Measures of ideal execution architectures

by Michael J. Flynn
Lee W. Hoevel

This paper is a study in ideal computer architectures or program representations. We define measures of "ideal" architectures that are related to the higher-level representation used to describe a program at the source language level. Traditional machine architectures name operations and objects which are presumed to be present in the host machine: a memory space of certain size, ALU operations, etc. An ideal language-based architecture is based on a specific higher-level (source) language, and uses the operations in that language to describe transformations over objects in that language. The notion of ideal is necessarily constrained. The object program representation must be easily decompilable (i.e., the source is readily reconstructable). It is assumed that the source itself is a good representation for the original problem; thus any nonassignment operation present in the source program statement appears as a single instruction (operation) in the ideal representation. All named objects are defined with respect to the scope of definition of the source program. For simplicity of discussion, statistical behavior of the program and language is assumed to be unknown; Huffman codes are not used. From the above, canonic interpretive (CI) measures are developed. CI measures apply to both the space needed to represent a program and the time needed to interpret it. Example-based CI measures are evaluated for a variety of contemporary architectures, both host- and language-oriented, as well as a CI-derived language-oriented architecture.

Introduction and some definitions

Inadequacies of many familiar machine architectures, both in program size and execution time, pose the problem of executable program representation [1–3]. Secondary effects of these architectures lead to complicated system structures and implementations, e.g., compilers and linkage editors, as well as difficulties in recognition and exploitation of parallelism [4]. A traditional architectural premise is that execution architectures must be fixed, and hence universal, languages.

In such architectures the relationships, actions, and objects required by a program are translated into descriptions of objects presumed to be present *in the machine:* registers, adders, etc. These architectures are *universal* in the sense that they do not change for the various higher-level languages (HLLs) that may be used. Architectures that correspond to a particular HLL are called Direct Correspondence Architectures (DCAs). Since a DCA architecture is oriented to a particular HLL, it allows more information to be preserved concerning language and user environment, while still realizing more concise representation and expeditious interpretation than host-oriented architectures. Thus, in this paper, we assume an *a priori* knowledge of the HLL to be used.

In earlier work [5] on this subject, we have referred to these language-oriented architectures as DELs—Directly Executed Languages. This term proved to cause confusion in apparently restricting attention to efforts that interpreted—

without compilation—HLL source text. As this was not our intention, we have chosen the DCA acronym as a more accurate description of various forms of language-oriented architecture.

Before proceeding, it is useful to introduce some terms which we use throughout this paper. An *instruction* is a function which determines the next state given a current state. The *machine* is a set of instructions, storage, and an interpretive mechanism that implement the state transitions determined by the stored instructions. Now the *interpretive mechanism* itself may be a machine—that is, with its own set of instructions, storage, and interpretive mechanism—in this case, the original machine is called the *image machine* and its interpretive mechanism is called the *host machine.* The program which controls a host machine acting as an interpretive mechanism is called the *emulator* program. The emulator is naturally also an interpreter, and we use these terms interchangeably. The set of all image instructions represents the *execution architecture* of the machine.

By convention, image machine instructions are called simply *instructions,* while host machine instructions are called *microinstructions* or *host instructions.* The instruction itself is a vector of bits partitioned into fields called *syllables.* The syllables identify properties of the instruction which include

1. format,
2. object identifiers,
3. operation identifiers,
4. sequence control.

The format syllable of an instruction specifies the number and type of the remaining syllables in the instruction, the location and use of implied objects, and the type of transformation rule to be associated with the operation (i.e., which objects are operation sources, and their precedence, and which object is the operation result). The object identifier is an explicit pointer to a variable, while the operation identifier specifies the functional transformation to be applied to the source operands. Frequently the format and operation identifier are encoded together. Sequence control selects the next instruction to be interpreted.

In this paper we further assume that programs are initially expressed in a fixed high-level language (HLL) which has been carefully selected by the user to suit the problem environment, and that it is a familiar procedure-oriented language. Within this framework, we will explore measures of ideal execution architectures for a variety of host organizations.

• *Identifiers, objects, and name spaces*

Programs use identifiers as surrogates for objects. Objects—arguments or results—are only associated with values during execution, as defined by the states of a computation. In this

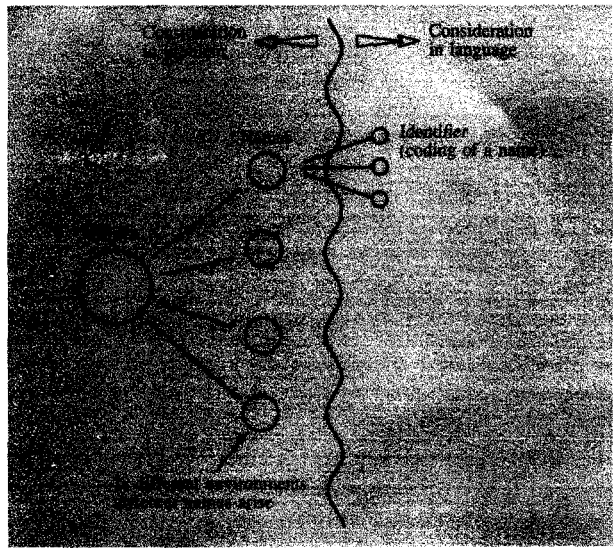

Figure 1

Objects, names, and identifiers.

sense, an identifier is only a specific instance of the abstract name for an object; the name exists independently of a language, while the identifier is closely tied to a particular syntax and semantics (**Figure 1**).

Typically, source-level identifiers are mnemonically selected alphanumeric strings. Image identifiers are usually one-, two-, or three-dimensional binary codes (e.g., base, segment, offset). At the host level, identifiers are simply physical addresses. Typical source-level objects are integer variables, program labels, and boolean flags; typical image-level objects are words, bytes, and bits; and typical host-level objects are registers, buses, and execution units. Potentially, image instructions may access or modify any object in the image storage. Fields within image instructions, called identifiers, are used to locate specific objects in the image store. The set of all locatable objects in the image store is called the *name space* of the execution architecture.

• *Scopes and working sets*

The scope of an identifier is the largest program fragment over which it has a consistent interpretation. For most programming languages, the term scope of definition is used to refer to a given lexical block (procedure, subroutine, function, or begin-end segment) associated with a given level of declarations.

At the machine level, however, the natural interpretation of a scope is as a range of instructions over which indexing registers remain effectively unmodified, so that a given operand identifier always refers to the same program object

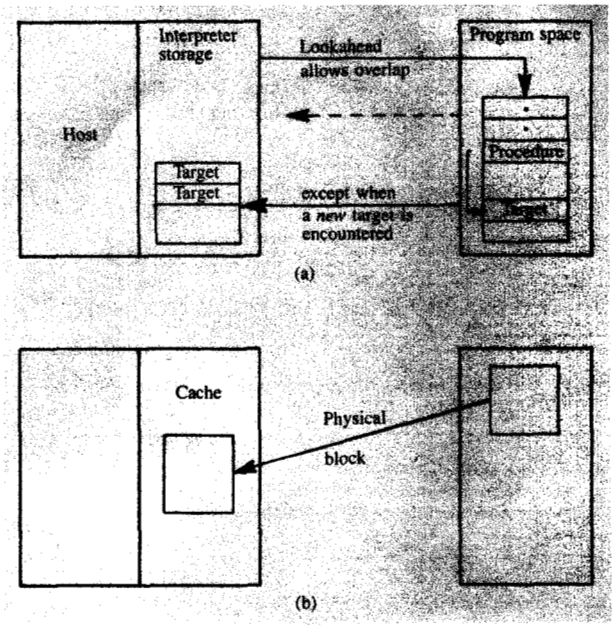

Figure 2

Referencing environments: (a) Language-oriented instructions; (b) Host-oriented instructions.

within a given scope. The term is often used imprecisely, however. The "scope of a procedure" may refer either to the set of statements outside the definition of a procedure in which its own name is a valid identifier, or to the set of statements immediately inside a procedure definition in which identifiers local to that procedure have consistent interpretations.

The basic model

The basic model [6] used in this investigation is illustrated in **Figure 2.** Its most obvious characteristic is that program evaluation is assumed to take place in two distinct phases. First, a source program is converted into an equivalent form during an initial compilation phase. Users retain this form, which becomes a surrogate for the original source version. In the second phase the surrogate undergoes a number of subsequent interpretations.

Two-phased evaluation may be used as the basis for a design model. The principal components of a system in this model include a source language—selected for its representational capabilities; a host machine—selected for its execution capabilities; a translator that takes a source program as input and produces a logically equivalent executable program; and finally an interpreter that enables the host machine to implement the state transitions specified by the output of the translator.

An *ideal execution architecture* is a representation defined with respect to the given source language and environment. The term "ideal" is used here to mean

1. Transparent—i.e., translation is a simple process which preserves equivalent source-state information, thus allowing a ready reconstruction of source constructs.
2. Optimal representation—i.e., space and time to compile and execute are minimized.

• *Translation*

Translation is the process of converting a program in one language into an equivalent program in another language. Equivalence refers to similarity of transformation in the eyes of a user—a program p in language P is equivalent to a program q in language Q if and only if users cannot distinguish between an execution of p according to the semantics of P, and an execution of q according to the semantics of Q.

The virtue of compilers is that they translate a source language program into a form that can be interpreted more efficiently. This means that some sort of reduction in computational complexity takes place. In general, compilation can be viewed as a (partial) binding of operands to storage cells and operators to computational structures.

The special case of direct interpretation of HLL source also fits this model since a translation of a statement is required, although this translation is internal to the interpreter. One may view this as a compilation-interpretation on a statement-by-statement basis in which the intermediate execution architecture is not visible to the user.

For our purposes, compilation involves two quite distinct processes: optimization and translation. We view optimization as occurring on a source-to-source basis. The optimizer rearranges the source program into its "best possible form" within the source language (or an extended version of the source language). The translator then takes this form of the source and creates a surrogate form within the syntax and semantics of the execution architecture. The remainder of this paper assumes that the given HLL program is already in its "best possible form"—i.e., that it has been optimized.

• *Interpretation*

The interpreter **[Figure 3(a)]** may be visualized as consisting of a primary control loop, a set of interface routines, and a set of semantic routines. The primary control loop maintains the DEL instruction stream, and extracts at least an initial format syllable that defines the layout for the rest of the instruction. It then transfers control to the appropriate interface routine, which actually parses any operand references within the instruction.

Interface routines are responsible for creating a standard interface that defines the location (and possibly the value) of

each operand in an instruction. The interface routine then returns to a known point in the primary control loop, which transfers control to the semantic routine that actually implements the action rule defined by the instruction [**Figure 3(b)**]. It does this by transforming the values previously loaded into standard interface. Upon completion of all semantic processing, control returns to the top of the primary control loop, and another cycle of interpretation begins. Results may be stored by the semantic routine itself, or within the primary control loop.

Within the contral loop, interface, or semantic processing, the state of the data store and program state vector is temporarily undefined. Definition occurs at the boundaries; the overall process will be correct if the data store and program status vector agree with defined constructs whenever the last instruction in the expansion for a source statement has been executed.

• *Level mappings and transparency*

Users relate the observable (but low-level) effects of executing a program to source-level semantics through an association established between the source-level name space and the host name space. In general, there is a map from any higher-level L to the next-lower-level $L-1$ that defines the way in which level L is realized at level $L-1$. There is also a dual map from level $L-1$ to level L that defines how an execution of a level $L-1$ program is to be visualized in terms of level-L semantics (**Figure 4**).

More formally, transparency is a property of an interlevel mapping, f from L to $L-1$, such that

i. State transitions in both L and $L-1$ occur in the same order.
ii. States are preserved at the end of a state transition in L; states in $L-1$ correspond directly to states in L, and no new states may be introduced.

The consequences of transparency are significant:

1. There is no hidden data state to save across transitions in L. This implies that there is no need to save data states (i.e., host registers) at an interrupt if interrupts are permitted to occur only at the end of a state transition in L. For example, a program is interruptable upon completion of the STORE corresponding to an HLL assignment, but not within the evaluation of an HLL expression.
2. Synchronization of concurrent processes, if valid at level L, is valid at level $L-1$ since there is an exact correspondence of states and order of interpretation between these two levels. Further, verification of computations is valid at level $L-1$ whenever it is valid at level L.
3. Optimizing compilers are clearly *not* transparent since they affect the number, type, and order of state transitions. Thus, further checking of optimized programs may be required in order to ensure correctness of operation.

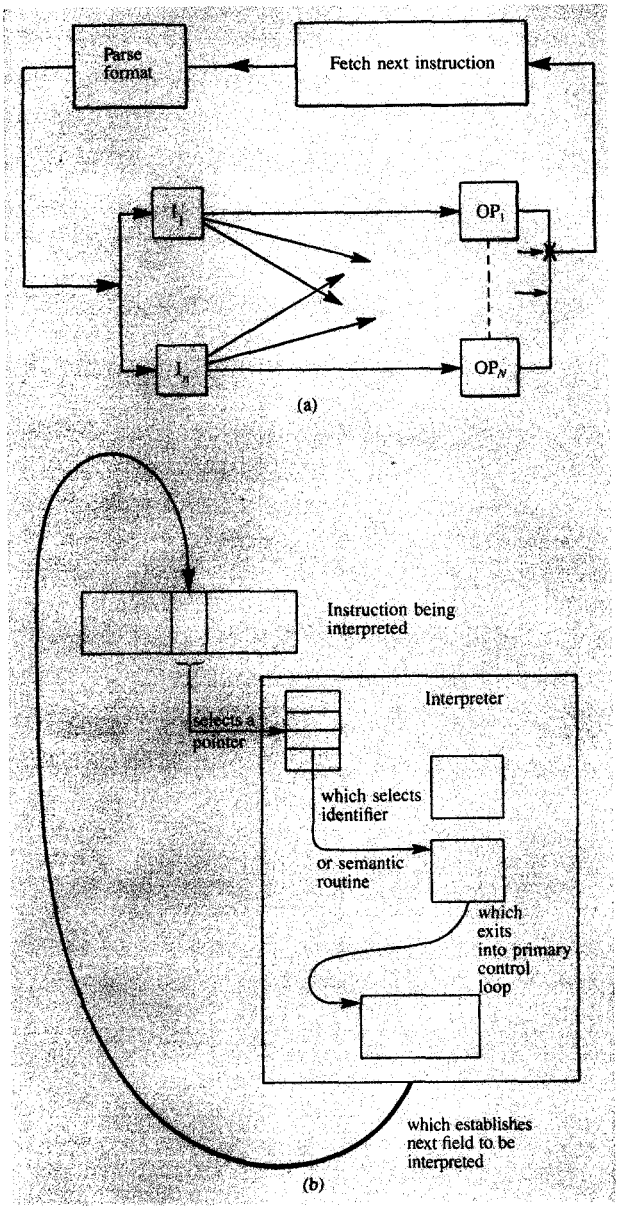

Figure 3

(a) Interpretation process; (b) Syllable interpretation.

Types of interpretive mechanisms: image-host correspondence

An image instruction is partitioned into syllables which must be interpreted by the host machine. Each of these syllables must be individually decoded by the interpretive mechanism

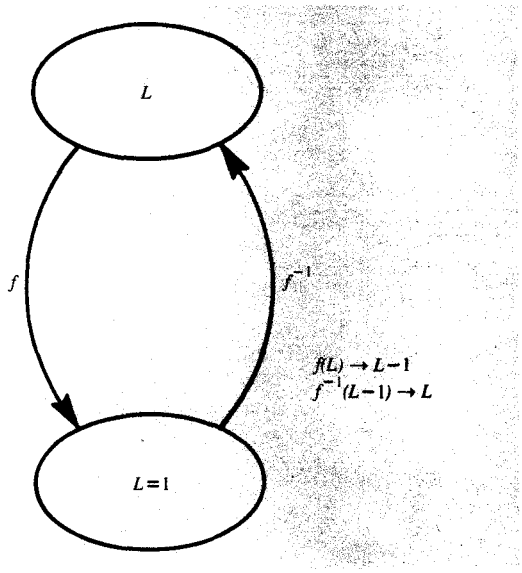

Figure 4

Transparency between a level representation and a level $L-1$.

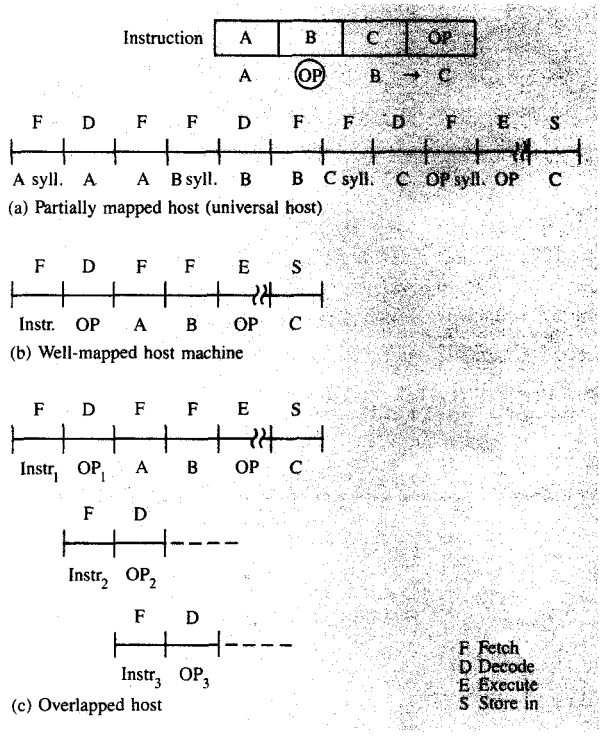

Figure 5

Three host timings.

and executed according to the semantics of the instruction. Interpretive mechanisms or host machines vary considerably in their ability to perform the decoding and execution of syllables concurrently. Below is a classification of interpretive mechanisms by the amount of hardware the host provides for the detection and current execution of image syllables and instructions.

1. Partially mapped host-image correspondence—In the usual form of this type of correspondence (**Figure 5**), the host has no *a priori* support for the interpretation of a particular image syllable. Thus each syllable of the image instruction is separately fetched (from the image storage), decoded, and executed. While Fig. 5 shows a single state transition for each of these actions, multiple transitions may be required—especially if the object size of the image machine is not supported by the basic data paths in the host machine.
2. The well-mapped machine—In this case the host is presumed to have hardware support for the syllable partitioning of image instructions. Thus, once an image instruction has been fetched, its various syllables can be decoded concurrently.
3. The overlapped machine—This class of machine has still additional resources to permit concurrent execution of syllables in the image instruction sequence to as great a degree as is permitted by the image semantics (i.e., dependencies that actually occur during execution must be observed). However, the final state transition in the execution of each instruction occurs in the order of the original sequence; that is, all updates of the image store are performed in logically correct order.
4. The confluent machine—If we remove the constraints of order preservation during instruction interpretation, transparency of execution is lost. The semantics of the instruction stream are preserved only on a process-by-process basis. This type of machine represents an unlimited collection of host resources dedicated to image program execution: unlimited cache, transformational resources, etc. Blocks of instructions are decoded simultaneously, and only the dependencies determined during this decoding process constrain concurrency of image instruction execution. Thus, this ultimate machine is limited by semantic dependencies which could not be anticipated. Its execution time is limited by inherent dependencies within the image code, and by the initial occurrence of various image program artifacts. For example, even if a machine is equipped with a cache of infinite size, it still may have a nonzero cache miss rate because of the misses caused by initial reference to various program and data blocks. Similarly, branch target buffers can be used for concurrent target instruction retrieval only after they have been initialized by a previous encounter during program execution.

Different artifacts of program behavior affect each of the above host machine types in significantly different ways. The total number of environments entered may be of little or no importance to the execution time of a partially mapped machine since it is dominated by the interpretation time of syllables within environments; on the other hand, a confluent machine faces precisely the reverse situation. Thus any attempt to develop measures of ideal execution must take all categories of host machines into account.

Ideal execution architectures

An ideal program representation and corresponding machine uses minimum storage space and requires minimum compilation and interpretation time. This does not imply a linear tradeoff between space and time; relative weights can only be determined within a given user and host context.

The notion of *environment,* however, is fundamental to the discussion of space/time optimality. Like many concepts dealing with computer systems, environment may be viewed as a hierarchical concept. The chosen higher-level language itself represents the highest level, while a specific program would represent a lower level, and individual statements within a program would represent a still lower level. Environment includes all information needed to interpret objects: i.e., all information available to an interpreter, as distinct from information supplied directly by the specific coding of a statement or instruction. Thus, the environment of a traditional machine language would include a program counter, address registers, interrupt status, etc. Within an environment, we can measure the activity required of an interpreter for a given HLL; the number of syllables, instructions, and nonlinear sequence control actions all affect image interpretation time in varying proportion depending on the host.

Associated with each environmental level is the property of *stability.* Intuitively, stability is probably best described as the number of statements or objects that are interpreted before the environment is changed. A change of environment is caused by anything that disturbs the interpreter by delaying it from completing the ordinary execution sequence of a program, statement, or operator. Instabilities arise from one of two basic causes. First, exogenous events outside the current environment—a time-sharing system, for example—may time slice over short intervals, and each program switch would necessitate a change in environment. Second, internal events associated with the nature of a program itself may cause a change in environment; for example, a single user program may involve several languages, or may defer binding between names and values until very late (as in the binding of actual to formal parameters upon entry to a subroutine). In this case, the act of binding the interpreter to a given language, or of binding a name to a given value, also causes a change of environment.

Some programs may repeatedly enter a certain set of environments during execution. The host and its interpreter may be able to take advantage of the limited number of unique environments encountered. This number is an important characteristic of program behavior, which we call *distance.* An associated measure is *distance ratio,* which is defined as the number of unique environments entered as a fraction of the total number of environments entered. The maximum distance ratio for any program is one, and a program attaining this limit would never enter the same environment twice. In many ways, distance is an ultimate limit on the speed of interpretation of a program, since it sets a limit on the possibility of retaining previously interpreted data.

Aspects of execution architectures

In this section we examine some significant aspects of an execution architecture and how they relate to a variety of program and host environments. For each combination, a measure can be found which describes the activity of an ideal architecture during execution of a given program. Many measures, however, are of limited interest; our purpose here is to select some of those combinations which provide a general insight into the relationship between an architecture and the HLL. We identify five criteria for measuring architectures:

1. Correspondence—that the representation correspond to the source representation.
2. Size—that the representation be concise, i.e., a minimal encoding.
3. Reference activity—that the representation minimize the total number of objects to be interpreted.
4. Stability—that the representation minimize the total number of environment transitions.
5. Distance—that the representation minimize the number of unique objects and environments to be interpreted.

• *Correspondence*

Correspondence measures the ability of the architecture to represent (i.e., stand as a surrogate for) HLL objects. Correspondence is not itself a direct measure of space or time; it is rather a quantification of the notion of transparency introduced above.

An "operation" in an execution architecture may correspond to

1. a program,
2. a subroutine,
3. a statement,
4. an operation.

For execution architectures meeting the statement transparency requirement, HLL operations are the most useful level of correspondence. Note that traditional

machine architectures usually employ host (ALU) operations, while special-purpose machines may select certain subroutines or even a program for their operation set. Thus, if

$\hat{A}_i$

is the number of static occurrences of HLL operations in a program, we would expect an ideal architecture to have $\hat{A}_i$ instructions in its representation of the same program. Similarly, if

A_i

is the number of dynamic HLL operations required to *execute* a source language program, we would expect A_i instructions to be executed during the interpretation of any ideal architectural representation of the same program.

Ideally, identifiers in the execution architecture should also correspond to the objects named by the higher-level language. In this case, however, the statement is the most natural environment level. If $\hat{A}_d$ and A_d are the number of unique static and dynamic variables in a given HLL program, we might expect an ideal architecture to contain $\hat{A}_d$ and A_d static and dynamic operands. The counting rules are somewhat complicated, however, since (ideally) only *unique* identifiers need be distinguished. For example, while there are three identifiers in the HLL statement

"$X := X + X$,"

we would count

$\hat{A}_d$

as one (the only unique identifier is "X"), and

A_d

as two (one to fetch the only unique argument value, and one to store the updated value in the only unique result). The variable "X" is counted twice in the dynamic statistic since it occurs in both the range and domain of the statement. Also, an ideal architecture should not introduce artificial operands (i.e., "temporaries"); thus, the HLL statement

"$B := A * B + B$"

would result in

$\hat{A}_d = 2$

and

$A_d = 3$.

Summarizing the above, the correspondence measure for a given source program is

$\langle \hat{A}_i, A_i, \hat{A}_d, A_d \rangle$

and the relative correspondence for any execution architecture a is measured by the quadruple

$$\frac{\hat{a}_i}{\hat{A}_i}, \frac{a_i}{A_i}, \frac{\hat{a}_d}{\hat{A}_d}, \frac{a_d}{A_d},$$

where

$\hat{A}_i$, A_i, $\hat{A}_d$, and A_d

are as above, and

$\hat{a}_i$, a_i, $\hat{a}_d$, and a_d

are the respective counts for the actual execution architecture.

• *Size*

An ideal representation must be concise in its coding of identifiers, yet not so concise that it exacerbates interpretation. Total program size is simply the product of the static count and size of the two basic identifier types noted above. Since the number of identifiers depends on the correspondence environment level (program, subroutine, etc.), however, program size also depends on environment level.

The basic question in measuring size, then, is the determination of a suitable environment and identifier encoding. Identifier size is determined by the number of objects that must be distinguished by a particular identifier, if identifier encoding is independent of frequency of occurrence. Thus, if there are F distinct operators used in an environment, each can be represented by $\lceil \log_2 (F) \rceil$ bits. Similarly, if there are V unique variables in an environment, each operand can be represented by $\lceil \log_2 (V) \rceil$ bits. Identifier encoding may vary in several ways:

1. fixed across environments,
2. variable by environment,
3. variable by frequency of occurrence.

In this work we do not consider frequency-encoded identifiers; given full frequency distribution statistics, the techniques for taking a nonfrequency-encoded scheme and transforming it into a minimal encoding is straightforward and treated elsewhere [7, 8]. Further, minimal encoding by frequency requires serial inspection of the bits within a field, thus increasing the complexity of interpretation as well as interpretation time.

The distinction between the fixed and variable approaches is merely a distinction between fixed and variable environments, and hence is included in environmental considerations. Environment may be determined by either the program, the HLL, or the host. In traditional execution architectures, the environment is determined by the host specification rather than host configuration. Since this environment does not change, identifier containers are usually fixed, and information capacity is wasted in small-capacity configurations.

For source-oriented execution architectures, any of the following may be used to define the number of entities which determine the identifier size (for each class of object—operator, operand, and label):

1. Language restriction—the number of operands and labels are usually not bounded by language syntax in a meaningful way. However, the number of operations are usually *a priori* limited and the number of operations allowed by the HLL definition is a possible bound on operation size.
2. HLL objects used in a program—i.e., the total number of distinct operations, operand names, or labels used in a program could form the domain of an identifier definition. This might be a natural approach for DCAs whose HLL does not have developed notions of subroutines and linkage (e.g., COBOL).
3. HLL objects used in a subroutine—In those HLLs with a well-defined notion of scope, each subroutine has its own scope of definition. This is an interesting environmental unit for operand and label identifiers.
4. The HLL statement—The even-lower-level concept of a statement for an environment might also form the basis of a size environment. However, since entry into and exit from an environment requires interpretation time, the statement may be at too low a level to provide an optimum space/time tradeoff. That is, at the level of the statement the setup time required may not offer a worthwhile space/time tradeoff, since the size of the identifiers increases as a log function while the interpretation time is linear in the number of statements to be interpreted.

At the subroutine level, setup time can be regarded as being relatively small compared to the interpretation time for the overall subroutine, while at the statement level this may not be true. However, for interpretive languages (APL, BASIC, etc.) where a line-by-line execution is performed, the HLL statement is a natural environmental unit.

From the above, an ideal architecture has a size environment whose domain is determined by the number of HLL objects used in either the representation, the program, the subroutine, or the statement. For familiar procedural languages (Pascal, FORTRAN, etc.), the statement level does not seem to be a good choice because of implied interpretive overhead, while an environmental unit consisting of the entire language or program seems equally inefficient in its lack of conciseness. For such languages, it appears that the subroutine level is a natural space/time optimum.

• *Dynamic activity*

Measuring the dynamic activity of an architecture is significantly more difficult than measuring program size. Interpretation time depends not only on a simple count of object references, but also on the kind of object being interpreted and underlying hardware support for special cases. To characterize this activity, we need multiple measures, each of which reflects a different level of program execution. We see the primitive dimensions of this measure in terms of the following concepts:

1. Referencing: execution is limited by the dynamic count of objects requiring interpretation in simple hosts—instruction and operand interpretation.
2. Stability: slightly more elaborate hosts can predict, and hence overlap (or hide), most object references; their execution time is dominated by the occurrence of unpredictable events (e.g., conditional branches).
3. Distance: the ultimate host has unlimited hardware resources, and can overcome most of the disruption caused by lack of stability through extensive use of information retention. For example, conditional branches can be anticipated by recalling previously encountered interpretations. Here, the "first encounter"—what we call *distance*—is the limiting factor.

• *Referencing activity*

Referencing activity is composed of instruction fetches and data accesses (both fetch and store activity).

Instructions Source-oriented execution architectures introduce the following straightforward considerations with respect to the instruction-fetch mechanism. For partially mapped hosts, the number of instruction syllables that must be interpreted is the primary measure of instruction activity. A basic measure of the number of HLL inferred syllables is simply $A_i + A_d$. This is an overstatement since A_d includes a double count for syllables used as both a range and domain element. Actually, the dynamic count of syllable A_s is

$$A_s = A_i + \sum_i (\hat{A}_d)_i ,$$

where the second term is the static number of data syllables contained in an instruction summed over the dynamic instruction stream. If the average number of syllables per instruction is M, then by definition $M \times A_i$ is the dynamic syllable count. For well-mapped hosts, the number of operations to be interpreted (A_i) is the primary measure of instruction activity.

Data Source-oriented execution architectures introduce the following considerations with respect to data access mechanisms: for a partially mapped host, reference activity is determined by *the relative data path width*,

$$N = \left(\frac{\textit{average operand size}}{\textit{host data path size}} \right),$$

and A_d. The dynamic reference count for operands is $N \times A_d$. For a well-mapped machine, there should be one

reference per identifier, and the operand reference count is simply A_d.

Notice that N is not a property of the program representation but rather a property of the host. Thus, it is irrelevant to discussions of ideal architecture but quite important in comparisons with hosts of varying data path widths and their ability to execute various program representations.

While a simple dynamic count of data references is a useful measure, the locality of such a reference does not necessarily correspond to reference counts, complicating performance models. Locality is not a direct property of a source program, but source aspects—notably stability, discussed below—have a direct influence on locality.

• *Stability*

Stability measures the number of changes to the way objects are interpreted. The effect of stability is more pronounced given a host that takes advantage of consistency in environments and predictability of the occurrence of objects and events.

a. Let S_e be the total number of environments encountered in the dynamic HLL program execution. Ideally, the number of calls/returns (or equivalent architectural artifacts like "perform" in COBOL) in the executable representation should correspond to the dynamic count of S_e.
b. Let S_c be the number of data items encountered whose location is not immediately determinable from the identifier; i.e., whose location must in some way be interpreted or computed (e.g., an array element or complex data structure). Ideally, there should be no more than S_c such items encountered in the execution of a program. Herein lies somewhat of a dilemma, since there is frequently a confusion between the representation of an item (its simple identification) and the implicit computation of its *location* in the data store. For example, the first occurrence of an array element $A(i, j)$ is an instruction to the interpreter to compute the location of this data object. However, subsequent use of $A(i, j)$, presuming that i and j have not been changed, may be interpreted as a simple surrogate identifier for the already computed storage location. Insofar as these items can be detected during the translation process, such overly complex surrogate identifiers may be removed from the image program; however, there may still remain data-dependent branches, etc., that cause this sort of situation to arise dynamically. Thus, the determination of a minimum S_c requires careful analysis of the HLL program representation.
c. Let S_b be the number of control actions dynamically interpreted in the HLL. HLL sequencing verbs each correspond to a single control action, regardless of whether the predicate of the control action is satisfied or not. Naturally, the frequency of occurrence of control actions is of importance given an overlapped host machine. Ideally, the number of control-type instructions in the image architecture should be equal to S_b as determined from the HLL (special cases of individual HLL verbs might be allowed due to known skews in the distribution of potential operands, such as the default increment of one in FORTRAN and PL/I looping structures).

Stability measures are an extension of activity in that they measure more global types of activity: the number of environments encountered, number of computed locations, and number of potential disruptions to serial instruction sequencing.

• *Distance*

Regardless of the amount of resources available to the interpretive mechanism, execution time must be limited by the initialization required. Even though the host may have an infinitely large cache, the first occurrence of a block of image program or data is a cache miss. Thus, even a confluent machine with infinite resources is limited by first encounters, which we refer to as the *distance measure*. Three components of distance are identified in this paper:

a. Let D_e be the number of *unique* environments entered.
b. Let D_c be the number of *unique* objects requiring interpretive definition (computed addresses).
c. Let D_b be the number of *unique* branch targets.

As the environment is first entered it becomes captured by a host with infinite resources; even with finite resources the principle of locality ensures a reasonable capture rate. As artifacts of an environment are subsequently encountered during the execution of an image program, they become immediately available to the interpretive mechanism. Thus, distance is indeed an ultimate lower bound on execution time.

A set of canonic interpretive measures

Summarizing the above discussion, a given program representation can be measured for execution parameters independent of actual hardware. We propose the following measures of execution architectures:

Correspondence

The quadruple $\hat{A}_i$, A_i, $\hat{A}_d$, A_d.

Size

Each operator is of size $\lceil \log_2 (F) \rceil$.
Each operand is of size $\lceil \log_2 (V) \rceil$.

Referencing
Syllable references are measured by A_s.
Instruction references are measured by the pair $\langle M, A_i \rangle$.
Data references are measured by A_d.

Stability
The CI measure of stability is the triple (S_e, S_c, S_b).

Distance
The CI measure of distance is the triple (D_e, D_c, D_b).

The measures, while interesting in themselves, are of considerably more value when compared to actual architectures. We call these comparative ratios the *relative canonic measures.*

Ideal execution space

Space is measured by the number of bits needed to represent the static definition of a program.

$$\textit{ideal size} = \sum_{\alpha} \lceil \log_2 (F_\alpha) \rceil \times \hat{A}_{i\alpha} + \lceil \log_2 (V_\alpha) \rceil \times \hat{A}_{d\alpha},$$

where the summation is over all environments, α, by the HLL program semantics.

Ideal execution time

CI dynamic measures are closely related to the interpretive mechanism of an image machine. For the partially mapped host, interpretation time is almost completely dominated by the program activity. If the host data paths do not at least match the image data widths, the host memory will saturate and limit execution performance. Naturally, given a universal host, where each image instruction is decoded syllable by syllable, stability and distance are poor estimates of actual interpretation time. Conversely, stability and distance are highly relevant given overlapped or confluent hosts, and size measures are of diminished importance.

In the following, we estimate the ideal execution time for various host organizations in terms of *execution cycles.* The number of cycles needed to interpret an image program is calculated as a weighted sum of the various measures defined above.

Partially mapped hosts We estimate the execution time for image machines with partially mapped interpretive mechanisms as

$$\textit{ideal cycles}_{\text{partially mapped}} \approx [a \times (M \times A_i)] + [b \times (N \times A_d)],$$

where a, b are relative weights for instruction interpretation and data accessing, respectively. M is the number of syllables per instruction multiplied by the number of cycles needed to decode a syllable and access the object it identifies. N is the relative data path size (i.e., the number of memory accesses required to retrieve or store a data item).

Well-mapped hosts Given a well-mapped host, execution time is best estimated by a straightforward linear sum of activity and environmental stability. We approximate this execution time as

$$\textit{ideal cycles}_{\text{well-mapped}} \approx a \times A_i + b \times A_d + c \times S_e.$$

Again, a, b, and c are arbitrary relative weights for transformational activity, data accessing, and environment change; in this case M and N are equal to 1. The additional term S_e is introduced to recognize that, for many programs, this may be an important factor—especially when the host is equipped with a cache. The effect of cache is to lower the b weighting factor by perhaps as much as a factor of 10, and indeed (for large caches) $c \times S_e$ may dominate $b \times A_d$. If the interpretive mechanism is a partially mapped machine, where M is on the order of 5–10, the relatively slow interpretation of image instructions ensures that the environment change time will be far less significant, hence our assignment of a zero weight to this factor in the previous equation.

Overlapped host As we move toward increasingly high-performance hosts, overlap of image instruction interpretation time increases, and activity $(a \times A_i + b \times A_d)$ is decreased in relative importance. However, for simple overlapped hosts, the branching stability is quite significant—perhaps even dominating transformational activity. Thus an execution time approximation for a simple overlapped host would be

$$\textit{ideal cycles}_{\text{overlapped}} \approx a_1 \times (A_i - S_b) + a_2 \times S_b + c \times S_c + d \times S_e.$$

where a_1 and a_2 are appropriate weights for nonbranching and branching instructions, respectively, and c and d are weights for name and environment stability. For such simple overlapped machines, the occurrence of a branch delays the interpretive mechanism until all preliminary operations (such as fetching and decoding) for the target are completed. Indeed, it would not be unusual for a_1 to be only one-tenth the magnitude of a_2—i.e., for it to take ten times longer to execute a branching instruction than to execute a nonbranching instruction, because data and instruction fetching activity can be overlapped while nonbranching instructions are being interpreted but not during or in the immediate vicinity of a branching instruction. The effect of name stability (S_c) may be significant in this case, although it can be ignored for simpler host organizations where each data item is serially interpreted and computed names probably represent a small increase in the overall execution time. On more powerful hosts, however, this factor can become significant—especially if there are no associative lookasides that can be exploited during address computation.

Confluent host The ultimate in high-performance hosts is a machine that has an interpretive mechanism that includes an

Table 1 Comparison for the example listing.

	370 optimized	*370 nonoptimized*	*CIM*
No. of instructions	15	19	6
Program size	368 bits	604 bits	30 bits
Data references	20	36	13
Instruction environments	1	1	1
Data environments	1	1	1 + 1

unbounded amount of interpretive support hardware. As mentioned earlier, such a host is still bounded in execution performance by distance parameters (the time required for first interpretations of various objects). Thus, it would have an execution time of

$$\textit{ideal cycles}_{\text{confluent}} = \max(\textit{ideal cycles}_{\text{overlapped}}, K \cdot D).$$

That is, its execution time is bounded by the maximum of either program activity and stability, or program distance where the constant K is a timing weight for the distance measures. The bound for the confluent processor is especially important since it forms an ultimate limit to program execution. Its dependence upon distance is particularly interesting, as the distance measure then limits execution performance for most familiar initial program representations. Of course, since significant variations in the distance measure of comparative architectures arise only in large program environments, it is a somewhat more difficult concept to evaluate.

Applying the measures: two examples

The following two examples illustrate the use of the CI measures, both as absolute measures and as comparative measures of traditional architectures. The first example is a very simple FORTRAN three-line program which illustrates the use of some of the simpler aspects of the measures. A detailed comparison with System 370 is also shown.

The second example, while still necessarily limited, is a more comprehensive use of the program. Comparisons are made for a number of traditional and language-oriented architectures.

• *Example 1: CI measures for a simple FORTRAN program*

The following three-line excerpt from a FORTRAN subroutine, taken from [9], illustrates the simpler CI measures:

1. $I = I + 1$.
2. $J = (J - 1) * I$.
3. $K = (J - 1) * (K - I)$.

Assume that I, J, and K are fullword (32-bit) integers whose initial values are stored in memory prior to entering the excerpt, and whose final values must be stored in memory for later use.

• *Canonic measure of the FORTRAN fragment*

Instruction count

Statement 1	1 instruction	(1 operator)
Statement 2	2 instructions	(2 operators)
Statement 3	3 instructions	(3 operators)
Total	6 instructions	(6 operators)

Identifier count

Statement 1	3 identifiers	(2 operand, 1 operator)
Statement 2	5 identifiers	(3 operand, 2 operator)
Statement 3	7 identifiers	(4 operand, 3 operator)
Total	15 identifiers	(9 operand, 6 operator)

Identifier size

Operation identifier size = $\lceil \log_2 4 \rceil = 2$ bits.
(operations are +, −, *, =)
Operand identifier size = $\lceil \log_2 4 \rceil = 2$ bits.
(operands are 1, I, J, K)

Program size

6	operator identifiers × 2 bits = 12 bits
9	operand identifiers × 2 bits = 18 bits
Total	30 bits

References

Instruction references—6 references
Data references—13 references

Stability

Instructions —1 environment
Data —1 environment load and environment store
—0 computed names

The following listing was produced on an IBM System 370 using an optimizing compiler (FORTRAN IV Level H, OPT = 2, run in a 500K partition on a Model 168, June 1977):

```
1. L   10,112(0,13)
   L   11,80(0,13)
   LR  3,11
   A   3,0(0,10)
   ST  3,0(10)
2. L   7,4(0,10)
   SR  7,11
   MR  6,3
   ST  7,4(0,10)
3. LR  4,7
   SR  4,3
   LCR 3,3
   A   3,8(0,10)
   MR  2,4
   ST  3,8(0,10)
```

A total of 368 bits are required to contain this program body (we exclude some 2000 bits of prologue/epilogue code required by the 370 Operating System and FORTRAN linkage conventions)—over 12 times the space indicated by the canonic measure. Computing reference activity in the same way as before, we find that 20 accesses to the process name space are required to evaluate the 370 representation—allowing one access for each 32-bit word in the instruction stream.

The increase in program size, number of instructions, and number of memory references is a direct result of the partitioned name space, indirect operand identification, and restricted instruction formats of the 370 architecture.

Table 1 illustrates the use of ratios for the foregoing example.

• *Example 2: CI measures for a more complex Pascal program*

Consider the Pascal example shown in **Figure 6**; *hardshuffle*—a program for shuffling array elements the hard way—which consists of shuffle procedure *swapvec* and the main program. *Swapvec* interchanges the elements of two arrays from the first element up to the parameter *limit*. *Hardshuffle*—the main program—creates two arrays: *identity*, consisting of the integers, and *sum*, which consists of the sum of the integers. For a variable *limit* ranging from *1* to *10*, *swapvec* is called to interchange some of the elements of the two arrays. Finally, the values of the arrays are written out.

Figure 7 is an evaluation for the example *hardshuffle* for the CI measures on a variety of architectural approaches. A fair comparison for a variety of architectures is a more formidable task than might first appear. The measures are significantly influenced by compiler strategies and run time environments as well as the basic architecture itself. Thus, the data in Fig. 7 require some explanation.

The first comparison is with an execution architecture called Adept [10], developed at Stanford and derived from principles of minimizing CI measures while maintaining transparency for Pascal programs. By using an additional format syllable in each instruction it matches most static and dynamic CI instruction count measures. It also matches the CI measures for memory activity. Additional syllables per instruction add about 5 bits to each instruction and hence account for about 110 additional bits in static program size. An additional 800 bits of Adept are used to hold constant values, array bases, and other environmental data. An Adept variable reference consists of the addition of an environmental pointer to a variable index whose container matches the $\log_2$ CI requirement. Each environment then has its own environmental pointer and container width. Some variables such as array elements have an address computation before the element can be retrieved from main storage, which contains the image array. Thus the address of the base of the array must be stored as well as the retrieved array element. The additional Adept space includes these address constants and other values containing information

```
program hardshuffle(output);
  {Shuffles two arrays the hard way}
  type indextype = 1..10;
    vector = array [indextype] of integer;
  var identity, sum: vector;
    i: indextype;

  procedure swapvec(var a1, a2: vector; limit: indextype);
    var index: indextype;
      temp: integer;
    begin
    for index := 1 to limit do begin
      temp := a1[index];
      a1[index] := a2[index];
      a2[index] := temp
    end {for}
  end {swapvec};

begin {hardshuffle}
identity[1] := 1;
sum[1] := 1;
for i := 2 to 10 do begin
  identity[i] := i;
  sum[i] := sum[i-1] + i
end {for};
for i := 1 to 10 do swapvec(identity, sum, i);
for i := 1 to 10 do begin
  write(identity[i]);
  write(sum[i]);
  writeln
end {for}
end {hardshuffle}.
```

Figure 6

Hardshuffle program.

	Pascal				System 370 Pascal		System 370 PL/I	
	CIM	Adept	PDP 11	P-Code	with lnkge	w.o. lnkge	with lnkge	w.o. lnkge
Static size (in bits)	277	1184	2800	4960	3056	4288	3668	4952
Static number of instructions	27	27	105	155	90	99	130	168
Dynamic number of instructions	435	435	2023	2404	1481	1536	2322	2522
Total no. of data references (local + global/computed)	830							
No. of main memory (read) refr.	139	139	963	2797	443	486	1450	1626
No. of main memory (write) refr.	130	130	479	2179	402	402	457	503
No. of syllables interpreted	1298	1923	10115	9976	7393	7668	11318	13198
No. of branches (dynamic)	108	108	230	202	219	261	188	211

Figure 7

Comparison for various architectures with CIM.

for the routine to execute properly. The object code for Adept is based upon a 1 + 1 pass compiler [10].

The pdp-11 (a trademark of the Digital Equipment Corporation) figures are based upon the Pascal compiler developed at Vrije University (the Netherlands), Pascal-VU. It produces an intermediate program representation, EM-1, developed by A. Tannenbaum [11], which is further translated into pdp-11 code. The static program size is the size of the instruction stream; however, in the pdp-11 architecture many of the data parameters are represented as immediate data in the instruction stream.

The P-code machine is actually a surrogate for the Pascal language. It is a stack-oriented machine and is meant to be a transportable medium for Pascal programs. Any host machine can compile into P-code from a Pascal source and (in theory at least) another machine equipped with a P-code interpreter can execute this compiled code. The emphasis for most P-code compilers is rapid compilation; thus the P-code statistics are derived from a nonoptimized compiler—much the same in philosophy as Adept.

The large number of dynamic instruction occurrences for P-code when compared to Adept—over 5 to 1—is largely accounted for by the *push* and *pop* instructions inherent in a stack machine. Notice that the dynamic number of P-code branches, for example, is less than a factor of two to one over the CI measure.

In comparing the System 370 to any of the other environments one is faced with immense problems. So far we have been discussing machines and measures in very limited run-time environments with relatively minimal generality in support for nonPascal system facilities, the antithesis of the generalized support provided by the 370 Operating System. While the 370 program size itself is 3056 bits, this *excludes* prologue, epilogue, and data space which alone—through a standard interface—is reserved at 16 000 bits. This overwhelms our comparison and since it contains or allows for a great deal more information handling and communications than required either in this program or by any of the other architectures, we eliminate (insofar as possible) instructions or data areas which are not specifically associated with the program *hardshuffle*. The column labeled "without linkage" represents the additional number of instructions in the minimum linkage path between the two routines. Excluded from this are the instructions executed as part of the linkage which are calls to common run time facilities, space allocation, etc. These are again excluded in our comparison since it seemed to us that the inclusion of such data is more a measure of run-time philosophy and its generality than a measure of architecture itself. Calls to such facilities during routine entry are not counted in the environment counts either. To fully include all instructions executed in a typical System/370 program plus all data areas and prologue and epilogue areas would increase the cited numbers by several times. Thus, the 370 numbers can be interpreted as minimum numbers in comparing with the other architectural figures. The 370 numbers reflect an estimate of the measures of the architecture in a very simple dedicated run-time environment which simply is not available to us to measure. As a further experiment on 370 the *hardshuffle* source program was recoded in PL/I and recompiled using an optimizing PL/I compiler. The increased generality of PL/I plays a role in limiting the compiler's ability to optimize the program.

It is interesting to note that, at least for this example, the more dramatic variations in architectural measures occur in measures—such as space, dynamic instruction count, and syllables interpreted—that affect simpler hosts, particularly partially mapped and well-mapped machines. In fact, compilers seem to play a more significant role than the architectural arrangements themselves. This supports the observation that is more or less a truism that compiler technology is even more important than the architecture as the interpreter and executor technology is enhanced, while for simpler interpreters (hosts) the architecture seems to play a dominant role in determining execution performance.

Conclusions

Traditional computer architectures (i.e., program representations) are created about objects, actions, and/or capabilities presumed to be present in a physical host computer—thus simplifying the interpretation process. This is done, however, at the expense of compilation, storage space requirement, and number of items to be interpreted.

An alternative is presented, created about the notion of an architecture in close correspondence to the high-level language that was used to originally represent the program. Various possibilities can be considered as candidates for the "ideal" architectural form characterized by the canonic interpretive measures, depending on host resources.

Traditional architectures are significantly inferior to CI measures (by a factor of 3 to 10), while architectures specifically designed to attain these measures are able to come rather close (within 1.3) to them in a number of examples studied.

Acknowledgment

The research described herein was supported in part by the U. S. Army Research Office–Durham under Contract DAAG29-82-K-0109, using emulation facilities supported by NASA under Contract NAGW 419.

References

1. Yaohan Chu, Ed., *High Level Language Computer Architecture*, Academic Press, Inc., New York, 1975.
2. Michael J. Flynn, "Trends and Problems in Computer Organizations," *Proceedings of the IFIPS Congress*, Stockholm, Sweden, August 1974, North-Holland Publishing Company, New Amsterdam, 1975, pp. 2–10.
3. W. M. McKeeman, "Language Directed Computer Design," *Proc. Fall Joint Computer Conf.* **31**, 413–417 (Fall 1967).

4. Ravi Sethi and Jeffery D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. ACM* **17,** No. 4, 715–728 (October 1970).
5. Michael J. Flynn, "The Interpretive Interface: Resources and Program Representation in Computer Organization," *Proceedings of the Symposium on High Speed Computers and Algorithm Organization*, University of Illinois, Champaign, IL (Pub. Academic Press Inc., New York), April 1977.
6. Lee W. Hoevel and Michael J. Flynn, "The Structure of Directly Executed Languages: A New Theory of Interpretive System Support," *Technical Report No. 130*, Computer Systems Laboratory, Stanford University, Stanford, CA, March 1977.
7. Eric C. R. Hehner, "Information Content of Programs and Operation Encoding," *J. ACM* **24,** No. 2, 290–297 (April 1977).
8. D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes" *Trans. IRE* **40,** No. 9, 1098–1101 (September 1952).
9. M. J. Flynn and L. W. Hoevel, "Execution Architecture: The DELtran Experiment," *IEEE Trans. Computers* **C-32,** No. 2, 76–174 (February 1983).
10. Scott P. Wakefield, "Studies in Execution Architecture," Ph.D. Thesis, Stanford University, Stanford, CA, 1982.
11. Andrew S. Tannenbaum, "Implications of Structured Programming for Machine Architecture," *Commun. ACM* **21,** No. 3, 237–246 (March 1978).

Received July 28, 1983; revised March 3, 1984

Michael J. Flynn *Stanford University, Computer Systems Laboratory, Electrical Engineering Department, Stanford, California 94305.* Professor Flynn received his B.S. in electrical engineering from Manhattan College in 1955, his M.S. from Syracuse University in 1960, and his Ph.D. from Purdue University in 1961. He joined IBM in 1955 and worked for ten years in the areas of computer organization and design. He was design manager of prototype versions of the IBM 7090 and 7094/11, and later was design manager for the System/360 Model 91 central processing unit. Professor Flynn was a faculty member of Northwestern University from 1966 to 1970 and of the Johns Hopkins University from 1970 to 1974. In 1973–74 he was on leave from Johns Hopkins to serve as Vice President of Palyn Associates, Inc.—a computer design firm in San Jose, California, where he is now a senior consultant. Since 1975 he has been Professor of Electrical Engineering at Stanford University, and was Director of the Computer Systems Laboratory from 1977 to 1983. Dr. Flynn has served on the IEEE Computer Society's Board of Governors and as Associate Editor of the *IEEE Transactions on Computers*. He was founding chairman of both the ACM Special Interest Group on Computer Architecture and the IEEE Computer Society's Technical Committee on Computer Architecture.

Lee W. Hoevel *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Hoevel joined IBM in 1978 as a Research staff member at the Thomas J. Watson Research Center. He is currently a member of the experimental systems structure group and is involved in the analysis of cache performance and the design of system extension mechanisms. Dr. Hoevel completed his undergraduate work at Rice University, Houston, Texas, in 1968, and later received a Ph.D. in electrical engineering from the Johns Hopkins University, Baltimore, Maryland, while a Research Associate at Stanford University. Dr. Hoevel is a member of the Association for Computing Machinery, the Institute of Electrical and Electronics Engineers, and Sigma Xi.