</u>. $\forall m \in \mathbb{N}, \forall G: r_m(H_2,G) \leq 3/2$.

<u>Remark</u>. Algorithms $H_3, H_4, \dots$ can be defined in an analogous way, however their value R_m is not better than that of H_2. It can be shown that for all $k \geq 2$ $r_m(H_k,G)$ can become arbitrarily close to 3/2 so that in this sense $H_3, H_4, \dots$ are no essential improvements of H_2.

References

[1] N.F. Chen, and C.L. Liu, "On a Class of Scheduling Algorithms for Multiprocessing Computing Systems", Proceedings of the Sagamore Computer Conference on Parallel Processing 1974, Lecture Notes in Computer Science 24, Springer-Verlag, (1975), pp. 1-16

[2] E.G. Coffman Jr., and P.J.Denning, <u>Operating Systems Theory</u>, Prentice Hall, (1973), 331 pp.

[3] T.C. Hu, "Parallel Sequencing and Assembly Line Problems", <u>Opns. Res.</u> 9 (1961), pp. 841-848

[4] D.S. Johnson, "Approximation Algorithms for Combinatorial Problems", Proceedings of the 5th Annual Symposium on Theory of Complexity, Austin, Texas, 1973, pp. 38-49

A LINEAR SCHEDULING ALGORITHM FOR A FOREST ON A MULTIPROCESSOR SYSTEM

D. Hennings, S. Schindler, M. Steinacker
Fachbereich 20 (Kybernetik)
Technische Universität Berlin

Summary

The problem is studied of scheduling N tasks - the operational precedence structure of which is represented as a finite, directed, weighted forest G (paths being directed from leaves to roots) - on a multiprocessor system consisting of M identical processors. The weight W_i of each node i, $1 \leq i \leq N$, is regarded as the processing time of the task represented by node i. We are concerned especially with the problem to construct time-optimal schedules of low complexity for complete processing of G by the M processors.

For the case $W_i = 1$ for $1 \leq i \leq N$ Hu's algorithm (see [1]) is a well known solution to this problem; a proof of correctness is given in [4], e.g. If we give up the assumption of unit processing times the problem becomes very difficult: it is known to be "polynomial complete" (see [5]) unless preemptions are allowed.

For arbitrary processing times W_i and an arbitrary finite number of preemptions the problem is tractable again. Muntz/Coffman (see [2]) obtained a polynomial bounded scheduling algorithm. One of the authors (see [3,4]) gave a characterization of the set of all time-optimal schedules in this case, or more explicitly, of the set of all "time-optimal" scheduling algorithms.

Starting from that characterization of the set of all "time-optimal" scheduling algorithms for this problem in [7] a scheduling algorithm of low complexity is isolated: the number of steps of this algorithm depends linearly on N, except for an initial sorting of G's tasks (performable by standard sorting algorithms in $O(N \cdot \mathrm{ld}\, N)$ steps at most, see [6]). This result seems to establish the lowest upper bound for the complexity (in N) of a scheduling algorithm under the above assumptions known until now.

References

[1] T. C. Hu: Parallel Sequencing and Assembly Line Problems, Operations Research 9, No. 6, 1961

[2] R. R. Muntz, E. G. Coffman, Jr.: Preemptive Scheduling for Realtime Tasks on Multiprocessor Systems, JACM, April 1970

[3] S. Schindler: On Optimal Schedules for Multiprocessor Systems, Princeton Conference on Information Sciences and Systems 1970

[4] S. Schindler: Quantitative Aspects of Optimal Schedules for Multiprocessor Systems, Technical Report 73-10, Juli 1973, Fachbereich 20, Technische Universität Berlin

[5] J. D. Ullman: Polynomial Complete Scheduling Problems, Technical Report 9, March 1973, University of California at Berkeley

[6] E. M. Reingold: Establishing Lower Bounds on Algorithms - A Survey, Spring Joint Computer Conference, 1972

[7] D. Hennings, S. Schindler, M. Steinacker: The Complexity of Preemptive Scheduling Algorithms for Multiprocessor Systems, Technical Report 74-20, December 1974, Fachbereich 20, Technische Universität Berlin

A GRAPH-THEORETIC CHARACTERIZATION OF A CLASS OF SYNCHRONIZING PRIMITIVES

P. B. Henderson and Y. Zalcstein
Department of Computer Science
SUNY at Stony Brook
Stony Brook, N.Y. 11794

Summary

Vantilborgh and van Lamsweerde [7] have introduced an extension of Dijkstra's PV system of primitives (dubbed PV_{chunk} or PV_c in [5]) allowing the semaphore to be updated by "chunks" that are arbitrary positive integers. The authors of [7] regarded this system merely as a programming convenience, believing that it is no more powerful than PV. The work of Lipton and his collaborators [5,6], however, has since established that there are simple synchronization problems that can only be solved by PV in rather pathological ways.

In this paper, a synchronization problem is viewed as a system of processes $\mathcal{P}$. A sequence $p_1p_2\ldots p_k$ of processes in $\mathcal{P}$ will be termed a computation provided that each p_i is not "blocked" following $p_1p_2\ldots p_{i-1}$. Many of the process synchronization problems studied in the literature may be expressed by a conjunction of finitely many conditions of the form "process p_ℓ blocks process p_m." Examples are: 1) the "reader-writer problem" [1], in which "a writer (process) blocks all readers (processes) and all other writers," and "each reader blocks all writers" (note that a reader cannot block another reader), and 2) the "five dining philosophers problem" [2], where each of the five processes blocks exactly two other distinct processes and in turn is blocked only by these two processes.

Utilizing the "blocking" relationship between processes, a synchronization problem may be represented by a directed graph $G = (N,E)$, where node set N corresponds with the set of processes, and E corresponds to the relation "blocks" on the processes (i.e. there is an edge (i,j) in E if and only if process p_i blocks process p_j). Precisely, a graph G is defined by a system of processes $\mathcal{P}$ provided that there is a one-to-one correspondence $\phi:\mathcal{P}\rightarrow N$, so that $p_1\ldots p_k$ is a computation iff $(\phi(p_i), \phi(p_j)) \notin E$, for all $1 \leq i < j \leq k$. In this paper we characterize the class of directed graphs that are definable by PV_c systems (PV_c definable), a problem left open in [5,6].

Let $\mathcal{P}$ be a system of processes and without loss of generality assume there is a single semaphore S [4]. $\mathcal{P}$ is a PV_c system provided that every instruction is either a $P(S|n)$ or $V(S|n)$, $n \geq 1$. $P(S|n)$ is an indivisible instruction such that $S \leftarrow S - n$ is executed only when $s \geq n$, otherwise control is interrupted until $s \geq n$. $V(S|n)$ is an indivisible instruction, $S \leftarrow S + n$. Since only $P(\cdot|\cdot)$ instructions can either block or be blocked it follows that if graph G is defined by PV_c system $\mathcal{P}$, then $\mathcal{P}$ consists only of $P(\cdot|\cdot)$ instructions.

Let $\mathcal{P}$ be a PV_c system of processes with semaphore S, whose initial value is t. It follows from the preceding discussion that a graph G is PV_c-definable (with respect to $\mathcal{P}$) iff there is a positive integer t such that the nodes of G can be labelled by positive integers $\leq t$, so that given node set F, the induced subgraph $\langle F\rangle$ is totally disconnected [3] if and only if $\Sigma_{z\varepsilon F}\ell(z) \leq t$, where $\ell(z)$ is the label associated with node z. That is, if $z = \phi(p_i) = \phi(P(S|n)$, then $n = \ell(z)$.

Theorem. A connected graph $G = (N,E)$ is PV_c-definable iff either G is empty or N can be partitioned into disjoint sets $C_1, C_2,\ldots,C_k,D_1,D_2,\ldots,D_{k-1}$ for $k \geq 1$, where each $\langle C_i\rangle$ $i = 1,2,\ldots,k-1$ is a nonempty clique, $\langle C_k\rangle$ is a possibly empty clique and each $\langle D_i\rangle$ $i = 1,\ldots,k-1$ is a possibly empty disconnected subgraph so that every node in C_i $i = 1,2,\ldots,k-1$ is adjacent to every node in C_{j+1} and D_j, for all $i \leq j \leq k-1$.

Corollary. Any PV_c-definable graph is an interval graph.

This statement may be interpreted as a hierarchical structure of asynchronous processes where each process in C_i can block (1) any process in D_j, $j \geq i$ and (2) any process in C_j, for all $1 \leq j \leq k$, while the processes in $\bigcup_{i=1}^{k} D_i$ cannot block each other. One can think of this situation as a "reader-writer" problem where there is a hierarchy of writers (the C_i's) and a hierarchy of readers (the D_j's). In the case $k = 2$ with C_2 empty, the normal form represents the classical reader-writer problem [1]. A complete version of this paper appears in [4].

References

[1] P. J. Courtois, F. Heymans and D. L. Parnas, "Concurrent Control with 'readers' and 'writers'", CACM (Oct.,1971) pp. 667-668.

[2] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes", in Operating Systems Techniques, C.A.R. Hoare and R.H. Perrott, ed., Academic-Press (1972), pp. 72-93.

[3] F. Harary, Graph Theory, Addison-Wesley, (1969).

[4] P.B. Henderson and Y. Zalcstein, A Graph-Theoretic Characterization of a Class of Synchronization Primitives, Computer Science, SUNY at Stony Brook, N.Y., TR #41 (March,1975) 30 pp.

[5] R.J. Lipton, On Synchronization Primitive Systems, Technical Report No. 22, Computer Science Dept., Yale University, (1973).

[6] R.J. Lipton, L. Snyder and Y. Zalcstein, "A Comparative Study of Models of Parallel Computation", 15th Annual IEEE Sym. of Switching and Automata Theory (1974), pp. 145-155.

[7] H. Vantilborgh and A. van Lamsweerde, "On an Extension of Dijkstra's Semaphore Primitives", Information Processing Letters, (Jan., 1972) pp. 181-186.

AN IMAGE ARRAY PROCESSOR FOR THE INVESTIGATION OF ARCHITECTURES AND ALGORITHMS[a]

R. M. Brown and P. L. Neely
Computer Sciences Corporation
Huntsville, Alabama 35802

Abstract -- Digital processing of image data has certain features which can be exploited in the design of processing hardware. Reconfigurability is required to implement a range of algorithms and processing tasks. However, a completely general purpose processor is not required, or even desirable, since it sacrifices processing speed and compromises data management. This paper discusses the design of a high-speed (200 nanosecond/picture element) image oriented processor with parallel architecture. Specific attention is focused on a processing element and data management scheme, optimized for vector operations, which are currently being developed for a prototype demonstration.

Introduction

Within the Data Systems Laboratory of NASA's Marshall Space Flight Center, there is an active project to study specialized architectural concepts and current technology suitable for the development of an Image Array Processor (IAP). The objectives of this project are:

1. To investigate the technology necessary to achieve pixel processing times (PPT) in the neighborhood of 100-200 ns in order to handle image data rates projected over the next decade.
2. To configure the IAP as a testbed with the ability to examine different processor architectures which take advantage of the unique characteristics of image processing.
3. To provide high speed image processing capability for users developing processing techniques and to allow their interaction with future hardware research.
4. To retain a modular configuration capable of incorporating advances in technology.

Although widespread use has been made of general purpose digital computers in image processing applications, particularly algorithm development, image data has several features which point to a more specialized form of processing [1,2]. Image data arrays can be characterized by two salient features:

1. Large size: e.g., 0.25 megapixels in a 512x512 video frame, 40 megapixels in a single channel 6300x6300 EOS frame.
2. High on-line processing rate: e.g., 100 ns/pixel PPT for NTSC video data, 400 ns/pixel PPT for EOS data [3].

The processing of such image data contains an inherent parallelism at two levels:

3. Within a specific algorithm, the same basic operations are performed on large data sets.
4. Between different algorithms, there are many common algebraic operations.

[a] This work supported by NASA-Marshall Space Flight Center under Contract NAS8-21805.

The above four features suggest that (i) Memory hierarchy and data management form a distinct and very important problem which may be separated from the processor design; (ii) Very high speed hardware processing units will be necessary; (iii) A parallel architecture should be employed for the IAP; and (iv) A trade-off can be made between optimized processing elements for selected functions and a reconfigurable hardware structure.

Candidate Architectures

There is a spectrum of parallel architectures which might be considered for use in the IAP design. The implications of these different architectures can best be examined by considering them in the context of a specific image processing task. It should be emphasized, however, that the selection of a particular task should not be construed to mean that the IAP is intended for any specific problems; quite to the contrary, the purpose of the IAP is to execute a wide variety of algorithms. With this view in mind, consider the vector dot product operation in the following analysis.

This operation is performed frequently in image analysis and is a good illustration of item (4) above. It forms, for example, the kernel of the Maximum Likelihood Classification Algorithm for multispectral data. It also is the basic operation of the Cubic Interpolation Algorithm used in the resampling process needed for picture registration [4]. In the latter application, which will be used as a benchmark, five vector dot products between four-component vectors are required to interpolate one pixel, a total of 20 multiplies and 15 adds.

The computation time needed to perform the benchmark on a GP computer ranges from approximately 35 μsec on a large scale machine such as the UNIVAC 1108 to 110 μsec on a minicomputer such as the PDP 11/45. Therefore, in order to achieve a PPT of 100-200 ns, a speed-up factor of about 350-1000 is needed.

At one end of the architecture spectrum, this speed problem might be solved through a general parallel processor array patterned after existing configurations [5,6,7,8]. The elements in this array could conveniently be bipolar microprocessors (μP's) such as MMI 6701 or Intel 3000 series. These devices use microcode to perform a multiply operation in 2.5-5.0 μsec, a time comparable with the PDP 11/45.

The benchmark PPT for one such μP is 50-100 μsec and thus a speed-up factor of about 500 is needed; that is, the array must contain between 250-1000 elements. However the problems of control and bussing (among other technical difficulties) associated with such a large array most likely prohibits this configuration of μP's despite its inherent advantages of reconfigurability and construction from low cost elements.

At the other end of the spectrum, cubic interpolation could be implemented through a special purpose pipeline processor [9]. Using ECL 10,000 logic, such as in the General Dynamics' High Speed Parallel Digital Processor [10], a PPT of 30-50 ns could be achieved. This approach, however, sacrifices reconfigurability since the algorithm is locked into the processor architecture.

An alternative approach is to divide the speed-up into two stages. A factor of 25-100 improvement can be provided through a programmable processor whose design is optimized to complete vector operations in 200 ns, a time requirement intermediate between the two other approaches. A parallel structure is used within this Vector Processing Element (VPE) in order to obtain these short execution times. One microprogrammed VPE can, therefore, execute the benchmark in approximately 1 μsec. The further speed-up needed can be achieved by configuring several (5-10) VPE's in parallel. Algorithm reconfigurability (for vector oriented data) is retained through such a VPE while the array size necessary to achieve the required PPT is small. In essence, the optimized design approach takes a basic high-speed programmable processor and adds to its list of op codes a specific function (or functions), characteristic of image processing requirements, through hardware and internal bus design.

This approach may be generalized to include a range of Optimized Processing Elements (OPE's) such as:

1. An Associative Processing Element with associative cache memory for correlation operations in pattern recognition, etc.
2. A Sorting and Counting Element for data organization and histogram construction.
3. A Generalized Function Element for ultra high speed function generation through table look-up and Taylor Series expansion.

The concepts of an OPE are similar to those of recently developed programmable signal processors and modular processors 2,11,12,13,14 . However, the objective for the IAP to be a flexible modular test bed configured as an array imposes certain requirements that the array elements should possess:

1. An appropriate I/O capability in order to allow the necessary array interconnect structure.
2. Control schemes/program stores that will allow the appropriate program partitioning and overall master control as a total unified testbed array.
3. Ultra high speed image processing arithmetic capability in order to limit array size and costs.

While it may be possible to utilize the signal or modular processors, or modifications of them, for some of the OPE's, in order to clarify the architectural features that the above requirements imply, a particular OPE, a VPE, has been studied in some detail as will be discussed in the following section.

Array Configuration

Three principal factors affect the IAP architectural design:

1. Sharing of costly resources.
2. IAP modularity.
3. IAP reconfigurability.

Computing elements are configured in an array not only to increase computing power but also to share computing resources. Without resource sharing, a collection of completely independent computers could achieve an overall high throughput by processing different segments of the data, e.g., distinct image frames. However, the cost of such a system rapidly becomes prohibitive since each machine requires its own memory, control structure, arithmetic section, I/O devices, etc. A true array is formed when some or all of the resources required by an individual processor can be shared by the ensemble. The most common form of array is to proliferate the lower cost (by present day standards) arithmetic processing elements and to share the more expensive resources. In a very high performance array, such as envisioned for the IAP, the problem of control and data flow implied by this resource sharing is complicated by the fact that the fundamental arithmetic elements have been highly optimized. Thus, speed losses due to systems overhead for control, data transfers, and path selection, which might represent a small fraction of the total computation time in a GP machine, would cause very great inefficiencies in the IAP.

The IAP array architecture needs to provide a flexible, modular capability to support both growth potential and varied applications. For example, even though a VPE will initially have four channels, its design from the outset should take into account the need for a variable number of channels to accommodate larger or smaller vectors. Flexibility is also needed in the precision or bit width of the processors and data paths. This can readily be provided by bit-slicing the design where possible.

The modularity and reconfigurability of the IAP are both related to the testbed nature of the IAP. The IAP facility can be thought of as providing the resources needed to study various architectural schemes. Such tests will involve both actual hardware and participation of algorithm developers/users. From an architectural point of view, the resources which must be provided for such an interactive test environment will include:

1. Initial input sources.
2. Memory hierarchy/management system.
3. Various unified control and/or timing units.
4. Processing element test berths.
5. Display and output systems.
6. Data base and management schemes for handling test data, benchmarks, etc.

These general resources, shown in Figure 1, will be used for the following types of testbed investigations:

1. Hardware integrity/reliability.
2. Throughput measures (speed)
3. Technical flexibility and programmability.
4. Application/production capability and evaluation.

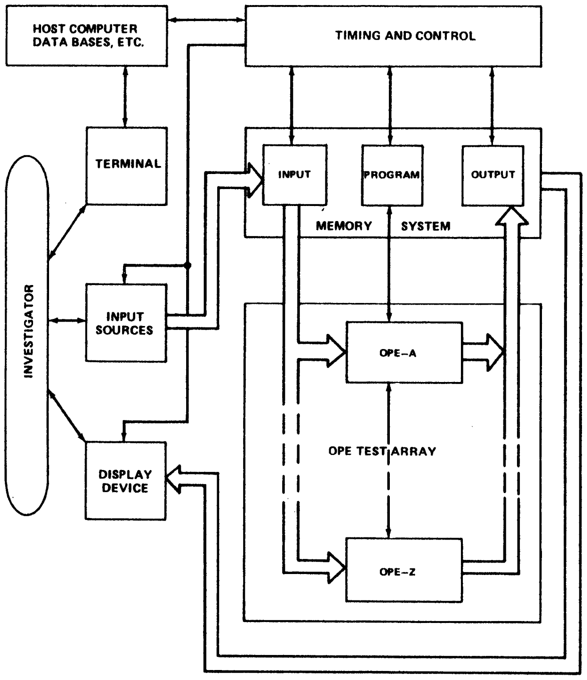


Figure 1. Resources Provided by the IAP Testbed

The overall system architectural approach should include the capability to allow the basic resources to be reconfigured in order to examine various specific architectural structures and their relationship (advantages, disadvantages, trade-offs, etc.) to actual image processing algorithms. This overall architectural reconfigurability can be separated into three levels:

1. OPE internal structures.
2. Logic controlled array structures (programmable interconnections).
3. Structures easily modified through mechanical changes--cables and plugs, card slots and cages, etc.

Such an overall system capability should be incorporated throughout the architectural planning for the IAP.

The factors discussed above could be accommodated by organizing the IAP into four major subsystems:

1. Control and Data Bussing Subsystem (CDBS) to manage the overall array.
2. Individual OPE's to provide high speed arithmetic capability.
3. Memory Hierarchy to reduce the cost of high speed accesses to large image data bases.
4. Host computer to support general services and interfaces to users and other computing systems.

The first three subsystems involve high speed, carefully balanced, state-of-the-art design concepts in order to maximize overall efficiency. Preliminary architectural considerations and possible implementations are discussed in the remainder of the paper. The host computer interface is anticipated to be reasonably straightforward and is therefore not discussed further.

IAP Subsystems

Control and Data Bussing Subsystem

The CDBS provides a framework for the solution of two related problems as the name implies:

1. Overall control of the many elements in the array.
2. Data transfers between the memory system and the elements or between elements.

These functions are interdependent for two reasons: (1) The data transfers may have to share common busses and thus require global control to coordinate the accesses of the elements; (2) When the data transfers occur depends on the "state of each machine" (element), i.e., how the programs are partitioned.

In conventional machines, there is usually no distinction made between "data" and "program." It is desirable for the IAP, however, that these two portions of image processing tasks be isolated since separate hardware for these two functions is an advantageous and natural way that parallelism can be built into the array. Furthermore, with such a structure, overlap of execution and instruction fetch can be performed more easily, and the different word width and processing requirements of program and image data can be accommodated.

Consequently, the duties of the CDBS can be divided between the data and the program. For image data, the CDBS will manage the necessary array bus structure by controlling data flow between memory and the elements and between the elements themselves. For program, the CDBS will be concerned with instruction fetch and decode at the array level and the distribution of the resultant tasks among the elements.

In addition, the CDBS will support the reconfiguration of the IAP through programmable interelement and memory busses.

Control Schemes. There are two types of control scheme that might be used in the array:

1. Interrupt driven control.
2. Predetermined assignments.

The first type is necessary when processing elements are functioning essentially asynchronously and have arbitrary tasks to perform; that is, programs which cannot be timed out exactly at the machine cycle level. This situation occurs, for example, because of data dependent computation times or branching. Thus, the elements must communicate control through a "task completed" message system. Such control strategy is common in general purpose multiprocessors executing a wide range of different tasks [7,15] and can best be used when the IAP is communicating with the external world.

The interrupt technique is more time consuming, particularly when implemented in software, than one based on predetermined assignments and timings. Since image processing involves very large volumes of well-ordered data with little data dependent branching and timing, the control structure for the algorithms can be assigned in advance. One example is a pipeline transfer system in which each element does "its own thing" but has strict protocol for passing on data to the next element at each "major cycle."

Another example is a commutator control scheme which assumes a set of homogeneous elements all executing identical programs on different data streams or portions of a picture. Thus the timing sequences are well defined. Assume, for example, that 16 cycles are needed for execution of the program. To optimize the efficiency of the memory system and CDBS, one should have 16 VPE's executing the program on 16 different "regions" of the picture. Schematically, the commutator control would consist of a 4-bit counter and a 4-to-16 line decoder. Such a scheme is shown in Figure 2. Each memory cycle, one of 16 lines would become TRUE and enable its corresponding VPE to access the memory. To synchronize this data transfer procedure and the program being executed in the VPE, two I/O buffers must be provided in the VPE input register system, one to store current data being processed, the other to receive or transmit data when its turn on the commutator occurs.

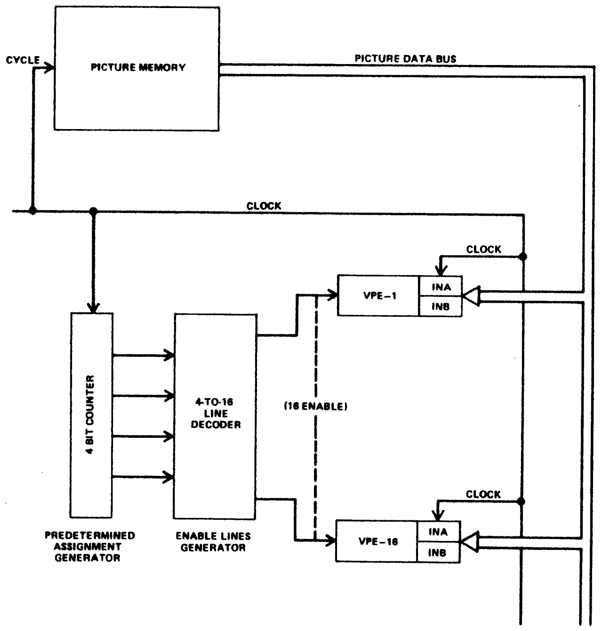


Figure 2. Commutator Control

This commutator control scheme is quite simple. It is given as one example solution to a very severe control problem: namely, to efficiently use an expensive, high performance memory resource, one has a very limited time to make "computations" or decisions about how the resource should be shared. In the example, only one memory cycle (200 ns) can be used in determining this routing. An extension of this simple commutator concept would allow preprogrammed access "segments" for each VPE.

Data Transfers. When elements need to transfer data between them, two techniques can be used as indicated schematically in Figure 3. Register-to-register direct transfer between elements will provide very high data bandwidths since a large number of data paths allow simultaneous transfers between many elements. Complexity of the interconnect structure increases substantially if programmability is to be maintained. The alternative to the direct pipeline transfer scheme is to use shared memory so that one element places data in a known common location to be accessed by other elements. This common memory technique provides a greater flexibility in access for data needed by

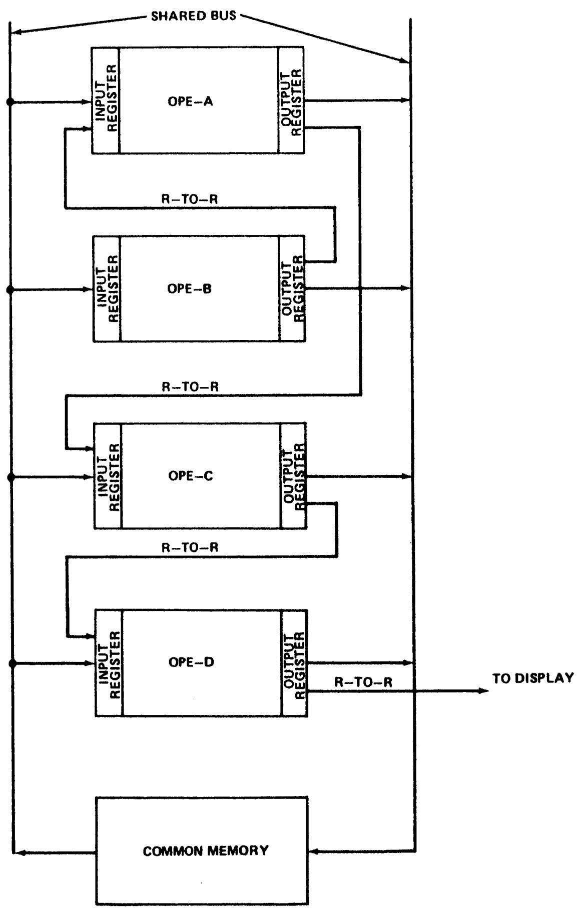


Figure 3. Types of Data Transfers

several elements. It also requires complex memory switching, but the technical difficulties are somewhat less than for programmable direct transfers because of a smaller number of data paths (memory ports). There are "lock problems" associated with this common memory type of element communication as well as its resulting lower speed. Both types of data transfer schemes have their respective advantages, disadvantages, and particular areas of applicability. The architecture of the IAP should provide the capability to investigate both types.

In order to maintain the very high data throughput demanded by the VPE and a control scheme such as described above, the image data memory and the data bus servicing the VPE's must have sufficient bandwidth. To illustrate the memory and bussing requirements, assume that the memory is divided into two image buffers, each configured to have 16 ports. (The need for this structure is discussed in the subsection on memory hierarchy.)

Control of the total buffer can conveniently be divided into two sections (Figure 4): a 16x16 switcher and an address mapper. All VPE's would provide the mapper with the required symbolic addresses. The mapper then would generate all 16 physical memory cell location requests needed to access data in the total buffer. Dividing the buffer control problem in this manner provides modularity, flexibility, and relieves the VPE from burdensome address calculations.

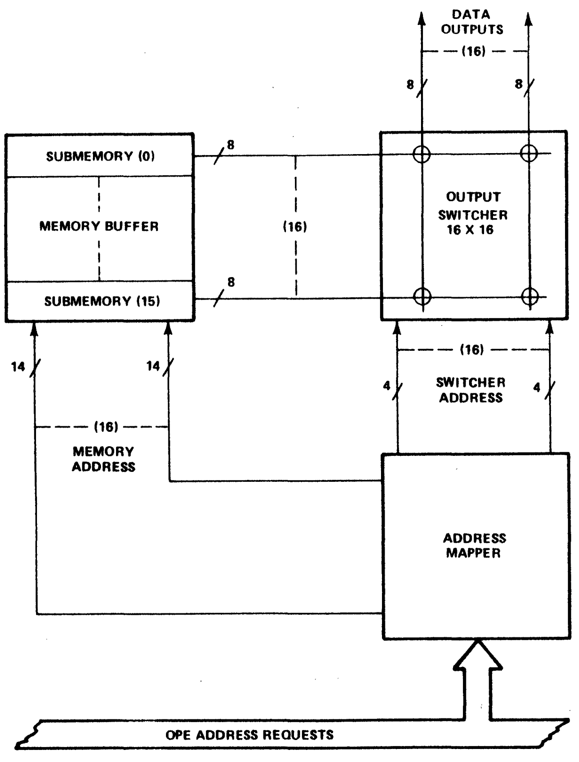


Figure 4. Picture Memory Buffer Control

The output of the "memory switch" would drive a 128 data line bus, 16 groups of eight bits each. This bus structure is shown in Figure 5. A bus from each buffer would feed direct transfer registers in each VPE as indicated above. Since any data could be put on any group of lines, it would be possible to partition the groups in terms of their destination in the VPE's. For example, data bytes 0-3 would be used for I/O in VPE channel 0 (the first vector component); 4-7 service VPE channel 1 (the second vector component); etc. Thus four vectors with four components could be transferred simultaneously on the single data bus. In the example, the first vector would use groups (0,4,8,12) for its four components.

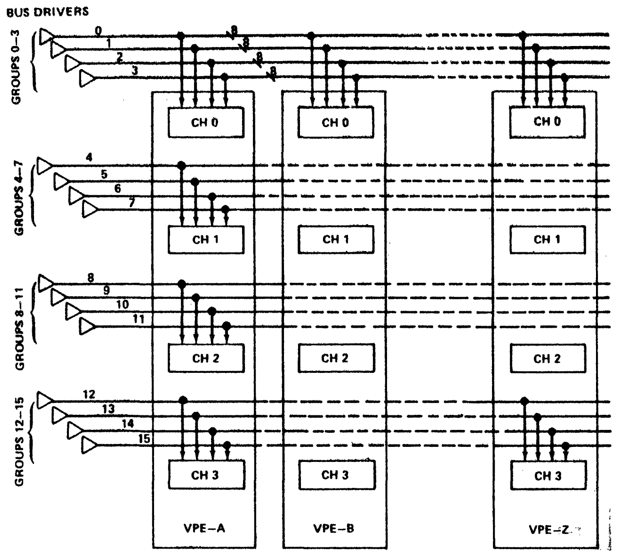


Figure 5. Picture Data Bus Structure

It is assumed that the "traffic" on these picture busses would be higher than that through any given VPE; the bus transfer rate would be 16 bytes/cycle because of its high bandwidth, although it may take many VPE cycles to "process" or use these 16 bytes. Thus the picture bus would be available for I/O transfers to other VPE's during this time. The amount of time available for such sharing would depend on the number of VPE cycles between data transfers, that is, on the algorithm complexity.

In general, a satisfactory solution to the data control problem is not obtained by connecting each of the VPE's to a particular memory port, at least not for image tasks that require many data points to initialize the algorithm. If various parameters all match, this port assignment might be possible. Thus, for example, if a particular algorithm requires 16 data points, 16 clock cycles and there are 16 VPE's, then each data port could be used to access the memory to obtain the necessary 16 data points for the next resampling during the 16 steps of the current process. This example is a special case but might be implemented as described. It would have the disadvantage not only of making the address mapper more complicated since it must handle 16 different portions of a 16 address sequence at once, but also of severely restricting the data accessible by any particular VPE.

Program Control. The host computer will be responsible for IAP program compilation or program access from bulk storage and for loading this program into the IAP master program memory. This program will consist of two parts: the processing algorithms executed by the elements of the array (OPE's) and the master CDBS control program for managing the implementation of these image algorithms. Execution of the master control program in the CDBS controller fetches the processing program from the master program memory, decodes and loads it into the control store of the OPE's. If the homogeneous array approach and predetermined assignment control of the data paths are used, all OPE's of the same class would receive the same decoded program. Execution of these OPE programs would follow initialization and receipt of a "start signal" from the CDBS master control program. Overall management of the OPE program execution by the CDBS controller will include, among other tasks, assignment of appropriate portions of the data base to each OPE, keeping track of the coordinate addresses of the image data in order that ends of lines and ends of frames may be recognized, and programming the commutator if that control scheme is utilized. Essentially, therefore, the individual elements in the array will operate on a "load and go" basis from their internal program memories which behave very much as the PROM's in a microprogrammed processor. In addition to task assignment and management of the OPE's, the CDBS will control the memory hierarchy transfers and other I/O devices.

VPE Structures

As indicated earlier, a common image processing task involves various forms of the multiply-add operation; e.g., the vector dot product. The intention of the Vector Processing Element (VPE) is to reduce such vector or multichannel operations to microinstruction level code so that they can be executed in one machine cycle like other GP basic operations such as ADD, SUBTRACT, JUMP. It is intended that the initial VPE be configured with four parallel channels and a vector dot product (contraction) capability. Such hardware will have associated with it an extended instruction set for vector operations. The initial design goal is a machine cycle time of 100 ns. A complete four vector dot product would require two such cycles--one to perform the four multiplies, and one to perform the contraction (three adds). Also two dot products between two 2-component vectors could be done in 200 ns.

This subsection contains a preliminary description of the internal structure of such a VPE. Several major tasks and thus components of the VPE have been identified. There are two types of such components--those that are replicated in each channel and those that deal with the overall operation of the VPE and are implemented only once. All four channels of a type I component will be referred to as the "system;" the individual channels will be labeled by a subscript. Thus, for example, the four multiplier units in the MULT system are designated MULT(0), MULT(1), MULT(2), MULT(3).

Type I: Each Channel
(a) Cache Memory System (CMS)
(b) Arithmetic and Logic Unit (ALU)
(c) Multiplexer Structure (MUX)
(d) Multipliers (MULT)
(e) Higher Level Arithmetic Functions (HLAF)

Type II: Overall Operations
(a) Microprogram Control Unit (MCU)
(b) Control Store and Scalar Memory (CSSM)
(c) Address Generator/Scalar Unit (AG)
(d) Special Control Registers (SCR), e.g., increment and test

A general block diagram of the VPE components is shown in Figure 6.

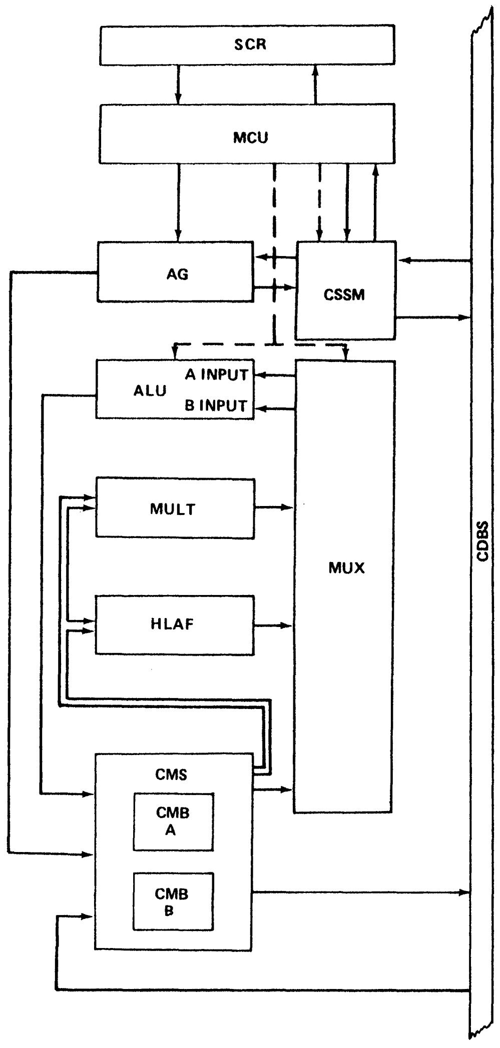


Figure 6. VPE Components

Cache Memory System. The VPE executes micro code programs which have been loaded by the CDBS into its control store. When issued a macro command from the master array control program, the VPE starts execution of the appropriate micro code and runs independently until that task is completed. All the picture data needed for such a processing task must be supplied to the CMS at the time of the macro call or must already be resident in the CMS. These special VPE picture data buffers act as ultra high speed cache memories (15 ns access time). They must have a very high bandwidth I/O port in order to be able to accept sufficient data for the complete macro operation. Such a structure can be accomplished by a combination of registers and RAM's. Two identical groups of register/RAM's are proposed for each channel of the VPE so that two vector operands can be accessed at once. They will be referred to as the IA and IB Cache Memory Buffer (CMB) for the I-th channel and respectively hold A and B operands.

Figure 7 shows one possible configuration for a cache memory buffer. Each memory is configured as a Large File and two Small Files. Each Small File provides the necessary bandwidth since it can accept four words simultaneously. Thus 16 words can be transferred to the CMS Small Files during each clock cycle. Two such Small Files are provided so that one can be accessed by the VPE while the other is loaded from the main memory structure. The Large File provides more storage capability, but with a lower I/O bandwidth. One word can be loaded into each Large File during each cycle; thus a complete four-component vector can be loaded into the VPE during each cycle in this Large File Mode.

Figure 7 shows a common 16-word bus for the two parts of the cache memory. However, it may be more desirable to separate the bussing to the Large File and Small Files at a later stage in the VPE development. The memory should be partitioned so that reading and writing can be performed simultaneously in different segments. The VPE channels access this combined cache through a common set of addresses and data paths and obtain a vector (four words) each memory reference. In Figure 7 the block marked output is used to indicate this address and data structure.

Initially these memories are configured with 16 bits in each word. This word width should be used for the prototype VPE evaluation in order to allow a greater range of programming possibilities. The memories will be bit sliced so that future modification of word width can easily be accommodated.

Multiplexers. Since the VPE has multiple arithmetic channels, it must have a fairly flexible internal data path structure. This feature, at the processing element level, is equivalent to the overall data flow within the IAP itself. However, it is necessary to control the VPE data paths through software by microcode logic since these elements are basic IAP building blocks. An example multiplexer structure is shown in Figure 8.

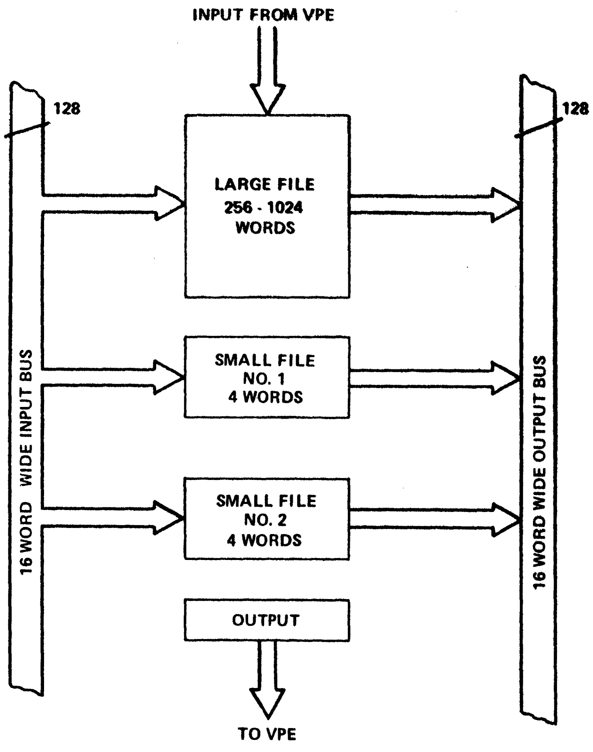


Figure 7. Cache Memory Buffer

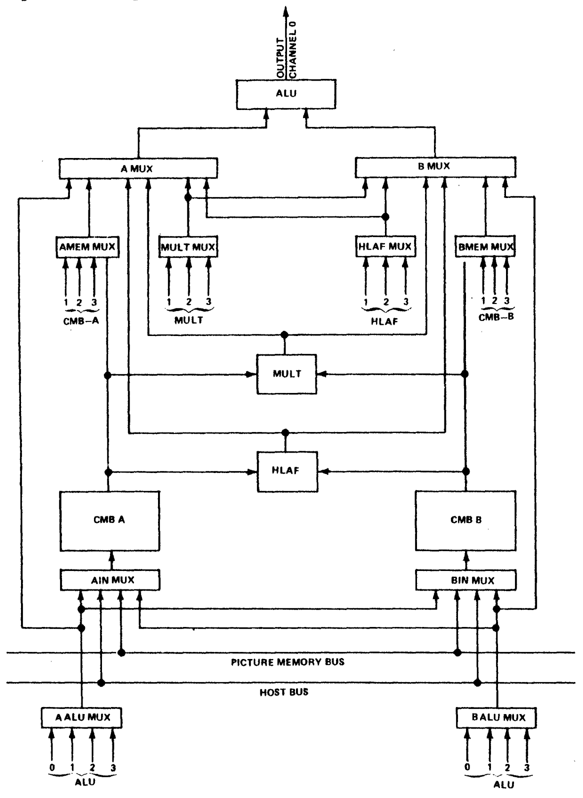


Figure 8. VPE Multiplexer Structure (Channel 0)

The equivalent control requirements at the array level could be relaxed since such control is used to investigate different fundamental architectural approaches to arrays and is not needed during program execution. At that level, such logic is essentially one of convenience. At the OPE level, however, this relaxation/convenience is definitely not possible. The microprogram must flexibly control the data paths in order for the elements to even function. The multiplexer system is used to provide this capability. The different data path combinations may at first sight seem more extensive than necessary. One must remember, however, that the VPE is an R&D tool to investigate such data path combinations (among other things) and therefore no restrictions have been placed on this part of the VPE.

Multipliers and Higher Levèl Arithmetic Functions. The fixed-point multipliers for a prototype VPE will accept 16-bit operands and produce a 32-bit product. A provision is made for selecting either the 16 most significant or least significant bits of the product. This output should be available at its appropriate ALU through the multiplexer structure. The ALU can perform further operations on the product or can transfer it directly to the accumulator.

It is anticipated that other mathematical functions such as divide and square root might be implemented in hardware in the VPE in a manner similar to multiply. Provision for these elements has been included in the multiplexer structure.

Arithmetic and Logic Unit. The ALU provides the usual Boolean, ADD, and SUBTRACT functions. It may be desirable to include a shifter structure at the output of each ALU(I) to perform fast left and right rotates and arithmetic shifts. Furthermore various flags will be set by the ALU components to indicate carry, overflow, +, 0, -, etc. More generally, as many advanced minicomputer type op codes, e.g., PDP 11, as possible will be incorporated into the VPE instruction set and will be performed in the ALU. Restriction may be necessary in the general branching instructions. This limitation, however, is quite appropriate for well-ordered image data in which data dependent branching for each pixel is not prevalent.

Microprogram Control Unit. The Type II VPE elements will be discussed below. The microprogram control unit performs various functions:

1. Timing
2. Instruction fetch and overlap
3. Any further decoding necessary
4. Testing of special registers
5. Multiplexer control
6. ALU and AG control

These functions, at the element level, correspond to the tasks of the CDBS at the array level.

Control Store and Scalar Memory. The microprogram control unit has its own storage which is loaded from the CDBS. This storage unit manages its multiport capability under interaction of the VPE control and the IAP master control. The CSSM contains the very wide words necessary to set the many internal logic lines in the MUX, ALU, etc.

Address Generator. In order to allow the arithmetic section to run with maximum efficiency on vector data, a separate unit should be used to handle address calculations and various scalar functions. In general, three addresses must be generated for each vector operation--two for operands, one for destination (result). The AG should provide indexing and incrementing capability to assist in such address calculations.

Special Control Registers. A very time-consuming function in handling image arrays in a GP computer is indexing the array. The VPE should have various special registers, independent of the ALU, to handle this function. For example, as one scans a picture using pixel coordinates (I,J), the following code, which would normally be executed for each pixel, should be explicitly executed in hardware by the SCR.

```
#1  ADD    1 to J
    JMP    $J \leq J_{END}$ to #1
    LOAD   J with $J_{BEGIN}$
    ADD    1 to I
    JMP    $I \leq I_{END}$ to start
```

Thus during the image task initialization, the SCR would be programmed for a particular array task. It then operates independently of the MCU except when the SCR interrupts the microprogram to indicate the scanning status.

Memory Hierarchy

The IAP has been proposed as a very high performance testbed with pixel processing times in the neighborhood of 200 ns. This high throughput demands very careful analysis of the memory subsystem. There are two differing requirements which must be balanced:

1. Very large volume of data storage (40-280 megapixels for multichannel EOS data).
2. Fast random access within large blocks of this data.

To meet these needs a hierarchical memory system is proposed, as illustrated in Figure 9.

It is assumed that the "original source" has the ability to supply data for such a high throughput facility. Thus raw data might come from

1. Live sensors
2. HDDT (15-30 Mbits/second, preferably 2-5 times higher bit rates)
3. Multiple CCT's, discs, etc.

"Circulating memory," either true discs or CCD storage, costs an order of magnitude less than random access memory, but still can have very high bit rates after "track acquisition." Therefore, it should be used for the basic "scene storage device." This temporary mass scene storage will be interfaced to the input buffer memories in block transfers possibly in a ping-pong fashion.

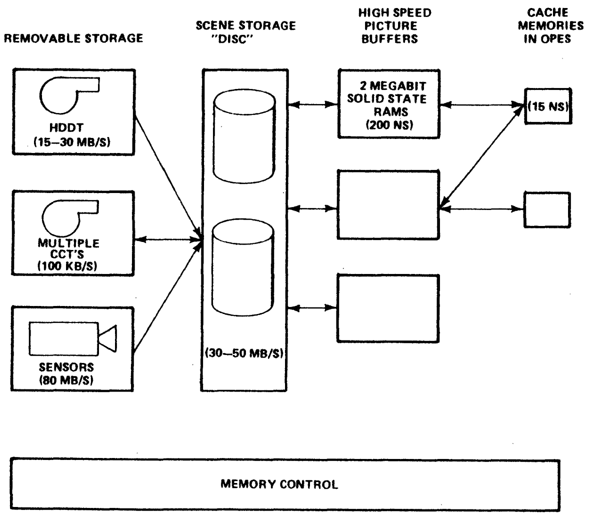


Figure 9. Memory Hierarchy

A convenient fundamental image size to be accessed by the IAP is a 512x512 image--the nominal size of a video frame. A picture buffer memory of 2 Mbits will hold such a frame with 8 bits assigned to each point. An I/O port suitable for real-time video applications has been developed for such a 2 Mbit buffer [16].

More than one such buffer could be used if more sampled data is associated with each point, for example, multispectral scanner data. Solid state random access memory with a maximum cycle time of 200 ns (preferably less) can be used for these buffers. Their data I/O ports would have a very high bandwidth, 640 Mbits/second, which could be provided by configuring them with 128 data lines in and 128 data lines out. These lines can be thought of as 16 eight-bit bytes and will be multiplexed through 16 data ports (bytes) by the CDBS; that is, any of the 16 input bytes can be written into any of the 16 memories and likewise any memory output can be obtained at one of the 16 output ports. Each of the submemory buffers is configured as 1 byte by 16K or 128K bits.

The final element in the picture memory hierarchy is the ultra high speed cache memories (15 ns) in the individual OPE's. These should be configured to allow transfer of input or output blocks of 16-32 pixel data to the picture buffers during one clock cycle.

Other types of memory are also needed for the operation of the array. Various levels of program store are anticipated. The overall control will be programmable and thus its master program requires a control store. This master control program has a counterpart at the individual OPE level. In addition to the cache picture memories, each OPE will have its own control store for the intermediate storage of addresses and partial results requiring larger words than the eight bits which are associated with data samples. This control store is similar to the VPE CSSM.

Homogeneous Array for Video Resampling

To illustrate and help to unify the above concepts, consider the following particular example problem. Can one perform real-time video resampling to provide synthetic rotation and translation? What kind of array configuration would be needed?

First, one needs two input buffers which operate in a ping-pong fashion--one receiving NTSC camera data (a sample point every 100 ns), the other providing data to an array of VPE's which perform the actual resampling. If a four-vector dot product requires 200 ns, then the actual interpolation is done in 1.0 μs in each VPE. One might allow six 100 ns clock cycles to perform address calculations since (1) appropriate increment and test registers are part of a VPE's control unit, and (2) there is a memory address mapper that accepts pixel addresses and returns the appropriate intensity data. The total resampling in one VPE thus would require 1.6 μsec and 16 such elements would operate in parallel.

The homogeneous array approach is most applicable for this problem. One would cycle the input memory every 100 ns and distribute the appropriate input image data to each of the 16 VPE's on a commutator basis. The total picture buffer can be loaded so that each of the 16 intensity values of any contiguous block of 4x4 image data is contained in a separate submemory. Therefore, they can all be accessed during one memory reference cycle. As a result, each 1.6 μs all VPE's can be loaded with the data needed for the resampling of the new pixel they are working on. The memory controller shares this total time among all 16 elements giving each one 1/16 of the time to access memory. This procedure is extremely well ordered and should cause no problems in implementation. All VPE's would be executing the same op code; furthermore, this redundancy is needed to achieve such high throughputs. The commutation technique would require that the VPE input buffers be "double buffered" to skew the data. As the resampled data is generated, it would be buffered by the scan line and then immediately displayed. Alternatively, it could be stored in an output picture memory similar to the input system.

The above memory controller/commutator is an extension of the "bit switch" concept used in multiprocessor configurations [7]. Because of the overall uniformity of the problem and image data, however, control of the switch is provided through the initial programming of the commutator and memory accesses, which would be costly in time and hardware, are not left up to the individual VPE's. The individual elements "get a chance" at the main (expensive) memory when their time comes and can access any locations for which they have calculated addresses. This type of overall control effectively solves the problem that 16 processors must all be coordinated and only 100 ns can be devoted to each for this purpose. Such high performance requirements involving very large data volumes are common in image handling, particularly in the area of preprocessing; a fast, orderly control system such as a commutator is most appropriate.

It is worthwhile to note the performance of this system. For each new (resampled) pixel, the IAP must perform:

20 multiplies, 15 ADDS	interpolation
4 multiplies, 4 Adds 2 increment 2 test (branch)	address calculation

If all operations are treated as equivalent, 47 operations are performed in 100 ns. Thus the 16 VPE system is operating at 470 MIPS. If the multiply is equivalent to three ADD instructions, this figure becomes 950 MIPS. Furthermore, these figures do not count the usual overhead operations associated with the pixel address generation.

The memory system, processing capability, and control described above have been evenly matched in throughput. Each subsystem is kept completely saturated at all times and the system overhead has been reduced to a minimum. Furthermore, the basic design concepts would allow this hardware to be reprogrammed to perform another completely different task, for example, maximum likelihood classification.

Acknowledgement

The authors wish to thank Mr. Ken Kadrmas, Information Analysis Branch, Data Systems Laboratory, NASA/MSFC, for his cooperation and helpful suggestions as technical monitor of this project.

References

[1] R. L. Lillestrand and R. R. Hoyt, "The Design of Advanced Digital Image Processing Systems," Photogrammetric Engineering, (October 1974), pp. 1201-1218.

[2] G. R. Allen, et al, "The Design and Use of Special Purpose Processors for the Machine Processing of Remotely Sensed Data," Proc. Purdue Conference on Machine Processing of Remotely Sensed Data, (October 1973), pp. 1A-25 - 1A-42.

[3] "A Proposal to Grumman for an EOS Data Processing System Study," Control Data Corporation, ADD503, (March 1974), p. 4-5.

[4] S. S. Rifman, "Evaluation of Digitally Corrected ERTS Imagery," Symp. on Management and Utilization of Remote Sensing Data, Sioux Falls, (1973), pp. 206-220.

[5] G. H. Barnes, et al, "The ILLIAC IV Computer," IEEE Trans. on Computers, (August 1968), pp. 746-757.

[6] A. J. Evensen and J. L. Troy, "Introduction to the Architecture of a 288-Element PEPE," Proc. Sagamore Computer Conference on Parallel Processing, (August 1973), pp. 162-169.

[7] W. A. Wulf and C. G. Bell, "C.mmp--A Multi-Mini-Processor," Proc. FJCC, (December 1972), pp. 765-777.

[8] J. A. Rudolph, "A Production Implementation of an Associative Array Processor," Proc. FJCC, (December 1972), pp. 229-241.

[9] F. Kriegler, et al, "Multivariate Interactive Digital Analysis System (MIDAS): A New Fast-Multispectral Recognition System," Proc. Purdue Conference on Machine Processing of Remotely Sensed Data, (October 1973), pp. 4B-51 - 4B-64.

[10] J. Gilbert, et al, "EOS Image Data Processing System Definition Study," NASA/GSFC Study Report, General Research Corp., (September 1973), pp. 51-74.

[11] W. R. Smith, et al, "AN/UYK-17 (XB-1)(V) Signal Processing Element Architecture," Communications Sciences Division, Naval Research Laboratory, NRL Report 7704, (June 1974), 103 pp.

[12] "SPS-81 Signal Processor: User's Manual," Signal Processing Systems, Inc., A008-01C1074, (1974).

[13] "Macro Arithmetic Processor Systems: Programmer's Reference Manual," CSP, Inc., Document No. JB 6000-001-00, (May 1975).

[14] "Introduction to the Hughes Modular Programmable Signal Processor," Hughes Aircraft Company, Report No. M75-09, (January 1975).

[15] F. E. Heart, et al, "A New Minicomputer/Multiprocessor for the ARPA Network," Proc. AFIPS NCC, (1973), pp. 529-537.

[16] P. L. Neely and R. M. Brown, "A Digital Video Image System," Submitted for publication to IEEE Trans. Comput.

PARALLELISM IN AI PROBLEM SOLVING: A CASE STUDY OF HEARSAY II

R. D. Fennell and V. R. Lesser
Department of Computer Science[1]
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

SUMMARY

This paper presents a design for the organization of knowledge-based AI problem-solving strategies which is felt to be particularly applicable for implementation on closely-coupled multiprocessor computer systems. The method of design is a result of formulating the problem-solving organization in terms of the *hypothesize-and-test paradigm* for heuristic search, where the various hypothesizers and testers are represented as knowledge sources applicable to the task domain of the problem being solved. A *knowledge source* may be described as an agent that embodies the knowledge of a particular aspect of the problem domain and is useful in solving a problem from that domain by performing actions based on its knowledge so as to further the progress of the overall problem solution. The hypothesize-and-test paradigm provides the conceptual means of coordinating these various knowledge source activities by suggesting that it is the function of some knowledge sources to create *hypotheses* representing a possible (perhaps partial) solution state for the given problem. Hypotheses are created in a global data base; these are available for inspection by all knowledge sources. It is the responsibility of other knowledge sources to evaluate these hypotheses in light of their own knowledge of the task domain, and either accept or reject the hypotheses, or propose their own alternative hypotheses (by either modifying the existing hypotheses or creating entirely new ones).

The Hearsay II speech-undestanding system (HSII), which has been developed at Carnegie-Mellon University using the techniques for system organization described here, has provided a context for evaluating this system architecture. The HSII organization provides the facilities necessary for knowledge-source cooperation through the hypothesize-and-test paradigm to be carried out in a highly asynchronous and *data-directed* manner, where knowledge sources are specified as independent processing entities capable of parallel execution; the activities of any given collection of such knowledge sources are coordinated by the hypothesize-and-test paradigm through the use of a shared global data base called the *blackboard*.

In specifying the blackboard as the primary means of interprocess communication, particular attention has been paid to resolving the *data access synchronization* problems and *data integrity* issues arising from the asynchronous data access patterns possible from the various independently executing parallel knowledge-source processes. A non-preemptive data access allocation scheme was devised in which the units of allocation could be linearly ordered and, hence, allocated according to that ordering so as to avoid data deadlocks. The particular units of data allocation were chosen as being blackboard nodes (hypotheses and links), these nodes also representing the units of data creation within the blackboard. Since the blackboard data base is a dynamically expanding structure, the mechanism of data access synchronization according to existing data objects is not sufficient to provide the access synchronization required when multiple knowledge sources are capable of simultaneously hypothesizing (creating) identical hypotheses on the blackboard without being required to link these hypotheses to previously existing data nodes. Assuming it is undesirable to have identical (duplicate) nodes in the blackboard (as is the case in the HSII organization, since one of the design goals was to minimize the duplication of information within the blackboard so as to minimize the duplication of processing which would result from such replicated information), a mechanism had to be provided whereby a knowledge source could acquire access to a section of the blackboard which did not yet exist. The *region locking* mechanism satisfies these requirements by viewing the blackboard as an abstract data space in which access rights to abstract regions could be granted without regard to the actual data content of these regions. However, since both the node accessing mechanism and the region accessing mechanism have the capability of allocating access rights to essentially the same data structure, the two forms of data access allocation must be closely coordinated so as to avoid data deadlocking and data access race conditions.

[1] This research was supported in part by the Defense Advanced Research Projects Agency of the Office of the Secretary of Defense (Contract F44620-73-C-0074) and monitored by the Air Force Office of Scientific Research.

Another area of concern relating to the use of a shared blackboard-like data facility relates to the assumptions made by the various executing knowledge sources concerning issues of data integrity and localized data contexts. Since the blackboard is intended to represent only the most current global status of the problem solution state, mechanisms were introduced to allow individual knowledge sources to retain recent histories of modifications made to the dynamic blackboard structure in the form of *local contexts*. Knowledge sources are also permitted to mark (*tag*) arbitrary fields (or nodes or regions) of the blackboard itself (without requiring continuing access rights to the field being tagged) so as to be able to monitor (in a non-interfering way) those locations for subsequent changes; the knowledge source will then be sent a *message* should any modification be performed upon a tagged field. Local contexts provide knowledge sources with the ability to create a local data state which reflects the net effects of data events which have occurred in the data base since the time of the knowledge source activation. Combined with the blackboard data tagging capabilities, local contexts also provide a means by which knowledge sources can execute quite independently of any other concurrently executing knowledge sources (and without interfering with the execution progress of any of these processes). When a knowledge source is about to modify the blackboard and has acquired exclusive access rights to the necessary data fields, it can request the receipt of any tagging messages that may have been sent to it; by interrogating its local context for the effects of these changes, a *revalidation check* may be performed on the advisability of proceeding with the intended blackboard modifications.

In an attempt to improve the problem-solving efficiency of a multiprocessor implementation of the system by increasing the amount of potential parallelism from knowledge source activity, the logical functions of knowledge source execution are split into separate processing entities (called precondition and knowledge-source processes). A *precondition process* is responsible for monitoring and accumulating blackboard data events which might be of interest to the knowledge source associated with the precondition; when the appropriate data conditions for the activation of the knowledge source exist in the blackboard, the precondition instantiates a process based on its associated knowledge source, giving it the data context in which the precondition was satisfied. Two primary mechanisms are provided to support the asynchronous form of precondition and knowledge source interaction that results from allowing preconditions and knowledge-source processes to execute concurrently. The first of these mechanisms relates to the way preconditions become activated; the second responds to the problems involved in having to schedule the many processes that may be capable of running so as to best serve the objectives of efficient problem-solving.

The process activity of HSII is intended to be very *data-directed* in nature, basing the decisions as to whether a knowledge source action can be performed on the dynamic data state represented in the blackboard data base. It is the responsibility of a precondition to test this data state for conditions which would warrant the instantiation of the knowledge source associated with the precondition. The activation of the precondition itself is also data-directed, being based on *monitoring* for the more primitive blackboard modification operations which knowledge-source processes may invoke to effect the results of their computation. This blackboard monitoring is implemented by having each blackboard modification operator be responsible for the activation of preconditions which are monitoring for data events being caused by the particular modification operation.

While precondition activity might be requested as the result of a blackboard monitoring operation, and knowledge source activity might be requested as a result of precondition satisfaction, some care must be exercised in allocating processing power to these possible sources of activity. In particular, it is likely that there will be many more processes capable of executing or requesting computing resources than can be serviced within the constraints of a reasonable problem solution time. Even if there is not an excess of requested processing activity, system performance can often be improved by the use of a *goal-directed scheduler* who is responsible for allocating processing resources so as to execute those processes first which can best promote the progress of the overall problem solution. The process evaluation functions used within the goal-directed scheduler are based on *attention focusing parameters* associated with the various components of the blackboard data base. *Policy knowledge-sources* are used in calculating these attention focusing parameters based on the occurrence of various important blackboard data events; such policy knowledge-sources are also responsible for propagating the effects of such events throughout the blackboard, so that proper attention is paid to these events and processing power may be allocated accordingly.

In order to indicate the nature of the performance of the HSII organization when run in a closely-coupled multiprocessor environment, a simulation system was imbedded into the multiprocess implementation of HSII on the DEC PDP-10. While the results of the simulation are admittedly based on a small (but computationally expensive) set of sample points, they have generally indicated the applicability of this system organization to such a hardware architecture. Given the knowledge-based decomposition of a problem-solving organization as prescribed by the HSII structure, effective parallelism factors of four to six were realized even with a relatively small set of precondition and knowledge-source processes, with indications that up to twelve processors could be totally utilized, given appropriate usage (or structuring) of the data access synchronization mechanisms. Experiments thus far have indicated that careful use of the locking structure is required in order to approach the optimal utilization of any given processor configuration (unless there exist so many ready processes that the number of suspended processes does not matter much, as is the case in configurations of four or fewer processors). An extended use of non-interfering tagging seems to be indicated, along with a reduction in the use of region-locking (perhaps substituting region-examining or node-locking wherever possible). Measurements were also made of various primitive operations at the systems level which are required in order to implement the data-directed multiprocess structure of HSII. While all these results are of a preliminary nature (and hence are subject to variation as various components of the given implementation are improved in their relative efficiencies), they seem to indicate that the HSII organization is indeed applicable for efficient use in a closely-coupled multiprocessor environment.

SELECTED REFERENCES

Baker, J. (**1974**). "The DRAGON System -- An Overview," in **Proc. IEEE Symp. Speech Recognition**, Carnegie-Mellon Univ., Pittsburgh, Pa., April 1974, pp. 22-26; also appeared in **IEEE Trans. on Acoustics, Speech, and Signal Processing**, ASSP-23, 1, pp. 24-29 (Feb. 1975).

Bell, C. G., R. C. Chen, S. H. Fuller, J. Grason, S. Rege, and D. P. Siewiorek (**1973**). "The Architecture and Application of Computer Modules: A Set of Components for Digital Systems Design," COMPCON 73, San Francisco, Calif.

Bell, C. G., W. Broadley, W. Wulf, A. Newell, et al. (**1971**). "C.mmp: The CMU Multi-mini-processor Computer," Tech. Rep., Comp. Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.

Coffman, E. G., M. J. Elphick and A. Shoshani (**1971**). "System Deadlocks," **Computing Surveys 3**, 2, pp. 67-78.

Erman, L. D., R. D. Fennell, V. R. Lesser and D. R. Reddy (**1973**). "System Organizations for Speech Understanding: Implications of Network and Multiprocessor Computer Architectures for AI," **Proc. 3rd Inter. Joint Conf. on Artificial Intel.**, Stanford, Calif., pp. 194-199.

Erman, L. D. (**1974**). "An Environment and System for Machine Understanding of Connected Speech," (Ph.D. Thesis), Comp. Sci. Dept., Stanford Univ., Stanford, Calif.

Erman, L. D., and V. R. Lesser (**1975**). "A Multi-Level Organization for Problem Solving Using Many, Diverse, Cooperating Sources of Knowledge," Tech. Rep., Comp. Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.

Feldman, J. A., and P. D. Rovner (**1969**). "An Algol-based Associative Language," **Comm. ACM 12**, 8, pp. 439-449.

Feldman, J. A., et al. (**1972**). "Recent Developments in Sail -- An Algol-based Language for Artificial Intelligence," **Proc. FJCC.**

Fennell, R. D.(**1975**). "Multiprocess Software Architecture for A.I. Problem Solving," Tech. Rep.(Ph.D. Thesis), Comp. Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.

Heart, F. E., S. M. Ornstein, W. R. Crowler and W. B. Barker (**1973**). "A New Minicomputer/Multiprocessor for the ARPA Network," **Proc. AFIPS**, NDD42, pp. 529-537.

Lesser, V. R. (**1972**). "Dynamic Control Structures and Their Use in Emulation," Tech. Rep. CS-309 (Ph.D. Thesis), Comp. Sci. Dept., Stanford, Univ., Stanford, Calif.

Lesser, V. R., R. D. Fennell, L. D. Erman and D. R. Reddy (**1974**). "Organization of the Hearsay II Speech Understanding System," in **Proc. IEEE Symp. Speech Recognition**, Carnegie-Mellon Univ., Pittsburgh, Pa., April 1974; also appeared in **IEEE Trans. on Acoustics, Speech, and Signal Processing**, ASSP-23, 1, pp. 11-23 (Feb. 1975).

Lesser, V. R. (**1975**). "Parallel Processing in Speech Understanding Systems: A Survey of Design Problems," in D. R. Reddy (ed.) **Invited Papers of the IEEE Symp. on Speech Recognition**, April 1974, Pittsburgh, Pa., Academic Press.

Newell, A. (**1973**). "Production Systems: Models of Control Structures," in W. C. Chase (ed.) **Visual Information Processing**, Academic Press, pp. 463-526.

Ohlander, R. B. (**1975**). "Analysis of Natural Scenes," Tech. Rep. (Ph.D. Thesis), Carnegie-Mellon Univ., Pittsburgh, Pa.

Swinehart, D. and R. Sproull (**1971**). **SAIL**. Stanford AI Proj. Operating Note 57.2, Stanford Univ., Stanford, Calif.

DATA FLOW LANGUAGES

James Rumbaugh
General Electric Research Center
Schenectady, New York 12345

Abstract -- The sequencing of data flow instruction execution depends only on the availability of required operands. Because the execution of each instruction is independent, concurrency is easily expressed in data flow notation.

Data Flow Programs

Data flow languages [1,2] are programming notations in which data dependencies are represented directly by program structure. A data flow program is a graph of functional operators connected by data links. Each type of operator has input and output links, and specifies a function (such as addition, multiplication, or comparison) from data values on input links to data values on output links. All operations are local; operators have no side-effects. Each data link connects the output of one operator to the input of another operator; links specify the data dependencies of a program. An entire data flow program defines a function which is the composition of the functions specified by the operators. (Fig. 1)

Tokens

During the execution of a program, data values reside on certain data links at any moment. A value on a link, called a token, represents an intermediate result of a computation which has not yet been used. Tokens can contain values from the domains of the primitive functions composing the instruction set, but each value is independent of values that are concurrently present on other links. There are no pointers, references, shared cells, or L-values in the data flow language; each value is accessible to only a single operator.

Instruction Execution

The execution of all operators is concurrent and independent. An operator becomes enabled (able to execute) when tokens are present on all its input links. The operator then swallows up its input values and computes the output values as functions of the input values. The output values are emitted onto the output links and the operator returns to the inactive state. Operator execution depends only on information local to an operator; there are no global variables or side-effects. Operators have no internal memory between executions. (Fig. 2)

Structures

Data flow structures are lists of components, which can be primitive values or simpler structures (no recursive structures are allowed). Data flow structures can be represented as trees with ordered branches whose leaves are primitive values. Data flow tokens can contain structures as values. Each structure is independent of all other coexisting structures. Structure operations include composition and decomposition of structures, extraction of components, and modification of components. (Fig. 3)

Control Operators

Functional operators are sufficient to build expression trees (and graphs). The Switch and Union operators permit the construction of conditionals and loops. A Switch accepts an arbitrary number of switched inputs of arbitrary types and one control input of boolean type. It has two sets of outputs, designated true and false outputs, corresponding to the set of switched inputs. When tokens are present on all its input links, the Switch operator copies each switched input value onto the corresponding output link according to the value of the control token.

The Union operator is the inverse of the Switch. It has two sets of input links and one set of output links. When a token appears on an input link, its value is copied onto the corresponding output link. (Fig. 4)

A conditional has a Switch at its head, two arms, and a Union at its foot. Each arm is a data flow program which is disjoint from the other arm and the rest of the program. All values for the arms enter or leave through the Switch or Union operators. A loop has a Union at its head and a Switch at its foot. The body of a loop must be disjoint from the rest of the program and all values must enter the body through the Union operator. (Fig. 5)

Program Syntax

Good behavior of programs can be guaranteed by simple syntax requirements. Well-formed data flow programs are determinate and deadlock-free. They are defined recursively as one of the following: an operator, an acyclic graph of data flow programs, a conditional whose arms are disjoint data flow programs, or a loop whose body is a data flow program. Such programs define partial functions from input values to output values (total functions if the programs always terminate).

Procedures

Any data flow program (without tokens) can be considered to be a procedure, which can be held as a token value. Procedures can be restricted to have a single input and a single output link without loss of generality, since input or output values can be structures. Procedures can be copied and passed as values. Procedures are called using the Apply operator, which requires a procedure-valued input and an arbitrary input as the argument. The Apply operator swallows its input values, places the argument value on the input link of the procedure, and allows the procedure to execute within the Apply operator. When the procedure execution terminates, the procedure is discarded and the result value is emitted by the Apply operator as output. Alternatively, the procedure execution can be regarded as occurring outside the Apply operator, in parallel with the execution of the calling procedure. Each instance of procedure execution, called a procedure activation, contains a return pointer to indicate the destination of the result. Because an activation may in turn create other subordinate activations, the state of execution at any time is a tree of procedure activations. The root of the tree is created in response to a request external to the model.

In a data flow program, each procedure activation, active operator, and data value is independent of all concurrent activations, operators, and values. Operations can therefore be performed concurrently without interference. In a well-formed program, the sequence of execution of parallel operators does not affect the eventual result. Because each operator executes as soon as its required operands are available, data flow notation does not impose arbitrary sequencing constraints on an algorithm.

Several independent data flow models have been developed, but they are all similar to the model described here. Some models have been used as the bases for highly parallel machines [3,4,5].

References

[1] James Rumbaugh, A Parallel Asynchronous Computer Architecture for Data Flow Programs, M. I. T. Project MAC, TR-150, (May 1975), 319 pp.

[2] Jack B. Dennis, "First Version of a Data-Flow Procedure Language," M. I. T. Project MAC, Computation Structures Group Memo 93-1, (Aug 1974).

[3] James Rumbaugh, "A Data Flow Multiprocessor," these proceedings.

[4] Jack B. Dennis, "Packet Communication Architecture," these proceedings.

[5] David P. Misunas, "Structure Processing in a Data-Flow Computer", these proceedings.

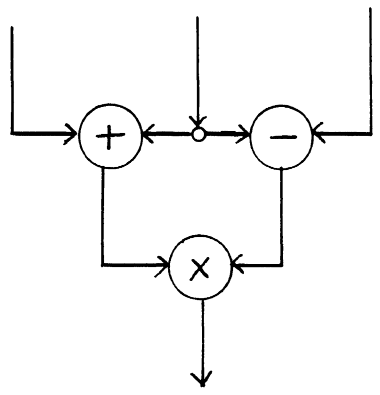


Fig. 1
Data Flow Program

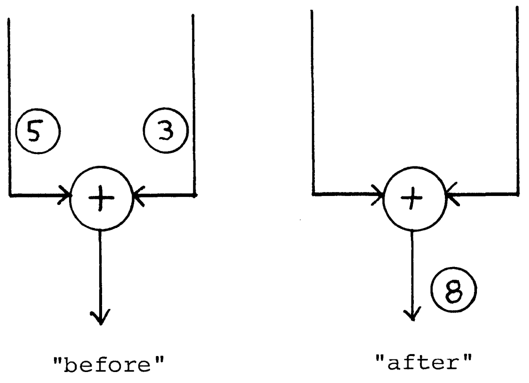


Fig. 2
Execution of an Add Operator

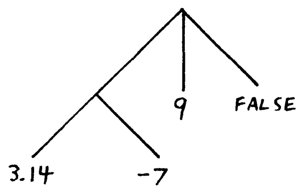


Fig. 3
Data Flow Structure

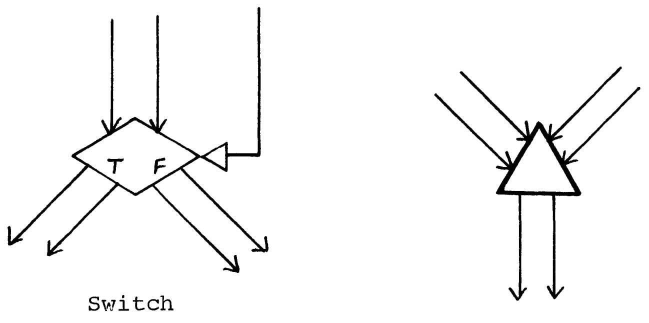


Fig. 4
Control Operators

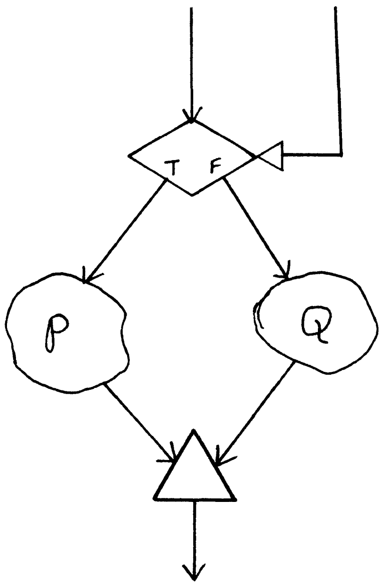


Fig. 5
Conditional

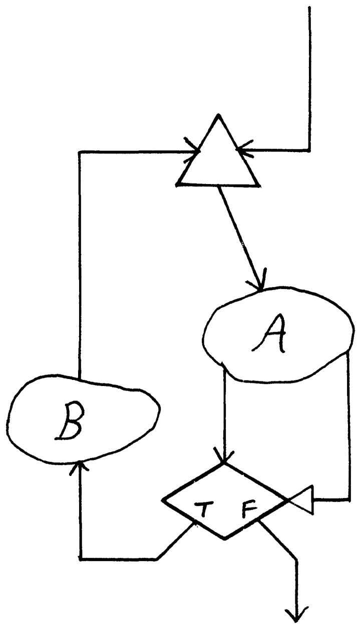


Fig. 6
Loop

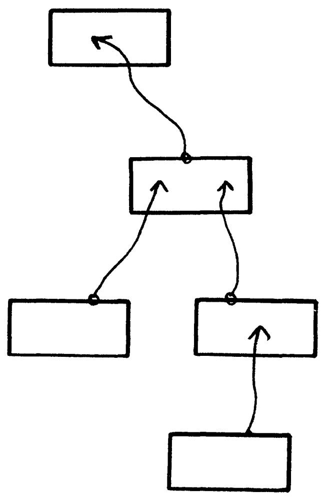


Fig. 7
Activation Tree

A DATA FLOW MULTIPROCESSOR

James Rumbaugh
General Electric Research Center
Schenectady, New York 12345

Abstract -- The Data Flow Multiprocessor is a network of independent modules which execute data flow programs. Modules communicate solely by passing asynchronous messages. The machine contains extensive concurrency.

Major Modules

The Data Flow Multiprocessor [1], shown in Fig. 1, executes programs expressed in data flow notation [2]. The machine is composed of several asynchronous major modules: Each of the activation processors holds and executes a single procedure activation. The scheduler module coordinates the processors and assigns activations to processors. The structure memory holds data structures too large to fit in activation processors. The structure controllers operate on structures for the processors. The program memory holds procedures which can be called. The swap memory holds procedure activations which are temporarily dormant. The swap network transfers procedure activations between swap and program memories and activation processors. The peripheral processor connects the machine to the outside world.

Each major module is composed of smaller basic modules. Each basic module is an asynchronous finite state machine which executes concurrently with and independently of all other modules. Modules interact by sending messages over one-way asynchronous communication channels connecting pairs of modules. The machine contains no global clocks. The machine description is very clean, because the observable effects of a module are limited to the sequences of messages it transmits.

Activation Processor

Activation processors, shown in Fig. 2, perform the actual computation of programs. Each processor contains the instructions and data of one procedure activation. Concurrent activations of a single procedure are independent. A processor comprises a local memory, an execution pipeline, and an activity counter.

Local Memory

The local memory contains instructions, data, and enabling counts for an activation. Each data flow operator is coded as one instruction. An instruction contains an opcode, the addresses of the operands and results, and the addresses of the successor instructions (those instructions that use the results. A data flow token is coded as a data value and an enabling count. The enabling count equals the number of missing operands for the corresponding instruction. The count is initially equal to the total number of operands required by an instruction; the count is decremented as each operand is stored in memory. When it becomes zero, the instruction is enabled. For example, the enabling count of a binary instruction would initially be two.

Execution Pipeline

The execution pipeline executes data flow instructions. The modules of the pipeline are organized into a circle, so that an instruction can initiate the execution of its successors. Each pipeline module performs one stage in the execution of an instruction. The modules are the activity list, the decoder, a set of functional units, and the updater.

The activity list holds the addresses of enabled instructions waiting their turns at execution. The decoder pulls the address of an enabled instruction off the activity list, obtains the instruction and its operand values from local memory, and forms the information into an instruction packet which is directed to the appropriate functional unit for execution. There is a functional unit for each type of data flow operator (one unit might perform several related operations, such as addition and subtraction). All the functional units are connected in parallel and execute independently. When it is free, a functional unit accepts an instruction packet from its input channel and performs the indicated computation. The result, together with the addresses of the result and the successor instruction, are formed into a result packet which is sent to the updater. The updater accepts result packets from all the functional units via an arbiter. When it receives a result packet, the updater stores the result value in local memory and decrements the enabling count of the successor instruction. If the enabling count remains positive, one or more operand values are still missing, so the updater goes on; if the enabling count becomes zero, then all required operands are present, so the updater resets the enabling count to its initial value and places the address of the instruction on the activity list.

Because concurrently enabled data flow operators are independent, concurrent activity in different pipeline modules is independent and need not be synchronized. A module will not write a message into a communication channel which is full, but this can be determined on a local basis by the module. Stages of instruction execution can be overlapped and the execution of slow instructions need not delay the execution of faster instructions. Activation processors therefore will execute programs very efficiently.

Each processor contains two functional units which lead outside the processor: The call unit sends procedure call requests to the scheduler and gets the results. The structure unit sends structure operation requests to a structure controller. Because these operations are correctly performed by the external modules, the "hole" in the pipeline is invisible to the rest of the processor.

Activity Counter

The activity counter does not affect instruction execution, but rather monitors the processor status for the benefit of the scheduler. The activity count equals the number of active pipeline modules or channels. The count is adjusted as activity is gained or lost. For example, a binary instruction with a single result loses one activity. If the count becomes zero, no pipeline module is active; unless the activation has terminated, it must be waiting for the return of a procedure call. Since an activation can remain dormant for an indefinite time, the scheduler can transfer the contents of a dormant activation's local memory to swap memory and assign the processor to another activation. When the activation becomes active again, it is swapped back into another dormant processor.

Structure Management

Structure Representation

Because local memories are limited in size, large data structures are stored in structure memory and represented within processors by their addresses in structure memory. Although data flow structures are conceptually unshared, structures are represented in memory as acyclic graphs in which common substructures are sometimes shared. However, the structure controllers simulate unshared structures for the processors. Each level of a structure, called a node, is stored as a contiguous vector of components: primitive values are represented directly, while substructures are represented by pointers to other nodes. Each node also has a reference count equal to the number of pointers referencing it in processors and in other nodes. The reference count is adjusted as operations change the number of pointers to a node. If its reference count becomes zero, a node is inaccessible and can be deallocated. Because data flow structures are acyclic, garbage collection is unnecessary.

Structure Controllers

The structure controllers prevent structure operations from causing side-effects. Read-only operations (such as extraction of components and structure length testing) are performed by moving pointers and adjusting reference counts. New structures (such as those formed by joining substructures) are created in free storage of structure memory. Only Alter operations (in which the value of a component is changed to yield a new structure) are affected by concurrent structure node references. If the reference count of a node is one, then the node is accessible only to the Alter operator, so it can be updated in place without causing a side-effect. If the reference count exceeds one, then the structure node is shared, and the node must first be copied before being updated. Note that only the top level of a shared structure need be copied in a single Alter operation.

Structure Cache

Because many structures are accessed sequentially, efficiency of structure operations can be greatly improved by providing each activation processor with a structure cache. Each cache would hold several pages from currently accessible structures. A page would be loaded from structure memory in a single operation. Many sequential structure operations could then be performed for every structure memory access. Because data flow structures are independent, modifications to structure pages could be done immediately in the cache, without need for immediate write-through to structure memory or interlocks among caches, as are required on conventional multiprocessors.

Scheduler

The scheduler creates, maintains, and destroys procedure activations and assigns them to processors for execution. Each activation is assigned a unique identifier, the activation pointer, when it is first created. The scheduler maintains three tables: The call list holds procedure call requests which are waiting to be processed. The processor table identifies the activations currently resident in processors. The activation table identifies the caller and calling location of each activation.

When the scheduler receives a call request from a processor, it appends the caller's activation pointer to the request, which it then places in the call list to wait its turn. When a processor becomes free through the termination or swapping of an activation, the first entry is removed from the call list and assigned an activation pointer. The caller and calling location are stored in the activation table, the program is read into the processor from program memory, and the argument value is fed into the processor. The processor is then free to execute.

When an activation terminates, the processor is deallocated, the name of the caller is retrieved from the activation table, and the processor table is examined to see if the caller is already in a processor. If so, the result is returned to the caller; if not, the caller is brought into the newly vacated processor from swap memory and reactivated.

Peripheral Processor

The peripheral processor is the interface with the outside world. On the inside, it resembles an activation processor. It can access structure memory and make procedure calls. Requests to the peripheral processor cause the creation of root nodes of the activation tree.

Formal Description

In [1] a formal specification of the Data Flow Multiprocessor is given as a set of simple programs which define the machine modules as finite state machines. A proof is given that the machine correctly implements the data flow language described there. The machine is shown to execute well-formed data flow programs without danger of deadlocks or race conditions.

Conclusions

The Data Flow Multiprocessor is simple to understand and modify because it is modular and asynchronous. Because data flow operations are independent and have no side-effects, a high degree of pipelining is possible for efficient hardware utilization. The isolation of procedure activations in separate processors simplifies the processor-memory interconnection problem. The machine should be suitable for LSI implementation: modules are self-contained with few interfaces and the organization is uncomplicated.

Because data flow notation contains no side-effects, the basic machine as described here cannot represent non-determinate computations (such as I/O and interprocess communication). Several methods of extending the model to include non-determinate operations are possible, but further work is needed.

References

[1] James Rumbaugh, A Parallel Asynchronous Computer Architecture for Data Flow Programs, M. I. T. Project MAC, TR-150, (May 1975), 319 pp.

[2] James Rumbaugh, "Data Flow Languages," these proceedings.

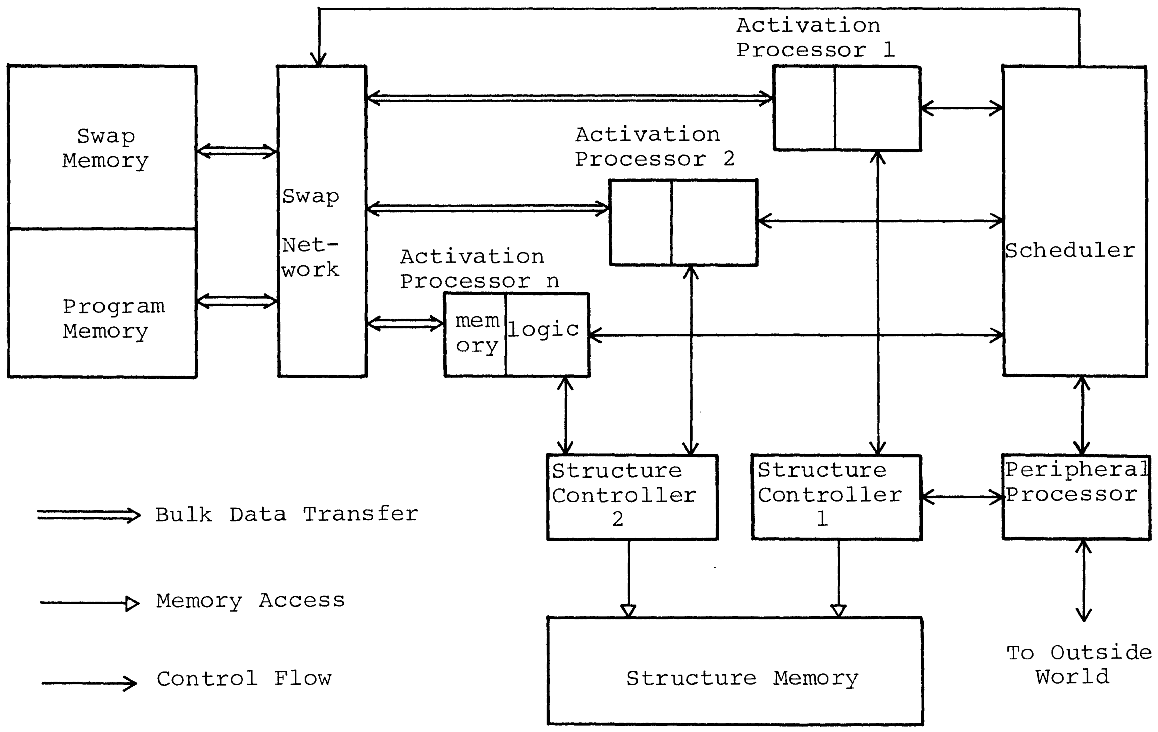


Fig. 1. Structure of the Data Flow Multiprocessor

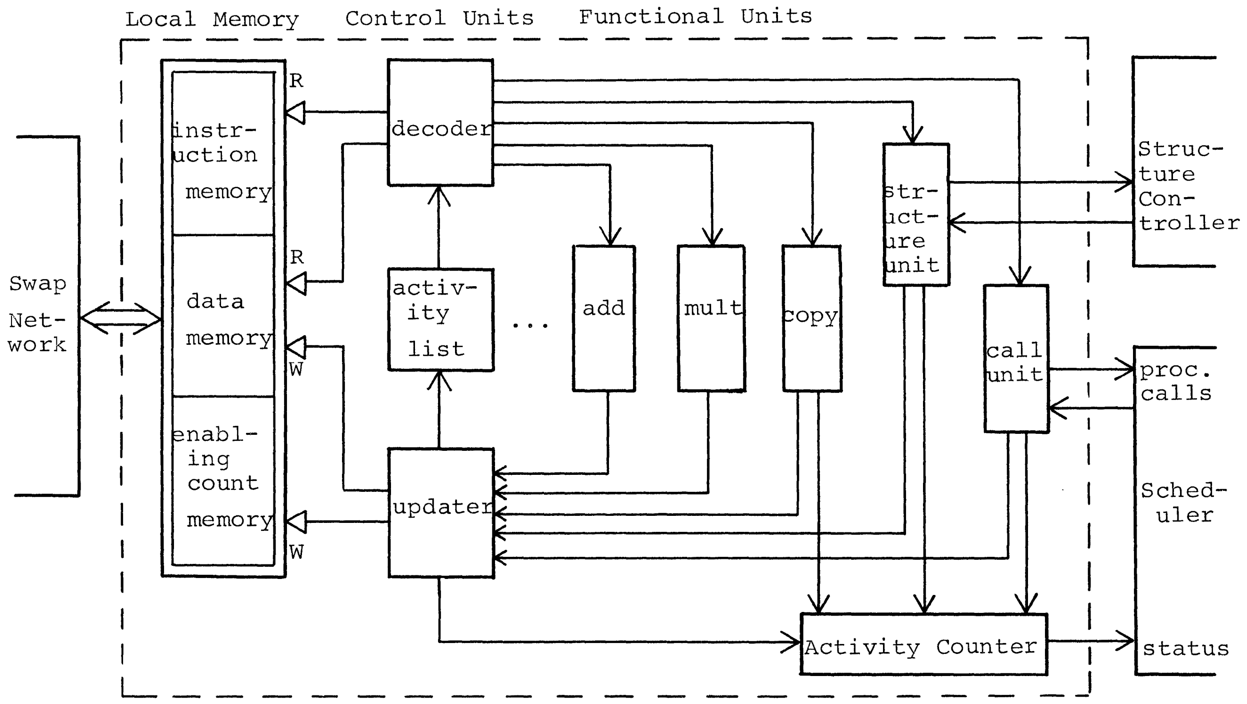


Fig. 2. Activation Processor

PACKET COMMUNICATION ARCHITECTURE

Jack B. Dennis
Project MAC
Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract -- Packet Communication Architecture is the structuring of data processing systems as collections of physical units that communicate only by sending information packets of fixed size, using an asynchronous protocol. Each unit is designed so it never has to wait for a response to a packet it has transmitted to another unit while other packets are waiting for its attention. Packets are routed between sections of a system by networks of units arranged to sort many packets concurrently according to their destination. In this way, it is possible to arrange that system units are heavily used provided concurrency in the task to be performed can be exploited. The packet communication principle is especially attractive for data flow processors since the execution of data flow programs readily separates into many independent computational events. In this paper we show how packet communication can be used in the architecture of memory systems capable of processing many independent memory transactions concurrently and having hierarchical structure. The behavior of these memory systems is prescribed by a formal memory model appropriate to a computer system for data flow programs.

Introduction

With the advent of LSI technology, the main direction of further advance in the power of large computer systems is through exploitation of parallelism. Attempts to achieve parallism in array processors, associative processors and vector or pipeline machines have succeeded only with the sacrifice of programmability. These large parallel machines all require that high levels of local parallelism be expressed in program formats that retain the notion of sequential control flow. Since most algorithms do not naturally exhibit local parallelism in the form expected by these machines, intricate data representations and convoluted algorithms must be designed if the potential of the machine is to be approached.

The alternative is to design machines that can exploit the global parallelism in programs, that is, to take advantage of opportunities to execute unrelated parts of a program concurrently. Conventional sequential machine languages are unsuited to this end because identification of concurrently executable program parts is a task of great difficulty. Data flow program representation are of more interest, for only essential sequencing relationships among computational events are indicated. An instruction in a data flow program is enabled for execution by the arrival of its operand values -- there is no separate notion of control flow, and where there is no data dependence between program parts, the parts are implicitly available for parallel execution.

Several designs for data processing systems have been developed that can achieve highly parallel operation by exploiting the global concurrency of programs represented in data flow form [1-6]. Two of these designs [3 ,6] are able to execute programs expressed in a conventional high-level language that exceeds Algol 60 in generality. These systems consist of units that operate independently and interact only by transmitting information packets over channels that connect pairs of units. The units themselves may have similar structure so the system as a whole has a hierarchical structure that we call <u>packet communication architecture</u>.

In this paper we discuss the principles of packet communication architecture, and illustrate their application to the organization of memory systems for highly concurrent operation.

The Packet Communication Principle

Suppose the data processing part P of the computer in Figure 1 is organized so many independent computational activities may be carried forward concurrently -- as would be true if P contains many independent sequential processors, or if P is designed to exploit the inherent parallelism of data flow programs. Activities in P will generate many independent requests to the memory system M for storage or retrieval of information. It is not essential that M respond immediately to these requests because, if P is properly organized, its resources (registers, instruction decoders, functional units) may be applied to other activities while some activities are held up by pending memory transactions. Thus the memory system need not be designed to complete one transaction before beginning the processing of other transactions. In fact, we will see how this freedom can be exploited in memory systems organized to process many transactions concurrently and keep their constituent units heavily utilized.

This is the first principle of packet communication architecture -- designing each part of a system so many activities are concurrent, and reasonable delays in information transmission between parts may be incurred without important loss of performance. This tolerance of delay permits a radically different approach to the design of switching mechanisms that interconnect sections of a computer system.

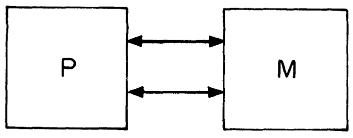


Figure 1. Computer system.

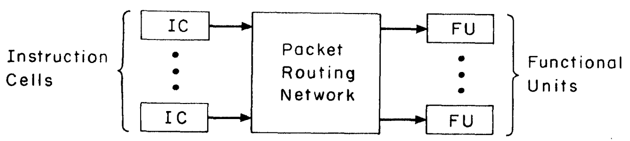


Figure 2. Application of a routing network.

In a system designed to profit from this principle, the response to a request may arrive only after many additional requests have been sent. In general the responses will occur in a different order than the requests raising the problem of how the requesting system part can relate each response to the request that generated it. In many cases it is possible to avoid the problem by including sufficient information in the response that the processing of it is determined without relating it to a specific request. This principle is followed in the several proposed data flow architectures and leads to some elegant data processing structures.

A function frequently required in a computer system is a mechanism to direct requests from one system part to the appropriate specialized unit of the system according to the nature of the request. Two examples are: an instruction and its operands must be sent to an appropriate functional unit for execution; a request by a processing unit for the contents of a specified memory location must be sent to the appropriate memory module. Figure 2 illustrates the former example in the case of a data flow processor [1]. Operands arrive at units called Instruction Cells to form Operation Packets which must be transmitted to the appropriate Functional Units. Some form of switching mechanism must provide a path for Operation Packets between each Instruction Cell and each Functional Unit. Because attaining minimum delay is a less crucial objective in a packet communication system than achieving high concurrency, Routing Networks such as shown in Figure 3 are an attractive form of switching mechanism.

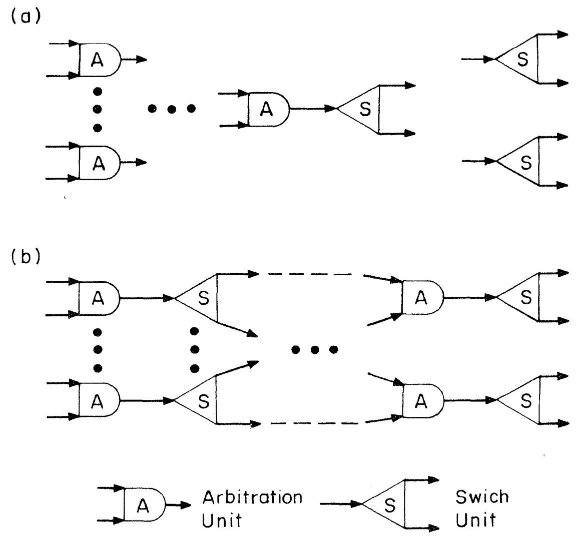


Figure 3. Routing network structure.

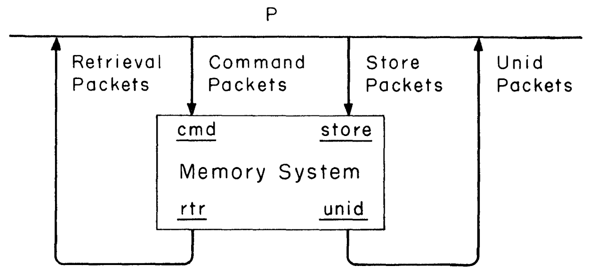


Figure 4. A memory system and its ports.

A routing network is itself a packet communication system built of two basic kinds of modules: Arbitration Units and Switch Units. An Arbitration Unit transmits packets arriving on either input channel to its output channel. A Switch Unit sends arriving packets over the output channel determined by some property of the packet.

If the Arbitration Units and Switch Units are grouped as in Figure 3a, there is a single channel through which all packets must flow -- probably a bottleneck. Alternatively, interleaving ranks of Arbitration Units and Switch Units as indicated in Figure 3b provides for concurrent flow of packets over disjoint paths.

A routing network that has few input ports, and therefore consists mostly of Switch Units is called a Distribution Network, and has the effect of sorting arriving packets according to their contents. In Figure 2 the routing network would sort operation packets according to the operation codes of the instructions they contain. A routing network that has few output ports, and consists mostly of Arbitration Units is called an Arbitration Network.

Packet Communication Memory Systems

As an example of packet communication architecture, consider the memory system shown in Figure 4 which is connected to a processing system P by four channels. Command Packets sent to the memory system at port cmd are requests for memory transactions, and specify the kind of transaction to be performed. Items to be stored are presented as Store Packets at port store, and items retrieved from storage are delivered as Retrieval Packets at port rtr. The role of port unid will be explained later.

For further discussion of the operation of this memory system, we must define the desired behavior -- the nature of the information stored, and how the contents of Retrieval Packets depends on the contents of Store Packets previously sent to the memory system. A precise specification of behavior may take the form of an abstract memory model consisting of a domain of values and a specification of each transaction in terms of the sequences of packets passing the ports of the memory system. We give an informal outline of such a memory model.

For simplicity, the value domain V is

$$V = E + [V \times V]$$

and is the union of pairs consisting of all ordered pairs of elements of V. This domain is recursively defined, and consists of all finite binary trees having elementary values at their leaves.

Our memory model must deal with the retention of information by the memory system. We use a domain of memory states which are acyclic directed graphs called state graphs. Each node of a state graph represents a value (binary tree) in V in the obvious way.

The transactions of this memory model are so specified that no outgoing arc is added or deleted from a node already present in the state graph, and hence the value represented by a node never changes. A memory system having this property is attractive for applicative languages such as pure Lisp and various determinate data flow languages. However, such a memory model is incomplete in that it does not support the updating of a shared data base, for example. The proper way to generalize this memory model is a matter of current research.

The basis of a memory state is a subset of the nodes of a state graph that includes every root node of the graph (Thus each node and arc of a state graph is accessible over a directed path from some basis node). Each basis node represents a value in terms of which the precessing system may request transactions by the memory system.

Each node of a state graph has an associated reference count which is the sum of two numbers--the number of state graph arcs that terminate on the node, and the number of "references" to the node (if it is a basis node) held in the processing system P. Each node of a valid state graph must have a reference count greater than zero.

We regard the memory system as holding a collection of items that represent a state graph in the manner of a linked list structure. To this end we require a set of unique identifiers for the nodes of state graphs. One may regard each unique identifier as corresponding to a unique site in the memory system that can hold a distinct item. The items held by the memory system are of two kinds:

1. Elementary items: ⟨elem, i, e, r⟩
   where i is a unique identifier
   e is an elementary value
   r is a reference count
2. Pair Items: ⟨pair, i, j, k, r⟩
   where i, j, k are unique identifiers
   r is a reference count

Elementary items and pair items correspond to leaf modes and pair nodes, respectively, of a state graph. In each item, i is the unique identifier of the item.

For the purpose of specifying the transactions of the memory system, it is convenient to suppose that it has the structure shown in Figure 5. Command Packets delivered at port rc (for reference count) of M are merged with Command Packets from P and presented to M at port cmd. We specify the behavior of the whole memory system by specifying the behavior of M. We regard the state of M as consisting of a collection of items and a collection of unique identifiers not in use. In the initial state of M the collection of items is empty and every unique identifier is not in use.

The specifications for the behavior of M state the response, if any, and change of state,

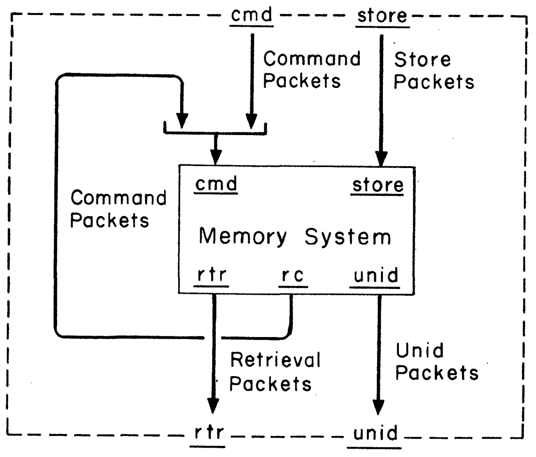


Figure 5. Structure of a memory system for specification of its transactions.

if any, that accompany each kind of transaction. In the simple memory system we are considering, there are five kinds of transactions -- four of these are associated with acceptance of Command Packets by M, and the fifth is associated with delivery of Unid Packets. The behavior of M for each kind of transaction is as follows.

Store Transaction:

$$\begin{tikzcd} \langle \underline{store}, i\rangle \arrow[d] & \left\{\begin{matrix}\langle \underline{elem}, i, e\rangle \\ \langle \underline{pair}, i, j, k\rangle\end{matrix}\right\} \arrow[d] \\ \underline{cmd} & \underline{store} \end{tikzcd}$$

In response to a store Command Packet, the item presented at port store is added to the collection of items held by M, and is given an initial reference count of one.

Retrieval Transaction:

$$\begin{tikzcd} \langle \underline{retr}, i\rangle \arrow[d] & \\ \underline{cmd} \arrow[r, dash, dashed] & \underline{rtr} \arrow[d] \\ & \left\{\begin{matrix}\langle \underline{elem}, i, e\rangle \\ \langle \underline{pair}, i,j, k\rangle\end{matrix}\right\} \end{tikzcd}$$

The item dilivered at port rtr is the item with unique identifier i in the collection of items held by M. The state of M does not change.

Reference Generation and Anihilation

$$\begin{tikzcd} (a)\langle \underline{up}\ i\rangle \arrow[d] & (b)\langle \underline{down}\ i\rangle \arrow[d] & (c)\langle \underline{down}\ i\rangle \arrow[d] & \\ \underline{cmd} & \underline{cmd} & \underline{cmd} \arrow[r, dash, dashed] & \underline{rc} \arrow[d] \\ & & & \begin{matrix}\langle \underline{down}\ i\rangle \\ \langle \underline{down}\ k\rangle\end{matrix} \end{tikzcd}$$

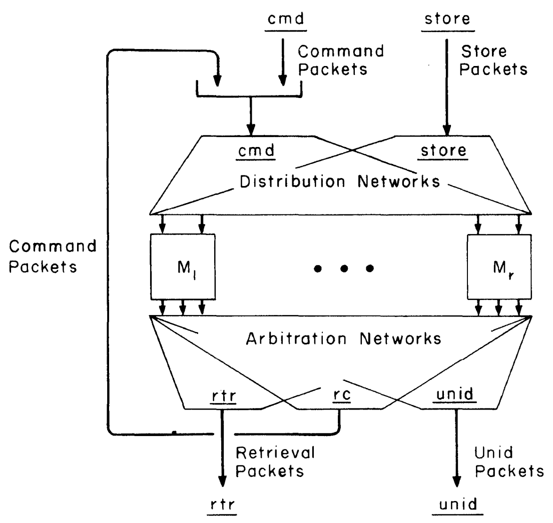


Figure 6. Memory system structure for concurrency.

The up command adds one to the reference count of item i; the down command decrements its reference count by one. If the reference count is reduced to zero by a down command, the item is deleted from the collection of items held by M and its unique identifier i is added to the collection of unused unique identifiers. Case (c) applies if the item deleted is a pair item since the reference counts of its component items must be decremented.

Unique Identifier Generation:

$$\begin{tikzcd} \underline{\text{unid}} \\ \langle i \rangle \arrow[u] \end{tikzcd}$$

Some unique identifier i is removed from the set of unused unique identifiers and delivered at port unid.

We have not specified the behavior of M under certain conditions that should not occur during normal operation -- for example, if a store Command Packet contains a unique identifier which is already the unique identifier of an item held by M. We assume the processing system is so designed that such ill behavior cannot occur. A discussion of these restrictions on processor behavior and how they might be implemented is beyond the scope of the present paper.

Memory System Structure

With this albeit informal specification of M, we are prepared to see how M may be realized by simpler subsystems interconnected as a packet communication system.

First we show how concurrent processing of many transactions can be achieved by distributing

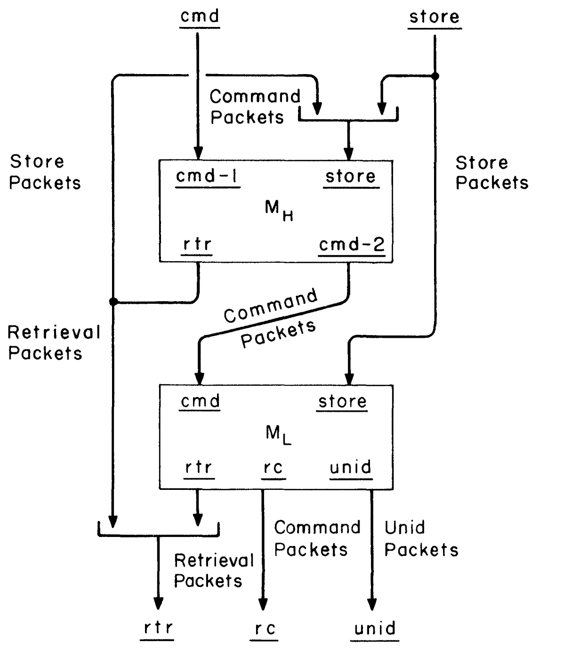


Figure 7. Hierarchical packet memory system.

Command Packets among many identical physical modules which can operate independently. Such a structure for M is shown in Figure 6. Each Command Packet and each Store Packet is distributed to one of the memory subsystems $M_1, \ldots, M_r$ according to some easily tested property of i, the unique identifier of the item to which the packet refers. The property might be the first p bits of the binary representation of the unique identifier where $r = 2^p$.

The subsystems $M_1, \ldots, M_r$ are memory systems having specifications identical to the specification of M except that the universe of unique identifiers for the items held by each subsystem is restricted to $(1/2)^p$ of the unique identifiers of M. This fact may be used to reduce the complexity of the memory subsystems.

The Retrieval Packets, Command Packets, and Unid Packets delivered by the memory subsystems at their rtr, rc, and unid ports are merged into common streams by three Arbitration Networks. Note that the Command Packets from subsystem rc ports must be recirculated through the Distribution Networks because, in general, the items they refer to will be held in subsystems other than the subsystem from which they originate.

Hierarchical Structure

Any realistic design for a large memory system must recognize the principle that only the most active information need be held in expensive fast-access devices; less active information should be held in slower devices. If the memory system is to support modularity of programming in its most general form, then information must be automatically redistributed among levels of the memory system as computational activity involves different portions of the stored information.

Figure 7 shows how a memory system M satisfying our specification may be realized by a hierarchical organization of two memory systems M_H and M_L. With an important exception explained later, the lower level subsystem M_L satisfies the same behavioral specification as the entire memory system M. The higher level subsystem M_H is arranged to hold copies of the most active items in M_L -- it acts as a cache memory so M is able to achieve a much lower latency in processing transactions than M_L could alone. Keep in mind that even though M_L may have a long latency, it may have a high rate of processing transactions due to its organization for highly parallel operation.

If there is no room in M_H for an item sent to M for storage, or for an item retrieved form M_L, then some item is selected for deletion from M_H. The criterion for selecting the item to be deleted could be any of the schemes used in contemporary cache memories or paging systems. The deleted item need not be sent to M_L because the memory system under discussion is organized so M_L holds a copy of every item present in the memory system M. However, M_L must know which items it holds have duplicates in M_H so it can tell whether it is safe to release the unique identifiers of deleted items for reuse. Hence each item in M_L includes an indicator f that tells whether the item is also held in M_H:

<u>Elementary</u> <u>Items</u>:	⟨<u>elem</u>, i,e,r,f⟩
<u>Pair</u> <u>Items</u>:	⟨<u>pair</u>, i,j,k,r,f⟩

where f is one of {<u>true</u>, <u>false</u>}

The tansactions of M_L have specifications just like those for the transactions of M, except for a few changes:

<u>Store</u> <u>Tansactions</u>: The indicator f of the item added to the collection is <u>true</u> since each Store Packet is sent to both M_H and M_L.

<u>Anihilation</u>: If the reference count of an item is reduced to zero, the item is deleted and its unique identifier released for reuse only if the indicator f is <u>false</u>.

<u>Done</u> <u>Transaction</u>: An additional form of Command Packet ⟨<u>done</u>, i⟩ is sent by M_H to M_L to say that item i has been deleted from M_H. Subsystem M_L responds by setting f to <u>false</u> and, if the reference count is zero, the item is deleted from M_L and its unique identifier is released for reuse.

In M_H items are held without reference counts:

<u>Elementary</u> <u>Items</u>:	⟨<u>elem</u>, i,e⟩
<u>Pair</u> <u>Items</u>:	⟨<u>pair</u>, i,j,k⟩

For the purpose of specifying the behavior of M_H, its state is simply a collection of items in these formats. A realization of M_H would require additional state information to implement the chosen criterion for deletion of items. Item removal is a routine used by store transactions of M_H:

<u>Item</u> <u>Removal</u>:

$$\begin{tikzcd}
\langle \underline{\text{done}}, i\rangle \arrow[d] \\
\underline{\text{cmd-2}}
\end{tikzcd}$$

The item to be removed is deleted from the collection of items held by M_H and a done Command Packet is sent at port <u>cmd-2</u>.

The transactions of M_H are:

<u>Store</u> <u>Transaction</u>:

$$\begin{tikzcd}
\langle \underline{\text{store}}, i\rangle \arrow[dd] & & & \begin{array}{c}\langle \underline{\text{elem}}, i,e\rangle \\ \langle \underline{\text{pair}}, i,j,k\rangle\end{array} \arrow[ddd] \\
& & \underline{\text{cmd-2}} \arrow[dd] & \\
\underline{\text{cmd-1}} \arrow[rru, dash, dashed] & & & \\
& & \langle \underline{\text{store}}, i\rangle & \underline{\text{store}}
\end{tikzcd}$$

The item is added to the collection after some chosen item is removed, if necessary, to create space. The store Command Packet is forwarded to M_L.

<u>Retrieval</u> <u>Transaction</u>:

$$\begin{tikzcd}
(a)\langle \underline{\text{retr}}, i\rangle \arrow[d] & \underline{\text{rtr}} \arrow[d] & & (b)\langle \underline{\text{retr}}, i\rangle \arrow[d] & \underline{\text{cmd-2}} \arrow[d] \\
\underline{\text{cmd-1}} \arrow[ru, dash, dashed] & \begin{array}{c}\langle \underline{\text{elem}}, i,e\rangle \\ \langle \underline{\text{pair}}, i,j,k\rangle\end{array} & & \underline{\text{cmd-1}} \arrow[ru, dash, dashed] & \langle \underline{\text{retr}}, i\rangle
\end{tikzcd}$$

Case (a) applies if an item with unique identifier i is in the collection held by M_H; otherwise case (b) applies, and the retrieval Command Packet is forwarded to M_L.

<u>Reference</u> <u>Accounting</u>:

$$\begin{tikzcd}
(a)\langle \underline{\text{up}}, i\rangle \arrow[d] & \underline{\text{cmd-2}} \arrow[d] & & (b)\langle \underline{\text{down}}, i\rangle \arrow[d] & \underline{\text{cmd-2}} \arrow[d] \\
\underline{\text{cmd-1}} \arrow[ru, dash, dashed] & \langle \underline{\text{up}}, i\rangle & & \underline{\text{cmd-1}} \arrow[ru, dash, dashed] & \langle \underline{\text{down}}, i\rangle
\end{tikzcd}$$

Up and down Command Packets are forwarded to M_L without action by M_H.

In addition, for correct operation of the whole memory system M, all Command Packets relating to item i must be delivered at port <u>cmd-2</u> in the same order as they arrive at port <u>cmd-1</u>.

Conclusion

Packet Communication architecture offers many attractions to the computer system designer: The units of a system interact through very simple interfaces and are easy to specify; timing hazards are eliminated through use of a strict speed independent communication discipline; and the principles are applicable to the organization of machines that support the execution of well structured programs expressed in high level languages. A high level of concurrent operation can be achieved with high equipment unilization if the global parallelism inherent in most data processing tasks is expressed in the program.

It will be interesting to see if these attractions can be realized in the design of practical computer systems.

Acknowledgment

This research was supported in part by the National Science Foundation under grant DCR75-04060 and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0661.

Bibliography

1. Dennis, J. B., and D. P. Misunas, "A computer architecture for highly parallel signal processing," Proceedings of the ACM 1974 National Conference, ACM, New York (November, 1974), 402-409.

2. Dennis, J. B., and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York (January 1975), 126-132.

3. Misunas, D. P. A Computer Architecture for Data-Flow Computation. SM Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass. (June 1975).

4. Misunas, D. P., "Structure processing in a data-flow computer," Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York (August 1975).

5. Project MAC Progress Report XI, Project MAC, M.I.T. (July 1973-74), pp. 84-90.

6. Rumbaugh, J. E., A Parallel Asynchronous Computer Architecture for Data Flow Programs, Project MAC, M.I.T., Cambridge, Mass., Report TR-150 (May 1975).

STRUCTURE PROCESSING IN A DATA-FLOW COMPUTER*

David P. Misunas
Project MAC
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- A data-flow computer uses a packet communication system to achieve highly parallel execution of programs expressed in data-flow form. The machine is composed of two sections which perform instruction processing and structure processing and share a common auxiliary memory. The structure processing section of the processor maintains data structures represented as acyclic directed graphs and is viewed as a functional unit by the instruction processing section; that is, instructions specifying structure operations are sent to the section, and the resulting values are returned to the instruction processing section. The organization of the structure processing section as a packet communication system permits the simultaneous processing of many structure operations, while avoiding the deadlock and synchronization problems often associated with systems that support concurrent memory transactions.

Introduction

The data-flow form of program representation has been developed as a method of expressing parallel activity [1, 2, 3, 5, 8, 9, 11]. The attractiveness of this form of representation lies in the fact that it is data-driven; that is, an instruction is enabled for execution when each required operand has been provided by the execution of a predecessor instruction.

The simplicity of this method of representation has led to the development of a number of computer architectures capable of executing programs expressed in data-flow form. Elementary forms of the data-flow language are utilized as the base language of a series of machines developed by Dennis and Misunas [6, 7]. The implementation of more complete data-flow languages, incorporating data structures and procedures, has been investigated by Misunas [10] and Rumbaugh [12, 13].

In the machines described by Dennis and Misunas [6, 7], the processing of instructions of a program is carried out in an instruction processing section which is structured as a <u>packet communication system</u> [4]. Sections of a machine are connected by interconnection networks which have a great deal of inherent parallelism, and the secions communicate by means of fixed size information packets. Each section is designed so that it never has to wait for a response to a packet it has transmitted if other packets are waiting for its attention. The extension of this concept to the organization of the structure processing section of a computer, described herein, proves very attractive, eliminating many of the deadlock and synchronization problems currently associated with systems that support concurrent memory transactions.

Data-Flow Structure Values

A program expressed in the data-flow language is constructed of two kinds of elements, called <u>actors</u> and <u>links</u>. An actor has a number of input arcs which supply values necessary for its execution and one output arc upon which results are placed. A small dot represents a link which has one input arc upon which it receives results from an operator and a number of output arcs over which it distributes copies of the results to other actors.

Values are conveyed over the arcs of a program by <u>tokens</u>, represented as large solid dots. An actor with a token on each of its input arcs, and no token on its output arc, is <u>enabled</u> and sometime later will <u>fire</u>, removing the tokens from its input arcs, computing a result using the values carried by the input tokens, and associating the result with a token placed on its output arc. In a similar manner, a link is enabled when a token is present on its input arc, and no token is present on any of its output arcs. It fires by removing the token from its input arc and associating copies of the value carried by the input token with tokens placed on its output arcs.

A value conveyed by a token is either an <u>elementary value</u> or a <u>structure value</u>. An elementary value is a single integer, real, string, or Boolean value. A structure value in a data-flow program is composed of a number of elementary values and is represented as an acyclic directed graph having one <u>root node</u> with the property that each node of the graph can be reached by a directed path from the root node. A node of the graph is either a structure node or an elementary node. A structure node serves as the root node for a substructure of the structure and represents a value which is a set of selector-value pairs

$$\{(s_1, v_1), \ldots, (s_n, v_n)\}$$

where

$$s_i \in \{\text{integers}\} \cup \{\text{strings}\}$$
$$v_i \in \{\text{elementary values}\} \cup \{\text{structure values}\} \cup \{\underline{\text{nil}}\}$$

and s_i is the selector of node v_i. An elementary node has no emanating arcs; rather, an elementary value is associated with the node. A node with no emanating arcs and no associated elementary value has value $\{\underline{\text{nil}}\}$.

*This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

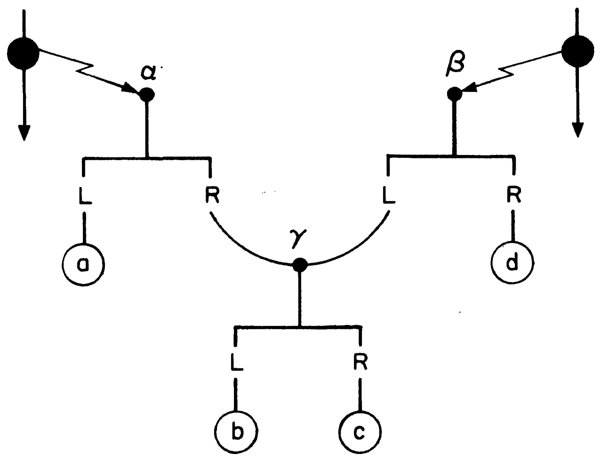


Figure 1. An example of two structures sharing a common substructure.

To illustrate the operation of the structure processing section of the processor, we shall limit our consideration to structures represented as binary trees. A selector of such a structure can have one of two values, L (left) and R (right), designating the left and right branches of the tree.

A structure value is represented by a data token carrying a pointer to the root node of the structure. In Figure 1 the structure α contains three elementary values a, b, and c, designated by the simple selector L and the compound selectors R·L and R·R respectively. Structure node γ of structure α is shared with structure β and is designated by a different selector in β than in α.

The data-flow program of Figure 2 transposes the elements of the four-element structure presented on its input. Initially, the input link of the program is enabled and, upon firing, creates four copies of the token conveying a pointer to structure α and places the copies on the inputs of the four select actors. Each select actor retrieves the value (either an elementary value or a structure value) at the end of the path specified as its argument. The resulting value is associated with a token placed on the output arc of the actor.

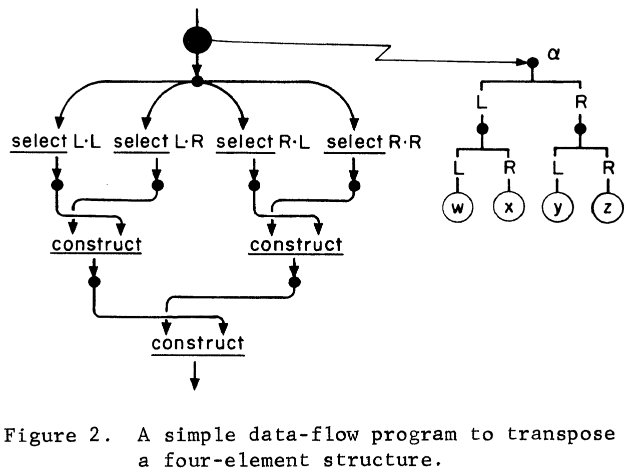


Figure 2. A simple data-flow program to transpose a four-element structure.

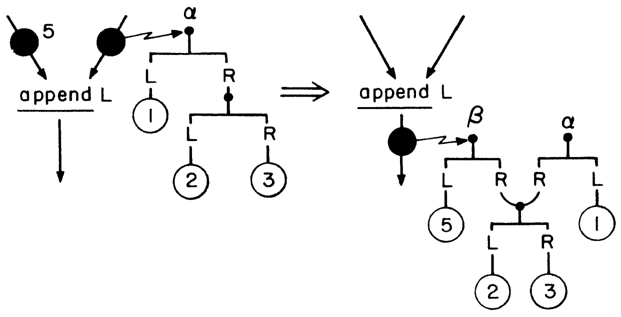


Figure 3. Operation of the append actor.

Each construct actor is enabled when it has a token on each input arc and, upon firing, creates a new structure of the values associated with the input tokens. In the program of Figure 2, the position of each input indicates the selector to be associated with the input in the resulting structure.

Structure values in a data-flow program are not modified; rather, new structure values are created which are modifications of the original values, while the original values are preserved. The append and delete actors provide the means of creating these new structure values.

The structure produced by the firing of an append actor is a version of the input structure which contains a new or modified component (Figure 3). If the specified node of the input structure has a selector corresponding to the selector argument of the actor, the value designated by that selector in the new structure is the input value. Otherwise the specified selector-value pair is added to the node of the new structure. Identical elements of the input and output structures are shared between the two structures.

In a similar manner, the structure appearing on the output arc of a delete actor is a version of the input structure in which the specified node in the new structure is missing the selector-value pair designated by the selector argument (Figure 4). As with the append actor, identical elements are shared between the input and output structures.

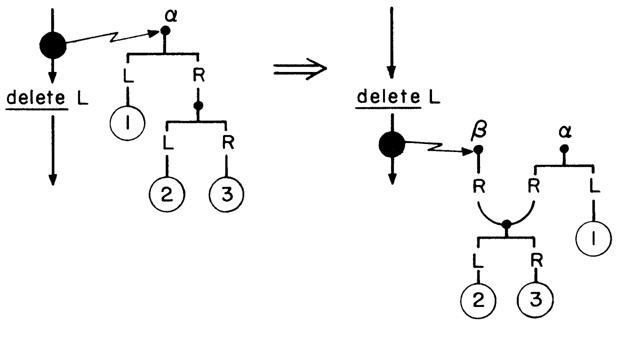


Figure 4. Operation of the delete actor.

Structure Representation

The storage of structures and the execution of instructions representing structure actors occurs in the structure processing section of a data-flow processor. The structure processing section consists of a Structure Operation Unit and a Structure Memory and attendant Arbitration and Distribution Networks. This section of the processor is viewed as a functional unit by the instruction processing section; that is, operation packets specifying structure operations are sent to the section, and data packets are returned. The organization of the structure processing section is shown in Figure 5.

Operation packets containing instructions representing structure actors are transmitted to the Structure Operation Unit by the instruction processing section. The Structure Operation Unit controls the execution of the instruction specified in each operation packet through instruction packets sent to the Structure Memory. The Structure Memory holds all structure values of the data-flow program, and all structure operations are performed in the Memory. Upon completion of a structure operation, the Structure Memory transmits a data packet containing the resulting elementary or structure value to the instruction processing section.

A node of a structure is contained in a two register Cell known as a Structure Cell and designated by a Cell identifier. The two registers of the Cell contain the left and right components of the structure, respectively; and hence no selector need be stored in a register. The first field of a register is a <u>use code</u> which indicates whether the item stored in the second field is the identifier of another Cell or an elementary value, or if the register is empty. A memory representation of the simple structure of Figure 1 is given in Figure 6.

The Structure Memory is composed of a number of Structure Cells. Each Structure Cell is capable of holding one node of a structure, and the identifier of the Cell specifies a path through the Distribution Network to the Cell. The Structure Memory receives instruction packets from the Structure Operation Unit commanding a specific Structure Cell to execute some structure operation upon the node located in the Cell. Upon completion of the operation specified in an instruction packet, a Structure Cell presents any result as a data packet to the Arbitration Network for conveyance to the instruction processing section. Any further structure operations are specified in instruction packets returned to the input of the Structure Memory.

A Structure Cell within the Structure Memory performs one of three operations upon the structure node contained in the Cell. The possible operations are:

1. <u>select</u>. Upon receipt of an instruction packet specifying a select operation

$$\left\{\begin{matrix}\underline{\text{select}}\ \text{dest}\\ s\end{matrix}\right\}$$

a Structure Cell follows one of two procedures, controlled by whether the selector s is a simple selector or a compound selector.

a. If s is a simple selector, L or R, the content c of the Cell register designated by s is used to form a data packet

$$\left\{\begin{matrix}\text{dest}\\ c\end{matrix}\right\}$$

which is presented to the Arbitration Network for transmission to the specified destination dest in the instruction processing section of the processor.

b. If s is a compound selector $s_1 s_2 \ldots s_n$, $s_i \in \{L, R\}$, the content β of the register designated by s_1 is the identifier of a Structure Cell and is used to form the instruction packet

$$\left\{\begin{matrix}\beta\\ \underline{\text{select}}\ \text{dest}\\ s_2 \ldots s_n\end{matrix}\right\}$$

which is presented to the Arbitration Network for transmission to the input Distribution Network of the Structure Memory. The process is then repeated with the selector s_2 at Structure Cell β.

2. <u>alter</u>. The receipt of an alter instruction

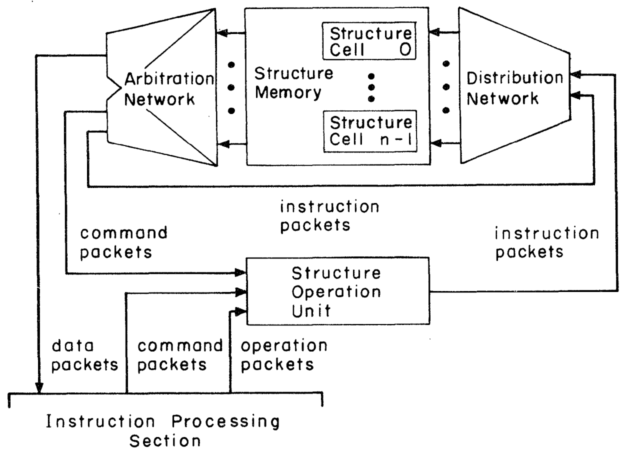


Figure 5. Organization of the structure processing section of the data-flow processor.

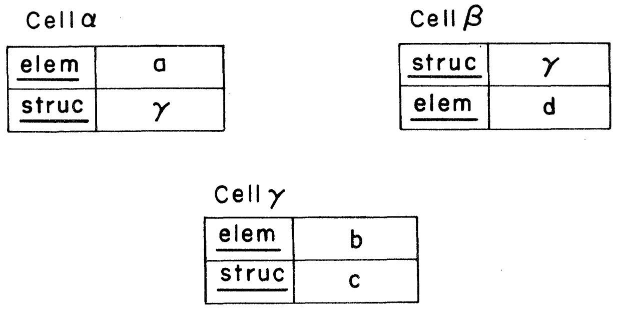


Figure 6. Memory representation of the structure of Figure 1.

$$\left\{\begin{array}{c}\underline{\text{alter}}\ \text{dest}\\ s\\ x\end{array}\right\}$$

indicates that the contents of the Structure Cell are to be modified so the component designated by the selector s is set to x. Since structure values are not modified, a Structure Cell that receives an alter instruction, must receive two alter instructions, one for each register. When both have been received, a data packet containing the Cell identifier β is returned to the instruction processing section:

$$\left\{\begin{array}{c}\text{dest}\\ \beta\end{array}\right\}$$

3. copy. A copy instruction

$$\left\{\begin{array}{c}\underline{\text{copy}}\ \beta\\ \text{dest}\\ s\end{array}\right\}$$

specifies that the content of the register designated by s is to be transmitted to Structure Cell β. An instruction packet

$$\left\{\begin{array}{c}\beta\\ \underline{\text{alter}}\ \text{dest}\\ s\\ c\end{array}\right\}$$

is formed of the register content c and is presented to the Arbitration Network for transmission to the input Distribution Network.

Instructions are transmitted to the Structure Memory as instruction packets, each consisting of a Cell identifier and an instruction. The Cell identifier specifies a path through the Distribution Network to the Cell, and the packet received by a Cell consists of merely the instruction portion of the instruction packet.

The Structure Operation Unit maintains the reference count of each node in the Structure Memory, specifying the number of arcs terminating on the node and the number of references to the node existing in the instruction processing section of the processor. When a node becomes inaccessible due to the execution of some instruction of the program, the reference count of the node becomes zero, and the node is placed on a free node list which is used for the allocation of new nodes during program execution.

The processing of all structure operation packets by the Structure Operation Unit permits the unit to properly decrement reference counts as references to items are deleted through instruction exeuction. References to items are created in the Structure Memory by execution of a select instruction if the selected item is a structure value and in the instruction processing section through execution of an instruction representing a link of a data-flow program. We must require that in either case, command packets of the form

$$\left\{\begin{array}{c}\text{node identifier}\\ \underline{\text{up}}\end{array}\right\}$$

are sent to the Structure Operation Unit, causing the reference count of the designated node to be properly incremented.

Now that we have considered the operation of a Structure Cell within the Structure Memory, we can describe the execution of each of the structure actors merely by listing the procedure followed by the Structure Operation Unit in processing the instruction. For the purposes of this discussion, it is assumed that all selectors are simple selectors.

The processing of a select instruction by the Structure Operation Unit merely causes the reference count of the designated node to be decremented. The content of the operation packet is then sent as an instruction packet to the specified node of the Structure Memory for execution of the select operation.

A construct instruction

$$\left\{\begin{array}{c}\underline{\text{construct}}\ \text{dest}\\ \text{L:}\ \alpha\\ \text{R:}\ \gamma\end{array}\right\}$$

specifies that a new node is to be created with components α and γ, designated by the selectors L and R. The instruction is implemented by the Structure Operation Unit as two alter operations in the following manner:

1. Accept an identifier β from the free node list.
2. Transmit to the Structure Memory the instruction packets

$$\left\{\begin{array}{c}\beta\\ \underline{\text{alter}}\ \text{dest}\\ L\\ \alpha\end{array}\right\} \quad \text{and} \quad \left\{\begin{array}{c}\beta\\ \underline{\text{alter}}\ \text{dest}\\ R\\ \gamma\end{array}\right\}$$

transferring the values α and γ to the correct registers of β.

An operation packet containing an append instruction is of the following format

$$\left\{\begin{array}{c}\underline{\text{append}}\ \text{dest}\\ \text{L:}\ \alpha\\ x\end{array}\right\}$$

where L is the selector of the element in Structure Cell α which is to be replaced by x in the new structure. The procedure followed by the Structure Operation Unit in execution of the instruction is as follows:

1. Accept an identifier β from the free node list.
2. Transmit the instruction packets

$$\left\{\begin{array}{c}\alpha\\ \underline{\text{copy}}\ \beta\\ \text{dest}\\ R\end{array}\right\} \quad \text{and} \quad \left\{\begin{array}{c}\beta\\ \underline{\text{alter}}\ \text{dest}\\ L\\ x\end{array}\right\}$$

to the Structure Memory to copy the register of node α designated by the selector R into Cell β and set the L-component of β to x.

An operation packet specifying a delete instruction is processed in a similar manner, causing the use code of the designated register to be set to empty.

To assure maximum use of the Structure Cells of the processor, the structure processing section utilizes a multi-level memory, so that only active structure nodes occupy the Structure Cells. The

Structure Memory acts as a cache for structure nodes; individual nodes are retrieved from the auxiliary memory as they become required for computation, and structure nodes are sent to the auxiliary memory upon creation through execution of an append, delete, or construct instruction. The structure of the auxiliary memory as a packet communication system is described by Dennis [4], and its use in conjunction with the structure processing section is presented in [10].

Conclusion

The described techniques for the implementation of data structures can be readily extended to larger and more complex structures. In order to implement structures with a fixed maximum number of arcs emanating from each node, the size of a Structure Cell is increased to accommodate the new node size. The use of arbitrary (to a fixed maximum size) integers or character strings as selectors is accommodated through the addition of a selector field to each register. A Structure Cell must then have the capability to choose from the node contained in the Cell an item whose selector matches a specified selector. These extensions allow the representation of a wide variety of structures, including the programs of the data-flow language [10].

References

1. Adams, D. A. A Computation Model With Data Flow Sequencing. Technical Report CS 117, Computer Science Department, School of Humanities and Sciences, Stanford University, Stanford, Calif., December 1968.
2. Bähr s, A. Operation patterns (An extensible model of an extensible language). Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972 (preprint).
3. Dennis, J. B. First version of a data flow procedure language. Lecture Notes in Computer Science 19 (G. Goos and J. Hartmanis, Eds.), Springer-Verlag, New York, 1974, 362-376.
4. Dennis, J. B. Packet communication architecture. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, August 1975.
5. Dennis, J. B., and J. B. Fosseen. Introduction to Data Flow Schemas. November 1973 (submitted for publication).
6. Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM 1974 National Conference, ACM, New York, November 1974, 402-409.
7. Dennis, J. B., and D. P. Misunas. A preliminary architecture for a basic data-flow processor. Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, January 1975, 126-132.
8. Karp, R. M., and R. E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM Journal of Applied Mathematics 14 (November 1966), 1390-1411.
9. Kosinski, P. R. A data flow language for operating systems programming. Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, 9 (September 1973), 89-94.
10. Misunas, D. P. A Computer Architecture for Data-Flow Computation. SM Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., June 1975.
11. Rodriguez, J. E. A Graph Model for Parallel Computation. Report TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969.
12. Rumbaugh, J. E. A Parallel Asynchronous Computer Architecture for Data Flow Programs. Report TR-150, Project MAC, M.I.T., Cambridge, Mass., May 1975.
13. Rumbaugh, J. E. A data flow multiprocessor. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, August 1975.

THE STABILITY OF A PARALLEL ALGORITHM FOR THE SOLUTION OF TRIDIAGONAL LINEAR SYSTEMS

Martin A. Diamond
Systems & Information Science Department
Vanderbilt University
Nashville, Tennessee, 32735

Summary

A parallel algorithm for the solution of tridiagonal linear systems of equations is given by Stone [1]. The algorithm factors the matrix of coefficients into a lower bidiagonal matrix and an upper bidiagonal matrix, and then solves the factored system. If the original system is $Mx = y$ where

$$M = \begin{pmatrix} d_1 f_1 & & & \\ e_2 d_2 f_2 & & & \\ & \cdot\ \cdot\ \cdot & & \\ & & e_{n-1} d_{n-1} f_{n-1} & \\ & & & e_n d_n \end{pmatrix}$$

then the factors of M are L and U where

$$L = \begin{pmatrix} 1 & & & & \\ m_2 1 & & & & \\ & m_3 1 & & & \\ & & \cdot\ \cdot\ \cdot & & \\ & & & m_{n-1} 1 & \\ & & & & m_n 1 \end{pmatrix}; \quad U = \begin{pmatrix} u_1 f_1 & & & & \\ & u_2 f_2 & & & \\ & & u_3 f_3 & & \\ & & & \cdot\ \cdot\ \cdot & \\ & & & u_{n-1} f_{n-1} & \\ & & & & u_n \end{pmatrix}.$$

The elements m_i of L and u_i of U are given by

$u_1 = d_1$, $u_i = d_i - (e_1 f_{i-1}/u_{i-1})$, $i > 1$;

$m_i = e_1/u_{i-1}$, $i \geq 2$.

Instead of computing these values directly, the u_i are computed as the quotients $u_i = q_i/q_{i-1}$ where $q_0 = 1$, $q_1 = d_1$, and $q_i = d_i q_{i-1} - (e_i f_{i-1} q_{i-2})$, $i \geq 2$. As Stone points out immediately below equation (16) of his paper, it is seen that

$$q_i = \prod_{j=1}^{i} u_j.$$

But this implies an instability for most systems. If the u_i satisfy $|u_i| > 1 + \varepsilon$ or $|u_i| < 1 + \varepsilon$ for some ε, then the q_i go to infinity or go to zero. Consider the system with $d_i \equiv 2$, $e_i \equiv 1.5$, and $f_i \equiv 0.5$. In this case, $u_i > 1.5$ for all i. Thus if n is 500, q_n is too large to be stored in an IBM/360 single REAL word.

If the e_i, d_i, and f_i go to limits, as they do in the above example, the instability can be remedied. This is done by finding in one step, the limit, u, of u_i and then in a single parallel operation scaling the coefficients e_i, d_i, and f_i. A different set of numbers $\hat{q}_i$ are then computed. These numbers remain bounded and are such that $\hat{q}_i/\hat{q}_{i-1} = u_i/u$. Thus, $u_i = (\hat{q}_i/\hat{q}_{i-1})u$ can be computed in a single operation.

Reference

[1] H. Stone, "An Effecient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", Journal of the ACM, Vol. 20, #1 (January 1973), pp. 27-38.

PARALLEL ALGORITHMS FOR EVALUATION OF ARITHMETIC EXPRESSIONS*

Ashoke Deb and Amar Mukhopadhyay
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242

Summary

An optimal algorithm for parallel processing of arithmetic expressions is presented in this paper. The arithmetic expression consists of a finite well-formed string of parentheses, operands and operators having a fixed precedence relation on them and having different execution times. A proof of the optimality is presented by providing an analysis which gives the best lower bound for the time of computation. The algorithm not only detects the parallelism but provides an exact scheduling of the processors for implementing it. Other advantages of the algorithm are: one-pass compilation, no conversion to polish string required, independent of initial form of the expression, etc.

We then investigate the effects of the application of distribution laws to override the precedence relation as presented in the expression. We develop several lemmas and theorems pointing out suitable strategies for the application of distribution laws for the purpose of reducing the total execution time. We then present an efficient algorithm to reduce the execution time of an arithmetic expression by the application of distribution laws.

References

[1] J.S. Squire, "A translation algorithm for a multiprocessor computer," Proc. 18th ACM Natl. Conf. (1963).

[2] H. Hellerman, "Parallel processing of algebraic expressions," IEEE Trans. Electronic Computers, vol. 15, no. 1 (1966).

[3] H.S. Stone, "One pass compilation of arithmetic expressions for a parallel processor," Comm. ACM, vol. 10, no. 4 (April 1967), pp. 220-223.

[4] J.L. Baer and D.P. Bovet, "Compilation of arithmetic expression for parallel computation," Proc. IFIP 68 (1968), pp. B4-B10.

[5] C.V. Ramamoorthy and M.J. Gonzalez, "A survey of techniques for recognizing parallel processable streams in computer programs," Proc. FJCC (1969).

[6] F. Mavaddat, "Using stacks to detect expression parallelism," Proc. IFIP (August 1971).

[7] D.J. Kuck, Y. Muraoka and S.C. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," IEEE Trans. Computers, vol. C-21, no. 12 (December 1972), pp. 1293-1310.

[8] A. Deb and A. Mukhopadhyay, Optimal Parallel Algorithm for evaluation of Arithmetic Expressions, Dept. of Computer Science, Univ. of Iowa Tech. Rept. No. 73-10 (Nov. 1973).

[9] A. Deb, Exploitation Laws for Parallel Evaluation of Arithmetic Expression, Dept. of Computer Science, Univ. of Iowa Tech. Rept. No. 74-06 (August 1974).

*The work has been supported by NSF grant GJ-723.

ON PARALLEL TRIANGULAR SYSTEM SOLVERS*

S. Chen** and A. Sameh**

Summary

Several parallel algorithms for the solution of triangular systems of linear equations have been reported by several authors, Chen and Kuck [1], Heller [2], and Orcutt [3]. Among these, the algorithm developed in [1] yields the highest speedup over the sequential algorithm with the fewest processors to achieve this speedup. In this paper, we will first present a simpler proof of this algorithm than that reported in [1].

Consider the triangular system $Ax = f$, where without loss of generality we assume that A is unit lower triangular of order $n = 2^k$, for a positive integer k. The solution x can be written in the form, $x = M_{n-1} M_{n-2} \cdots M_2 M_1 f$ where $(M_{n-1} \cdots M_2 M_1)$ is the product form of A^{-1}, in which $M_i = (I - a_i e_i^t)$ where $a_i^t = (0, \ldots, 0, a_{i+1,i}, \ldots, a_{n,i})$. Thus, x can be computed in parallel in log n stages. Throughout this paper log n denotes $\log_2 n$. At each stage j we perform inner products of vectors the maximum length of which is $1+2^{j-1}$ elements. Assume each arithmetic operation takes one unit of time. Then, by using enough processors, the time required at stage j is $1 + \lceil \log(1+2^{j-1}) \rceil = (1+j)$ steps. Thus the total time required for solving the triangular system is $\sum_{j=1}^{\log n} (1+j) = \frac{1}{2}(\log^2 n + 3\log n)$. In the case that A is a banded lower (or upper) triangular matrix of bandwidth $(m+1)$, i.e., $a_{ij} = 0$ for $i - j > m$, the algorithm requires

$$T_P = (2+\log m) \log n - \frac{1}{2}(\log^2 m + \log m) \qquad (1)$$

steps, where we assume that m is an integer power of two and $m + 1 \leq n$.

The maximum number of processors required by this algorithm is given in [1],

$$P \leq \frac{3}{4} m^2 n + O(mn) \qquad \text{for } m+1 \ll n,$$
$$\leq \frac{n^3}{68} + O(n^2) \qquad \text{for } m+1 \leq n. \qquad (2)$$

However, in practice we may deal with the case when the available number of processors q is less than P above, i.e., $1<q<P$. Several techniques (folding, cutting and sweeping) are available [4, 5] for mapping the above algorithm onto a small set of processors and generally increasing the efficiency of the computation as well. We present here one of these schemes.

*This work was supported in part by the National Science Foundation under Grant DCR73-07980 A02.

**Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Theorem 1 Let T_P and P be as in Eqs. (1) and (2). Then, by applying the cutting scheme, we have

$$T_q \leq 2\lceil m/q \rceil (n-1) \qquad \text{for } 1 < q \leq m, \qquad (3)$$

$$\leq \frac{\beta}{72} n q^{\frac{-1}{3}} (\log^2 q + 27\log q + 144) + \frac{2mn}{q} \qquad \text{for } m < q < m^2 \qquad (4)$$

$$\leq \frac{\beta}{72} n q^{\frac{-1}{3}} (\log^2 q + 27\log q + 144) \qquad \text{for } m^2 \leq q \leq m^3, \qquad (5)$$

$$\leq \beta \frac{m^2 n}{q} (\log m \log q + 2\log q - \frac{5}{2} \log^2 m - \frac{7}{2} \log m + 1) \qquad \text{for } m^3 < q < P, \qquad (6)$$

where $\beta(m,n,q)$ is a small constant.

A backward error analysis of the algorithm for solving the dense unit lower triangular system $Ax = f$ shows that if $\tilde{x}$ is the computed solution, then $(A+\delta A)\tilde{x} = f$ where

$$||\delta A|| \leq \alpha \varepsilon \kappa^2(A) \, ||A||^2$$

in which ε is the unit roundoff, $\alpha = \log^2 n + O(\log n)$ and $\kappa(A)$ is the condition number of A. $||.||$ stands for 1, ∞, or Frobenius norms. This bound can be very large compared to that of the serial algorithm in which $||\delta A|| \leq \varepsilon n ||A||$.

References

[1] S. C. Chen and D. J. Kuck, "Time and Parallel Processor Bounds for Linear Recurrence Systems," IEEE Trans. on Computers, Vol. C-24, No. 7, pp. 701-717, July, 1975.

[2] D. Heller, "A Determinant Theorem with Applications to Parallel Algorithms," Carnegie-Mellon University, Dept. of Comput. Science, March, 1973.

[3] S. E. Orcutt, "Parallel Solution Methods for Triangular Linear Systems of Equations," Rpt. 77, Digital Systems Lab., Stanford University, 1974.

[4] D. J. Kuck, Y. Muraoka and S. C. Chen, "On the Number of Operations Simultaneously Executable in FORTRAN-Like Programs and Their Resulting Speed-Up," IEEE Trans. on

Comput., Vol. C-21, pp. 1293-1310, Dec., 1972.

[5] S. C. Chen, "Parallel Schemes for Solving Triangular Systems," to be published.

SORTING ALGORITHMS FOR PARALLEL PROCESSING

C. C. Lee, T - Y Feng
Department of Electrical & Computer Engineering
Syracuse University
Syracuse, N.Y. 13210

Summary

A sorting network consists of a number of basic comparison elements (BCE's). Each BCE is a device that can compare inputs and yields ordered outputs. An algorithm that is the most efficient known so far for parallel processing has been described by Batcher [1]. The algorithm is Batcher's bitonic theorem. The introduction of perfect shuffle [2] to the bitonic sorting networks has shown that many standard modules of convenient size can be saved while the number of iterations is increased.

In this research, it is based upon two important factors to determine the best sorting algorithm for parallel processing. They are: (1) the number of BCE's, and (2) the number of comparison levels. The basic notion involves the interconnections between comparison levels. The interconnections can be expressed as a mathematical transformation between input data and output data. This transformation preserves the information content while at the same time allowing information to be manipulated by data manipulating functions [3] in a manner that is more efficient than could be done on a random processing.

As far as the number of comparison levels is concerned, bitonic sorting networks are superior to those existing so far. One of the reasons for their superiority is that they have the advantages of flexibility and modularity. The modularity implies that a larger network can be built up by several identical modules. Moreover, the perfect shuffle applied to the bitonic sorting networks can even eliminate the modularity. This can be achieved simply by introducing a column of BCE's and a data manipulator performing the perfect shuffle.

The process in which the two-way merge operation takes any two ordered groups of 2^{v-1} items, $0 < v \leq [\log_2 u]$, to form a single ordered group of 2^v items is called a stage, where $u(\leq N)$ is the number of input items to be sorted. It is known that each stage of bitonic sorting networks, which can be divided into p stages for $N = 2^p$ items, has j, $1 \leq j \leq p$, different interconnections. These j different interconnections turn out to be an identity permutation that can also be performed by a much easier manner in which a certain interconnection is repeatedly applied j times in jth stage. This tremendous saving is easy to see. Since executing a certain interconnection j times is much easier than executing j distinct interconnections in the jth stage. One of the contributions of this research is that we have shown that only one interconnection is needed in each stage of bitonic sorting networks and it is applied j times in jth stage. Hence there are p-1 interconnections needed for sorting $N = 2^p$ items (the interconnection for the first stage is trivial).

The operations of BCE's can be defined as "0", "1" and "Ø". The "0" or "1" operation is defined as the largest of input items coming out of 'H' output lead, which is situated either at the bottom or at the top of BCE's. The "Ø" operation means that all inputs go straight through BCE's without changing relative positions at their output leads.

We introduce a graphic method to analyze the characteristics of multitonic sequences with respect to 2-sorters. In general, a multi-tonic sequence, or m-tonic sequence, with some special conditions can be sorted into two m/2-tonic sequences of their original half length in one level of comparisons. Hence a random ordered input list of $N = 2^p$ items can be first sorted into a 2^{p-1}-tonic sequence. Secondly, the 2^{p-1}-tonic sequence is sorted into a bitonic sequence through a process of p-2 stages with p iterations in each stage. Finally, the bitonic sequence is sorted in order by bitonic theorem [1]. The feature of these 2-sorter perfect shuffle networks is that only "0" and "1" operations are allowed to perform in BCE's. The construction not only eliminates "Ø" operation of BCE's [2] but also gives more flexibility. The flexibility means that the operations of some BCE's can be all "0" or "1" to replace "Ø" operation without changing the sorting outcome. The elimination of "Ø" operation for BCE's, of course, attains a simpler logic for output control of BCE's.

Similarly, the graphic method is used to analyze the characteristics of multi-tonic sequences with respect to 3-sorters. For $N = 3^p$ items, the selected input is a 3^{p-1}-tonic sequence, which can be easily arranged with N/3 3-sorters, rather than a tritonic sequence. The construction of 3-sorter perfect shuffle networks for $N = 3^p$ items is thus developed.

References

[1] Batcher, K.E., "Sorting Networks and Their Applications", Proc. AFIP Spring Joint Comp. Conf., pp. 307-314, April 1968.

[2] Stone, H.S., "Parallel Processing with the Perfect Shuffle", IEEE Trans. on Computers, Vol. C-20, No. 2, pp. 153-161, February 1971.

[3] Feng, T-Y, "Data Manipulating Functions in Parallel Processors and Their Implementation", IEEE Trans. on Computers, Vol. C-23, No. 3, pp. 309-318, March 1974.

SHORT-TERM WEATHER PREDICTION ON ILLIAC IV

James Daley
and
B. D. Underwood

Institute for Advanced Computation
1095 East Duane Avenue
Sunnyvale, California 94086

Summary

We have implemented a short-term numerical weather prediction model on ILLIAC IV, which runs fast enough to be used operationally. The model, originally developed by Kaplan and Paine [1], produces an 18-hour forecast for a rectangular window about the size of the United States in 14 minutes. The physical model permits small scale atmospheric motions, generally filtered in other models, and their exchange of energy with large scale motions. Kaplan and Paine have shown that the inclusion of these mesoscale effects results in good forecasts, but the necessary fine-mesh and short time-step are computationally taxing. Since it is desired to run this model operationally, it is a good candidate for a high-speed parallel processor.

The original version of the model runs on a Univac 1110 on a 49 x 49 x 10 mesh. We have increased the mesh size to 64 x N x 10, with N variable, so as to cover a larger window and make a longer range forecast. The effect of surface friction has been added. A performance comparison of our version with a similar sequential implementation is shown in the facing table.

ILLIAC IV consists of 64 processors which execute a single instruction stream, each processor using its own memory of 2048 64-bit words [2]. Although some minor mathematical changes were required, the vector nature of our problem made it possible to efficiently utilize all 64 processors.

The desire to use large and variable meshes necessitated our using the high-speed ILLIAC IV disk memory (I4DM). The geometrical placement of data on I4DM is under programmer control, and this enabled us to virtually eliminate latency time. Although the total I/O time is surprisingly small, it is possible to overlap it with CPU time.

The requirement of parallel execution and the interdependence of mesh values with others at many neighboring mesh points made the small processor memory of ILLIAC IV a potential problem. A program structure was devised which minimizes processor memory, allows a variable number of mesh rows to be read in or out at a time, and makes the code manifestly reflect the problem definition.

With a high-level language and without optimization, the speed-up over the 360/67 is 147 with asynchronous I/O or 116 with synchronous I/O. Using ILLIAC IV 32-bit mode, we could expect to double these figures.

	ILLIAC IV	360/67
Language	CFD	FORTRAN
Mesh	64 x 64 x 10	64 x 64 x 10
Word Size	64 bits	32 bits
Central Memory Used	88K	570K
Disk Memory Used	310K	0
CPU Time	11 min.	27 hrs.
I/O Time	3 min.	0

References

[1] Kaplan and Paine, "The Proposed AFGWC Operational Primitive Equation Model," given at the Fifth Conference on Weather Forecasting and Analysis, March 4-7, 1974, St. Louis, Mo. Published by the Amer. Meteor. Soc.

[2] Barnes, et. al., "The ILLIAC IV Computer," IEEE Trans. on Computers, Vol. C-17 (August, 1968, pp. 746-757.)

THE FEASIBILITY OF USING ASSOCIATIVE PROCESSORS IN CHANGE DETECTION*

P. Bruce Berra and Ashok K. Singhania
Department of Industrial Engineering
and Operations Research
Syracuse University
Syracuse, N.Y. 13210

Summary

Change detection involves comparison of a pair of given reference and collateral images in order to detect differences between the two. One of the major problems in this process [4] is the spatial alignment of the images. Correlation techniques are employed for this purpose - called image registration. A second problem that arises is due to differences in transparencies and contrast between the images. Adjustment for transparency differences - called photo normalization, together with image registration will produce a normalized and warp corrected collateral image that is ready for subtraction on a point by point basis from the reference image in order to produce the change data.

The inherent parallelism in most of the above processes can be effectively exploited through the use of an associative processor (AP) [1]. We have presented a procedure for the change detection process and have developed algorithms for implementing the various modules of the process on a "general purpose" associative processor. Corresponding timing equations are developed, in particular for the Goodyear/STARAN AP [2].

The major part of the change detection time is spent in the correlation process, as indicated in the results. A reduction in the amount of time to perform correlations would thus cause a significant reduction in time for the overall process. The inherent parallelism involved in the correlation, warp correction, photo normalization and subtraction processes indicates that a large reduction in time should be achieved by implementing these processes on an associative processor. Details of the processes involved and the timing equations can be found in reference [2].

As an example, for a pair of images with 2.5×10^6 picture cells, and with 250 correlation areas of 101 x 101 cells on the reference image, the process would require a total of 200 seconds if carried out on a STARAN with 1,024 words. This assumes that the grey intensities are encoded on a 16-point scale (requiring 4 bits). The above figure reflects software floating point arithmetic functions performed on the STARAN and would be lowered substantially with the availability of hardware arithmetic. The results indicate the attractiveness of the associative processor for change detection.

References

[1] P. Bruce Berra, "A Synopsis of Research Results in the Application of Associative/Parallel Processors to Operations Research, Data Management and Change Detection", 1972 Sagamore Computer Conference Proceedings, Syracuse University, August 23-25, 1972.

[2] P. Bruce Berra, and Ashok K. Singhania, "The Feasibility of Using Associative Processors in Change Detection", Final Report RADC Contract F30603-72-C-0281, May 1974.

[3] Goodyear Aerospace Corporation, "STARAN S Reference Manual", GER-15636, June 1972.

[4] R. L. Lillestrand, "Techniques for Change Detection", IEEE Transactions on Computers, Vol. C-21, No. 7, July 1972.

* This work supported by the USAF Electronic Systems Division through RADC Contract F 30602-72-C-0281, Large Scale Information Processsing System.

PLASMA SIMULATION USING AN ASSOCIATIVE PROCESSOR

Keki B. Irani and Daniel S. Lo
University of Michigan
Ann Arbor, Michigan

Summary

The execution times required by plasma simulation on both a sequential and an associative processor (STARAN) [2], [3], are analyzed. The problem considered is that of a one-dimensional plasma. The simulation uses the particle-in-cell method with the electric potential represented by a continuous piecewise linear function and the electric field by the derivative of the potential. The initial velocities of electrons and positrons are assumed to be selected randomly from, for example, Maxwellian distributions. The particles may be assumed to be positioned initially in electron-positron pairs with uniform distribution. For more details the reader is referred to Lewis, Sykes and Wesson's [1].

We assume that each word in the array memory of an associative processor corresponds to one particle and that various quantities such as the position, the velocity of a particle and the value of electric field at the position of each particle are stored in various fields of the word corresponding to the particle.

In our analysis we concentrate on computation of two quantities, viz the charge density and the electric field. Because of the way we arrange our data in the array memories of an associative processor, the total computation time for updating the positions and velocities of the particles is a constant while for a sequential processor it is proportional to the number of particles.

In the computations of charge density and electric field, most of the time in an associative processor is consumed in rearranging data when the number of particles becomes large. This data rearrangement is equivalent to the information exchange among the processing elements. Unless the tasks are completely independent, information exchange among tasks being executed by processing elements is always necessary. The time required to rearrange data can be reduced if this rearrangement involves shifting data by integer powers of two. This can be obtained, if number of particles and number of cells are selected integer powers of two. Usually this selection does not constitute a restriction on the problem of plasma simulation.

It is demonstrated that for a large number of particles the execution time for both the sequential and associative processors is proportional to the number of particles. However, the constant of proportionality and hence the execution time can be at least ten times larger in the case of a sequential processor. This analysis is based on the assumptions that the number of particles is not greater than the number of words in the array memory of STARAN and core memory in sequential computer is large enough to accommodate the total information about all particles.

References

[1] H.R. Lewis, A. Sykes, and J.A. Wesson, "A Comparison of some Particle-in-Cell Plasma Simulation Method", J. Comput. Phys. 10 (1972), 85-106.

[2] STARAN APPLE Programming Manual, GER-15637A, Goodyear Aerospace Corporation (August 1973).

[3] STARAN MACRO Programming Manual, GER-15643, Goodyear Aerospace Corporation (August 1973).

SOME TIMING FIGURES FOR INVERTING LARGE MATRICES USING THE STARAN ASSOCIATIVE PROCESSOR*

P. Bruce Berra and Ashok K. Singhania
Department of Industrial Engineering
and Operations Research
Syracuse University
Syracuse, New York 13210

Summary

The purpose of this research is to determine the feasibility of solving a system of linear equations on the order of 2000 unknowns using an associative processor of the Goodyear Aerospace STARAN type. A series of timing equations have been developed for complete inversion of the coefficient matrix using Gauss Elimination. Several curves have been plotted [1] under the following assumptions: software or hardware floating point arithmetic; 4 or 16 associative arrays with each array having 256 words of 256 bits each; 32 bit word size for each matrix element; and the data are available in auxiliary memory when and where desired. A summary table is shown below. Based upon these data we believe that the results obtained thus far are promising and that it is reasonable to continue with our effects. We plan to solve actual problems on STARAN and to investigate additional methods for solving systems of linear equations that are amenable to associative processing.

[1] P. Bruce Berra and Ashok K. Singhania, "Timing Figures for Inverting Large Matrices Using the STARAN Associative Processor" RADC-TR-75-73, March 1975

SUMMARY TABLE

	4 Arrays		16 Arrays	
Rank of Matrix	Software Floating Point	Hardware Floating Point	Software Floating Point	Hardware Floating Point
500	4.0**	0.28	3.28	0.22
1000	19.1	1.5	14.0	1.0
1500	86.0	6.6	35.3	2.6
2000	152.9	11.8	62.8	4.6
2500	358.2	27.7	119.5	9.3

** all times are in minutes

* This work was supported by the RADC Post-Doctoral Program, RADC Contract F30602-72-C-0360, Inter-System Communication Electronics.

AN EXPERIMENTAL COMPARISON OF CDC STAR-100 AND 7600 COMPUTER SPEEDS FOR EXPLICIT FINITE-DIFFERENCE HYDRODYNAMICS CALCULATIONS*

Timothy E. Rudy
Computation Department
Lawrence Livermore Laboratory
Livermore, CA 94550

Summary

We evaluated the performance of a 1-D hydrodynamics program on the CDC 7600 [1] CDC STAR-100 [2] computers. The finite-difference algorithm used in the program is explicit in that no recursions appear. That is, every calculation may be expressed in the form

$$\vec{A} = \vec{B} \text{ (operation) } \vec{C}$$

where $\vec{B}$ and $\vec{C}$ are known vectors. Both serial and vector versions of the program are considered.

At LLL, the LRLTRAN [3] langauge with vector extensions is currently being used on the 7600 to compile programs for the STAR. The development of STACKLIB [4] on the 7600 allows programs using the vector extensions to be debugged. STACKLIB uses the instruction stack and concurrency features on the 7600 to obtain increased efficiency and to allow the simulation of a significant portion of the STAR instruction set.

Table 1 shows the performance of various versions of the hydrodynamics program. With the present LRLTRAN compiler, pure serial programs are about six times slower on the STAR than on the 7600. The poor performance of STACKLIB is partly due to short vector length, compiler linkage, and initialization overhead. The STAR vector version is three times faster than the 7600.

Table 1. Serial and vector performance characteristics for 100 zones and 5000 time steps.

	zone-cycles/sec
Serial	
7600-FORTRAN	44.2×10^3
STAR-FORTRAN	7.8×10^3
Vector	
7600-STACKLIB	28.9×10^3
STAR	138.9×10^3

On the STAR, with each vector instruction a sequence of microcode is executed to control the initialization, execution, and termination of the instruction. For timing purposes, the delay associated with these control steps is termed "start-up" time. For example, a full-precision floating-point multiple has a 156-cycle start-up time and results are produced at every cycle.

The available timings [5] enable us to compute a theoretical value for the number of machine cycles required to compute one time step for this 1-D code.

For the 66 vector instructions used in this program, the total number of start-up cycles is 8047. Since the 66 instructions consume 59*N cycles per time step, a vector length of 136 is required to obtain 50% efficiency. Obviously, the longer the vector the more efficient the STAR.

Depending on the density of the data, a programmer may perform the calculation using meshwise, compressed, sparse, or serial techniques.

Several of the key issues in moving a program from a serial environment to a STAR have not been discussed. These include data base reorganization, boundary conditions, and algorithms not readily vectorized (e.g., table look-up and solution of implicit difference schemes).

Based on these experiments, we conclude that a STAR may be degraded to 6600 level by treating it as a serial, or scalar machine. However, if some effort is expended in reprogramming to utilize the vector capability of the machine, a significant improvement in performance may be realized.

*This work is performed under the auspices of the Energy Research and Development Administration.

References

[1] Control Data 7600 Computer System: Reference Manual, Control Data Corporation, St. Paul, Minnesota, Publication 60100000 L (1971).

[2] Control Data STAR-100 Computer System: Hardware Reference Manual, Control Data Corporation, St. Paul, Minnesota, Publication 60256000 06, (1973).

[3] J. T. Martin, R. G. Zwakenberg, and S. V. Solbeck, LRLTRAN Language Used with the CHAT and STAR Compilers, Lawrence Livermore Laboratory, Rept. LTSS-307 (1973)

[4] F. H. McMahon, L. J. Sloan, and G. A. Long, STACKLIB -- A Vector Function Library of Optimum Stackloops for the CDC 7600, Lawrence Livermore Laboratory, Rept. UCID-33083 (to be published).

[5] Control Data Preliminary Instruction Execution Timing Manual, Control Data Corporation, St. Paul, Minnesota, Publication 60440600 (1974).

EVALUATION CRITERIA FOR PROCESS SYNCHRONIZATION

R. J. Lipton[+]
L. Snyder[++]
Computer Science Department
Yale University
New Haven, Connecticut 06520

Y. Zalcstein[+++]
Computer Science Department
State University of New York
Stony Brook, New York 11794

Abstract -- While there are by now well-established criteria for evaluating serial algorithms, such as space and time measures, these criteria cannot be readily applied to asynchronous algorithms. We propose a method for the evaluation of the performance of an asynchronous algorithm. This method is based on the study of delays that are often introduced when one solves a synchronization problem. We then illustrate this method by proving results about the efficiency of various solutions to synchronization problems.

1. Introduction

A central problem in computer science is that of evaluating competing algorithms for the same task. In the case that the algorithms are to be executed sequentially, several evaluation criteria are commonly used. First, it is easy to express the idea that two algorithms "do the same thing" by the requirement that they have the same input-output behavior. Secondly, given that two algorithms have the same input-output behavior, they may be compared by considering the execution time required, memory space required, numerical (or other type of) stability and so on. By contrast asynchronous algorithms cannot be evaluated so easily, due to several important reasons.

First, asynchronous algorithms -- especially those used in operating systems -- are not necessarily supposed to halt. Indeed, considerable effort is often required to guarantee that they do not halt, i.e. do not deadlock or crash. Therefore, it often makes no sense to discuss the input-output behavior of these asynchronous algorithms. Thus it is not at all clear when two such algorithms "do the same thing".

Another difficulty is that of measuring efficiency. Simply counting the number of steps required to accomplish a task does not reflect the utilization of multiple processors. Are algorithms requiring more steps -- which can be done in parallel -- to be preferred over those requiring fewer steps -- which cannot be done in parallel? Similarly, algorithms requiring less memory are not clearly superior if referencing this memory causes processor interference.

Since we are mainly concerned with synchronization, the questions of efficiency can be stated as: How much overhead is required (and how much is acceptable) to accomplish process synchronization? Will the method we chose to solve our synchronization problem cause delays or interference which are unacceptable?

In this paper we present a criterion for evaluating asynchronous algorithms. Rather than attempt to assign absolute measures of resource utilization -- a task that may well be impossible to do in a useful way -- we define, relative to a suitable measure of time, for each non-negative integer k, a relation, $\underline{\text{simulate}_k}$ between asynchronous algorithms. For asynchronous algorithms Q and P

$$Q \text{ simulate}_k \ P$$

will mean that there is a mapping from computations (state changes) in Q to computations in P. This consideration of state changes avoids the difficulty of non-halting algorithms not being input-output comparable. The efficiency of this correspondence (i.e. the amount of overhead Q requires to accomplish the same effect as P) is measured by the integer k. k measures how closely the "parallelism" of Q and P are related. When $k = 0$, Q uses multiprocessors as efficiently as P, but as $k \to \infty$, Q uses multiprocessors less and less efficiently. Thus there will be a sequence of relations

$$\text{simulate}_0, \text{simulate}_1, \ldots$$

which allow increasing freedom with a corresponding decrease in efficiency.

2. The Model

We have used a more "program oriented" model to study related problems ([5] - [8]). However, experience has shown that, as far as the analysis of synchronization is concerned, it is possible to abstract the model further to the language theoretic one which we present below.

(+) Part of this work was done while at IBM Research Center at Yorktown Heights, and part was supported by ONR under Grant N00014-75-C-0752.

(++) Supported, in part, by ONR under Grant N00014-75-C-0752.

(+++) Supported, in part, by NSF Grant DCR75-01998.

The model will ignore such issues as what kind of language the algorithm is specified in, how the actual scheduling is determined, and, most importantly, how the algorithms are actually implemented. These are, of course, important considerations, but it is our contention that a study of the logical implementation of asynchronous programs is of prime importance.

Let Σ be a finite set. Elements of Σ will be thought of as actions (instructions or statements). Informally, a computation is any sequence of actions that respects the control flow of an asynchronous program P (we assume fixed initial values of all variables so that different sequences represent true asynchronous behavior and are not merely a reflection of different inputs to P). Clearly, if x is a computation, then so is any prefix (initial subsequence) of x. We formalize this notion as follows.

Definition: Let Σ be a finite non-empty set. An asynchronous program is a subset P of Σ^*, the set of all sequences of elements of Σ, which is closed under the operation of taking prefixes. Elements of Σ are called actions and elements of P are called computations.

Definition: Let $P \subseteq \Sigma_P^*$ be an asynchronous program. A cost function is a function $c:\Sigma_P^* \to N$, where N is the set of non-negative integers which is additive with respect to concatenation, i.e. $c(xy) = c(x) + c(y)$. Intuitively, c measures "time".

Let $P \subseteq \Sigma_P$ be an asynchronous program and c be a cost function. Define a delay function.

$d_c:\Sigma_P^* \times \Sigma_P \to N \cup \{\infty\}$ by

$d_c(x,f) = \min \{c(y): y\in\Sigma_P^*$ and $xyf \in P\}$, where

$d_c(x,f) = \infty$ if there is no such y. If $c(x) = \text{length}(x)$ we will denote d_c by d.

$d_c(x,f)$ measures the minimal amount of "time" as measured by c that must elapse before f can execute following x. This quantity is important for several reasons. In a real time system, the value of $d_c(x,f)$ may be critical to the correctness of the system. Also, given additional structure in the model, the delay function acts as a quantitative measure of how well multiprocessors can be utilized.

When comparing two asynchronous programs $P \subseteq \Sigma_P^*$ and $Q \subseteq \Sigma_Q^*$, it is convenient to think of one of them, say Q, as implementing the effect of P by using more primitive operations. According to this view, Q is the "compiled" or "macro expanded" version of P. One can then consider a mapping M from P to Q representing this compilation process. In the model presented here, it will be more convenient to consider the "inverse", say h, of M, from Q to P. Thus a sequence of actions $\alpha f \beta$ in Q will be the "expansion" of a single action g in P. The action f will be considered to implement the action g while α and β will be considered as bookkeeping operations or overhead.

Our model also requires that h be a homomorphism which simply means that flow of control in Q is a copy of the flow of control of P.

Formalizing the discussion above, we obtain the following

Definition: Let Q and P be asynchronous programs, i.e. $Q \subseteq \Sigma_Q^*$ and $P \subseteq \Sigma_P^*$. Then h is a decoder from Q to P provided h is a string morphism from Σ_Q^* into Σ_P^*, i.e., $h(xy) = h(x)h(y)$ for all x,y in Σ_Q^* and $h(f) \in \Sigma_P \cup \{\Lambda\}$ for all $f \in \Sigma_Q$ where Λ is the empty string and $h(Q) = P$. $f \in \Sigma_Q$ is called observable if $h(f) \neq \Lambda$, otherwise it is called a bookkeeping action.

We can now define simulate$_k$.

Definition: Let Q and P be asynchronous programs over alphabets Σ_Q and Σ_P respectively, and let c be a cost function on Σ_Q. Then

Q simulate$_k$ P

provided that there is a decoder h from Q to P such that for all $x \in \Sigma_Q^*$ and $f \in \Sigma_Q$, f observable,

$h(x)h(f) \in P$ implies $d_c(x,f) \leq k$.

Intuitively, if, after a sequence of actions h(x), P is not "stopped" i.e. some action g may proceed, then Q may be stopped at x but only temporarily, in the sense that there is a bound k on the amount of time, as measured by c, that must elapse before the action f corresponding to g can be "released". This is our measure of efficiency.

The smallest k such that Q simulate$_k$ P will be denoted by delay (Q,P).

3. Examples and discussion

In this section we illustrate the preceding definitions by examples from the literature. To simplify the discussion, we will use a suggestive informal notation, as is commonly employed in the synchronization literature. It should be pointed out that the advantage of the abstract definition of an asynchronous program is its conceptual economy and aid in simplifying proofs. For describing particular examples, a "program oriented" notation is clearly preferable. This is quite analogous to the description of languages by grammars.

Consider the following asynchronous programs which we take as defining the semantics of the "first reader-writer problem" of [1] (for a discussion of the semantics of synchronization problems see [5], [6]).

reader-i ($1 \leq i \leq n$)	writer
c_i: P(S)	j: P(S|n)
e_i: read	k: write
h_i: V(S)	l: V(S|n)

where S is a global variable (semaphore) whose initial value is n and P,V are Dijkstra's primitives, while P(S|n), V(S|n) are the generalizations of these primitives [9]. P(S|n) is an indivisible action of the form

<u>when</u> $S \geq n$ <u>do</u> $S \leftarrow S - n$

the assignment $S \leftarrow S - n$ is executed only when $S \geq n$, otherwise, control is interrupted until such time as $S \geq n$ is satisfied. V(S|n) is an indivisible action of the form

<u>when</u> true <u>do</u> $S \leftarrow S + n$.

Each of the processes <u>reader i</u> and <u>writer</u> is cyclic so that for example, j can proceed after execution of jkl. Let us denote the set of computations of this program by P. For example, $c_1c_2 \in P$, while $jc_2 \notin P$.

Now let Q_1 be the asynchronous program of figure 1. This program corresponds to the solution to the first reader-writer problem found in [1].

```
integer readcount; (initial value 0)

semaphore M,W; (initial value 1)

reader-i  1 ≤ i ≤ n
A_i : P(M)
B_i : readcount ← readcount + 1
C_i : if readcount = 1 then P(W)
D_i : V(M)
E_i : read
F_i : P(M)
G_i : readcount ← readcount -1
H_i : if readcount = 0 then V(W)
I_i : V(M)

writer
J : P(W)
K : write
L : V(W)
```

<u>Figure 1.</u> First solution to reader-writer problem.

We will now study the relationship between Q_1 and P. First let h be the mapping defined by:

$$\begin{aligned} h(C_i) &= c_i \\ h(E_i) &= e_i \\ h(H_i) &= h_i \\ h(J) &= j \\ h(K) &= k \\ h(L) &= l \\ h(X) &= \Lambda \text{ for all other actions } X. \end{aligned}$$

It is not difficult to verify that $h(Q) = P$. For instance the computation

$A_1B_1C_1D_1A_2B_2C_2D_2$

maps under h to c_1c_2.

We wish to measure the efficiency of this solution. First, we claim that for the given decoder h, Q_1 simulate$_k$ P implies $k \geq 3$. To see this, take $x = A_1B_1C_1$ and $f = C_2$, then the shortest y such that $xyf \in Q$ is $D_1A_2B_2$, which exits from the critical section of reader-1 restores the semaphore M to 1 and then enters the critical section A_2-D_2 of reader-2. Thus $d(x,f) = 3$, while $h(x)h(f) = c_1c_2 \in P$. By a straightforward analysis of cases, based on the observation that one need execute at most three actions between two "successive" observables, it follows that, for this decoder, $k \leq 3$. Next, we claim that under no decoder h_1, can either A_i or B_i be observable actions. Assume the contrary and let $h_1(A_i) = c_i$ Since $A_1J \in Q$ and since clearly $h_1(J) = j$, $h(A_1J) = h(A_1)h(J) = c_1j \notin P$, which is a contradiction since h maps computations into computatio Similarly, $h_1(B_i) \neq c_i$. Thus either $h_1(C_i) = c_i$ for all i and we can argue as before that $k \geq 3$, or, for some i, $h_1(D_i) = c_i$, but, in the latter case $d(\Lambda, D_i) = |A_iB_iC_i| = 3$ so $k \geq 3$ in all case Hence delay $(Q_1,P) = 3$.

Thus Q_1 introduces new delays, but delay (Q_1,P) is <u>fixed</u>, independently of the number n of readers.

Let us now compare Q_1 with an alternat solution, Q_2, represented in figu

```
$S_1 = S_2 \ldots = S_n = 1$ initially

reader i   $1 \leq i \leq n$              writer

$C_i$:   $P(S_i)$                   $J_1$:   $P(S_1)$
                                    ⋮
$E_i$:   read                       $J_n$:   $P(S_n)$

$H_i$:   $V(S_i)$                   K:       write

                                    $L_1$:   $V(S_1)$
                                    ⋮
                                    $L_n$:   $V(S_n)$
```

Figure 2. Second solution to reader-writer problem.

Clearly any decoder h from Q_2 to P will have to map $h(C_i) = c_i$, $h(E_i) = e_i$, $h(H_i) = h_i$ and $h(K) = k$. If $h(J_i) = j$, $i \neq 1$, then for $x = J_1$ and $f = C_1$, $d(x,f) = n + 1$ and $h(xf) = c_1 \varepsilon P$, while if $h(J_1) = j$, for $x = J_2$ and $f = C_2$, $d(x,f) = n + 1$. Thus delay $(Q_2,P) \geq n + 1$.

Thus Q_2 introduces delays that are unbounded as a function of the number of readers present. Therefore, delay (Q,P) is a quantitative measure of efficiency which agrees with the intuition that Q_1 is a better solution than Q_2.

The above differences become even more interesting if we allow different cost functions.

For example, we may want to use a weighted length function c. Observing that most of the time is actually spent in the "read" and "write" sections of the program, we may assign to the "read" and "write" actions weight $t > 1$ while all other actions in Q_i and P get assigned weight 1.

With respect to this cost function, delay (Q_1,P) is still 3. However, delay $(Q_2,P) \geq n + t$ since for the above values of x and f, the program has to go through the "write" section in order to release the reader.

4. Existence Theorems

In this section we give proofs of various simulation results concerning Dijkstra's P and V primitives.

In our previous work [5], [6], [8], we have shown that with respect to a suitable notion of "simulate", PV systems are too weak and cannot simulate even rather simple synchronization problems. Many readers of our work objected to "simulate" as being too strong, based on the intuitive feeling that PV is "universal". Using the "simulate$_k$" relation, we now show that PV is "universal" in the sense that for any asynchronous program P, there is a PV program Q such that Q simulate$_k$ P, for some k. However, k grows unboundedly as a function of the size of P.

In the following, we will use the when...do notation introduced in [5]:

$$\underline{\text{when}}\ \beta\ \underline{\text{do}}\ \Theta$$

where β is a predicate and Θ is a statement means that Θ is executed only if β is true. Otherwise, control is interrupted until such time as β is true.

Definition: A PV asynchronous program is an asynchronous program P such that there is a distinguished subset $\mathcal{S}$ of the program variables (the elements of $\mathcal{S}$ are called semaphores) which can only be used by actions of the form P(S) or V(S), $S \in \mathcal{S}$, ([2]) where

$$P(S) \text{ is } \underline{\text{when}}\ S > 0\ \underline{\text{do}}\ S \leftarrow S - 1 \quad \text{and}$$

$$V(S) \text{ is } \underline{\text{when}}\ \text{true}\ \underline{\text{do}}\ S \leftarrow S + 1$$

Theorem 1. *For every asynchronous program P, there exists a non-negative integer k and a PV asynchronous program Q such that Q simulate$_k$ P, with length as cost function.*

Proof. Let P be an asynchronous program and suppose P consists of n actions of the form

$$(1)\ \text{when}\ \beta_1\ \text{do}\ \Theta_1$$
$$\vdots$$
$$(n)\ \text{when}\ \beta_n\ \text{do}\ \Theta_n$$

We construct Q as an asynchronous program containing n + 1 processes, where the first n are constructed from the n actions of P and the n + 1 -st is a "monitor". The monitor can actually be incorporated into the individual processes, but its isolation as a separate processes enhances the efficiency and readability of the program.

For the i-th action of P, construct the following process in Q:

```
                    process-i
(1)   L_i:          P(S_i)
(2)                 P(S)
(3)                 w ← w + 1
(4)                 if w = np then V(E)
(5)                 V(S)
(6)                 P(E)
(7)                 if ok = 1 then
(8)                     Θ_i
(9)                 ok ← 0
(10)                w ← w - 1
(11)                if w > 0 then V(E)
(12)                         else V(M)
(13)                goto L_i;
```

Where S is a mutual exclusion semaphore (initial value 1) used to protect the critical section (2)-(5), E is a semaphore (initial value 0) used to release all actions that may execute at a given step (the "ready-set" in the terminology of [5]). M is a semaphore (initial value 1) used for communication with the monitor. S_i is a local semaphore (initial value 0) which is enabled at a given step if the i-th action <u>may</u> execute at that step. The variable w is a counter, np is a variable giving the <u>number</u> of actions that may execute at any step and ok is a flag.

The monitor process is given by

```
LM:    P(M)
       ok ← 1
       np ← $\phi(\beta_1,\dots,\beta_n)$
       t ← $\psi(\beta_1,\dots,\beta_n)$
       if q(t,1) then V(S_1)
          ⋮
       if q(t,n) then V(S_n)
       goto LM;
```

ϕ is a function that computes the number of processes that may execute given the values of the β_i's and ψ is a function that figures out which processes may execute and encodes this information into t. q(t,i) will decode t and enable S_i accordingly.

The monitor starts and enables some process-i then enters its critical section (2)-(5). Inside the critical section it acknowledges that it is ready to execute and waits on (6) for release. If all pending processes have so acknowledged, one of them is enabled in line (4). Note that the scheduling responsibility has <u>not</u> been usurped by this simulation in the sense of deciding <u>which</u> process will execute next.

Assuming process j is the first to execute beyond line (6), and since ok will be 1, it will execute Θ_j, disable all others from executing (9) (since executing Θ_i could have changed which processes may now proceed), acknowledges that it has passed (10) and then releases another process if not all have been released (11), otherwise it releases the monitor (12).

Let $h((8)_i) = i$ and $h(f) = \Lambda$ for all other actions in Q. Evidently $h(Q) = P$. To bound the efficiency of the simulation, observe that between any two consecutive observable actions $(8)_i$ and $(8)_j$, r bookkeeping actions are executed, where

$$r \leq 5u + 2n + 8 + 5v$$

where u is the number of processes that may execute after $(8)_i$ and v the number that may execute after $(8)_j$. Since $u,v \leq n$, k is bounded by $12n + 8$.

The proof of Theorem 1 suggests another cost function -- the number of observable actions in a word. Let us denote this cost function by c_1. Then we have the following:

<u>Corollary</u>. <u>For every asynchronous program</u> P, <u>there exists a PV asynchronous program</u> Q <u>such that</u> Q <u>simulate</u>$_0$ P <u>with cost function</u> c_1.

<u>Proof</u>. Immediate from the proof of Theorem 1, since there are only bookkeeping actions between $(8)_i$ and $(8)_j$.

In [8], we have shown that PV systems with only binary ({0,1}-valued) semaphores are strictly weaker than PV systems in the sense that there are PV systems that cannot be simulated by any PV system with only binary semaphores. However, we have the following

<u>Theorem 2</u>. <u>For every PV asynchronous program</u> P, <u>there is a PV asynchronous program</u> Q <u>with only binary semaphores such that</u> Q <u>simulate</u>$_3$ P, <u>with length as cost function</u>.

Proof. The construction is essentially sketched in [2]. For each semaphore S in P, add a new mutual exclusion semaphore E (initial value 1) and a new integer variable x (initial value 0). Q is obtained from P by replacing each P(S) by

```
P(E)
$x \leftarrow x - 1$
if $x < 0$ then begin V(E);
                     P(S) end
                else V(E);
```

and V(S) by

```
P(E)
$x \leftarrow x + 1$
if $x \leq 0$ then V(S)
V(E)
```

Let the third line in each expansion be the observable action. Then clearly $h(x)h(f) \in P$ implies $d(x,f) \leq 4$.

Acknowledgements

We would like to thank M. Condry, E. Ekanadham, P. Henderson, N. Pippenger, and A. Silberschatz for several helpful discussions.

References

[1] P.J. Courtois, F. Heymans and D.L. Parnas, "Concurrent control with 'readers' and 'writers'", CACM 14 (Oct. 1971) pp. 667-668.

[2] E.W. Dijkstra, "Cooperating sequential processes", in Programming Languages (F. Genuys, ed.), Academic Press, pp. 43-112.

[3] R.M. Karp and R.E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing", SIAM J. Applied Math. 14 (1966) pp. 1390-1411.

[4] R.M. Karp and R.E. Miller, "Parallel program schemata", J. Comput. Syst. Sci. 3 (1969), pp. 147-195.

[5] R.J. Lipton, "Limitations of synchronization primitives with conditional branching and global variables", Proc. Sixth ACM Symposium on Theory of Computing (1974) pp. 230-241.

[6] R.J. Lipton, Limitations of synchronization primitives, Research Report #31, Computer Science Department, Yale University, (Aug. 1974), 51 pp.

[7] R.J. Lipton, "Reduction: A new method for proving properties of systems of processes", Conference Record of the Second Annual ACM Symposium on Principles of Programming Languages (1975), pp. 78-86.

[8] R.J. Lipton, L. Snyder and Y. Zalcstein, "A comparative study of models of parallel computation", Conference Record of the Fifteenth Annual IEEE Symposium on Switching and Automata Theory (1974), pp. 145-155.

[9] H. Vantilborgh and A. van Lamsweerde, "On an extension of Dijkstra's semaphore primitives", Information Processing Letters 1 (1972) pp. 181-186.

EXPLOITING VECTOR MODE IN AN SISD COMPUTER

by

B. L. Buzbee & L. E. Rudsinski
University of California
Los Alamos Scientific Laboratory
Los Alamos, New Mexico 87545

Summary

Many modern SISD computers embody features with which data streaming can be established, i.e., vector mode. In this paper, we briefly describe how vector mode is achieved and discuss our experience with it on the CDC 7600.

Frequently, the speed of the central processing element (CPE) is much greater than memory speed on a modern high-performance computer. For example, the CDC 7600 has a 27-ns (nanosecond) CPE cycle time and a 270-ns memory cycle. Thus, in order to keep the CPE busy, designers have divided memory into separate independent modules that can be in simultaneous operation, and successive memory references are processed in parallel provided the references are to different modules. Conversely, successive references to the same module result in a long delay, called a bank conflict.

If we have several operands enroute to the CPE from memory, then we must have a buffer to receive them, and multiple registers often play that role. However, they provide another important function in that they can hold intermediate results and thereby eliminate the overhead of storing such results in memory.

Suppose that we have interleaved memory, multiple registers, and independent pipelined functional units (FU's). Conceptually, we then have the possibility of a stream of operands from memory to the registers, a stream from the registers through the FU's and back to the registers (perhaps cycling through several FU's), and finally a stream of results back to memory. This memory-to-memory streaming is what we define as vector mode. In order to achieve it, we must have complete control of all traffic between memory and the CPE. Thus, we cannot tolerate instruction fetching during vector mode as it will almost certainly produce bank conflicts, thereby disrupting the stream. A short loop instruction stack in the CPE eliminates instruction fetching; thus, it is a necessary feature to achieve vector mode.

When executing object code generated by a FORTRAN compiler, the CDC 7600 averages about 9 to 12 million instructions per second (MIPS), whereas in vector mode it averages 20 to 25 MIPS. Fortunately, a lot of work is often concentrated in short loops. Thus, the CDC 7600 productivity can be increased by vectorizing those loops, that is, replace them by calls to carefully written assembly language modules in which vector mode is achieved. This was done with many of our large production codes, and their productvity was increased by 30 to 40% [1, 2].

From this experience, we find that the SISD environment offers some nice features with respect to vector implementation. For example, we can have (1) arbitrary spacing in memory between vector elements subject to the constraint that successive elements reside in distinct modules. The programmer can easily achieve such separation when he lays out his data. This is an important feature because there are numerical algorithms where one operates on both rows and columns of an array. Also, (2) multiple registers minimize memory requirements for intermediate storage and the associated delays. For example, $\vec{F} = \vec{A} + \vec{B} + \vec{C} + \vec{D}$ can be evaluated without any intermediate storage. Finally, (3) the complexity of the "vector function" is limited only by the length of the instruction stack. That is, we have the full generality of an SISD computer at our disposal but constrained by the stack length.

This experience has also illustrated the importance of scalar speed in a vector machine. Note that although the CDC 7600 vector mode execution rate is about twice its scalar rate (FORTRAN), vectorization only increased productivity by 30 - 40%. Thus a scalar machine which is twice as fast as the CDC 7600 scalar mode will significantly out perform a CDC 7600 even if vector mode is used whenever possible. This is consistent with [3] and is due to the fact that 25 - 50% of the total work in an average program must be done in scalar mode. We conclude that scalar speed is a very important parameter in the performance of a vector machine.

REFERENCES

[1] B. L. Buzbee and Fred W. Dorr, "The Direct Solution of the Biharmonic Equation on Rectangular Regions and the Poisson Equation on Irregular Regions," SIAM J. Numer. Anal., 11 (1974) pp. 753-763.

[2] F. McGirt, L. Rudsinski, and K. J. Melendez, "Computer Program Optimization Using Software Monitoring Techniques," presented to the 1973 AEC Computer Information meeting, May 10-11, Lawrence Livermore Laboratory, Livermore, CA. Copies available from the authors.

[3] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967 JJCC of AFIPJ, vol. 30, pp. 483.

AUTHOR INDEX

AUTHOR INDEX (CONT'D.)

REVIEWERS

Prof. J. L. Baer	University of Washington
Dr. Kenneth Batcher	Goodyear Aerospace Corporation
Prof. P. Bruce Berra	Syracuse University
Dr. K. Chandy	University of Texas, Austin
Dr. I-Ngo Chen	University of Alberta/Syracuse University
Dr. Wei-ith Cheng	IBM Corporation
Mr. John Cornell	Systems Development Corporation
Prof. Edward Davidson	University of Illinois
Prof. Jack Dennis	Massachusetts Institute of Technology
Dr. Robert Downs	Systems Control Incorporated
Prof. Caxton Foster	University of Massachusetts
Dr. Garth Foster	Syracuse University
Prof. Domenico Ferrari	University of California, Berkeley
Prof. Bernard Galler	University of Michigan
Dr. Oscar Garcia	University of South Florida
Prof. Mario Gonzalez	Northwestern University
Dr. Bill Hays	Brigham Young University
Mr. Lee Higbie	Massachusetts Maritime Academy
Mr. John Horn	Rome Air Development Center
Dr. Chao P. Hsieh	Syracuse University
Prof. M. K. Hu	Syracuse University
Prof. Keki Irani	University of Michigan
Mr. Larry Jack	Honeywell, Incorporated
Dr. Robert Johnson	Rome Air Development Center
Prof. J. Robert Jump	Rice University
Prof. Robert Keller	Princeton University
Dr. Alan Klayton	Rome Air Development Center
Dr. Peter M. Kogge	IBM
Prof. David Kuck	University of Illinois, Urbana

REVIEWERS

Dr. Leslie Lamport	Massachusetts Computer Associates
Dr. Duncan Lawrie	University of Illinois, Urbana
Prof. Gerald Lipovski	University of Florida
Prof. C. L. Liu	University of Illinois, Urbana
Mr. David McIntyre	University of Illinois, Urbana
Prof. John Marzolf	Syracuse University
Mr. W. C. Meilander	Goodyear Aerospace Corporation
Dr. Steve Nuspl	Honeywell, Inc.
Prof. Suhas Patil	Massachusetts Institute of Technology
Mr. James L. Peterson	University of Texas, Austin
Mr. James L. Previte	Rome Air Development Center
Prof. C. V. Ramamoorthy	University of California, Berkeley
Dr. S. S. Reddi	Rice University
Mr. Oskar Reimann	Rome Air Development Center
Dr. John Sammon, Jr.	Pattern Analysis & Recognition Corp.
Dr. H. Gregory Schmitz	Honeywell, Inc.
Mr. Henry D. Shapiro	University of Illinois, Urbana
Mr. Henk Spaanenburg	Syracuse University
Prof. Edward Stabler	Syracuse University
Mr. Armand Vito	Rome Air Development Center
Mr. K. P. Wang	Syracuse University
Mr. Kuo Y. Wen	University of Illinois, Urbana
Dr. R. Wishner	Systems Control Incorporated
Dr. Sun-maw Yang	Syracuse University
Prof. Stephen Yau	Northwestern University
Prof. R. J. Zingg	Iowa State University