Application System/400 performance characteristics

by B. E. Clark
M. J. Corrigan

The operating system for Application System/400™ (AS/400™) provides an unprecedented breadth of function and system services in a single, integrated system. The majority of functions are implemented on top of an abstract, high-level machine interface in a hardware-independent manner, using many architectural characteristics normally associated with poor performance. Despite these architectural and functional traits of the operating system, the AS/400 exhibits excellent price and performance characteristics for commercial applications and is a competitive system in the mid-range commercial application arena. A number of design and optimization techniques, many of them unique or innovative, were incorporated into the AS/400 to achieve a combination of advanced design, function, and performance and are the main subjects discussed in this paper.

Many of the basic architectural characteristics of the hardware and operating system of Application System/400™ (AS/400™) originated with the System/38, one of its predecessors. Some of the basic system objectives and requirements underlying the design of the System/38 included: hardware independence for the operating system, enhanced productivity for system and application programmers, optimization of system for interactive processing, greater integrity and reliability for interactive processing, major usability improvement over predecessor systems, extendability for the operating system and its applications, and leading-edge commercial application support.

The System/38 requirements applied equally well (if not more so) to the development of the AS/400 family of computers. In addition, several major objectives also existed for the AS/400, including compatibility with System/36, System/38, and Systems Application Architecture; a selection of products ranging from the size of System/36 to double the size of System/38; improved personal computer affinity via seamless interfaces; and market leadership in communications.

Some of the key AS/400 architectural characteristics that were developed to support these objectives included:

- High-level, abstract machine interface (MI)
- Pervasive late binding
- Capability-based (object-oriented) operating system (Operating System/400™)
- Segment-based virtual addressing (hardware and licensed internal code)
- Relational database management system (RDBMS)
- High level of inherent operating system integrity and reliability

- Consistent interfaces to lower-level services
- Wide range of high-function primitives
- High-function program model (automatic and static storage initialization, exceptions, debug, trace, and so on)
- High degree of "fault tolerance" and "fault isolation" in the system software support
- Major application programming interfaces (APIs) of predecessors fully supported

Many of these characteristics are commonly associated with poor performance. In the AS/400, a number of hardware and software design and architectural approaches were used, often in a unique or innovative manner, to provide the benefits of these characteristics without incurring the performance overhead normally associated with them.

The hardware design features include tagged storage for pointers, high-function input/output processors (IOPs) to offload processing from the CPU, and high-function microcode primitives and services.

The software architectural features include single-level storage management and automatic utilization of all of main storage as a DASD (direct-access storage device) cache, high-function MI primitives and services, object-oriented architecture, a single, common code generator producing re-entrant programs, an integrated, natively supported System/36 execution environment, and cooperative processing (involving personal computers).

System structure overview. A review of AS/400 system structure and terminology is necessary prior to discussing specific AS/400 performance characteristics.

The hardware and licensed internal code implement an instruction set and multiprogramming primitives called the Internal Microprogrammed Interface (IMPI). The licensed internal code portion of the system is implemented using the IMPI instructions and contains the traditional operating system kernal-type functions (storage management, resource management, authority, low-level Systems Network Architecture [SNA] layers of I/O operations, and so on) as well as the basic object handlers that provide the foundation for object orientation of the operating system. See Reference 1 for more detail on the processor and IMPI design.

The licensed internal code implements a higher-level interface known as the MI. This MI instruction set, although giving the appearance of being directly executed, is compiled into IMPI instructions via a licensed internal code component known as the translator.

The operating system proper (Operating System/400, or OS/400™) is implemented on the MI layer and, in concert with the licensed internal code, contains all of the traditional operating system functions plus many services normally provided as separate products on other systems (communications, RDBMS, automatic configuration, performance data collection, and so on). OS/400 supports a free-format command language (CL) which can be either interpreted or compiled, extensive system displays and menus, and system services in support of both licensed programs (compilers, editors, office, programming workbench, and so on) and the largest inventory of commercial applications in the industry available at this stage in the life cycle of a system.

The IMPI instruction set is similar to the System/360–System/370 instruction set.

Figure 1 illustrates the system structure.

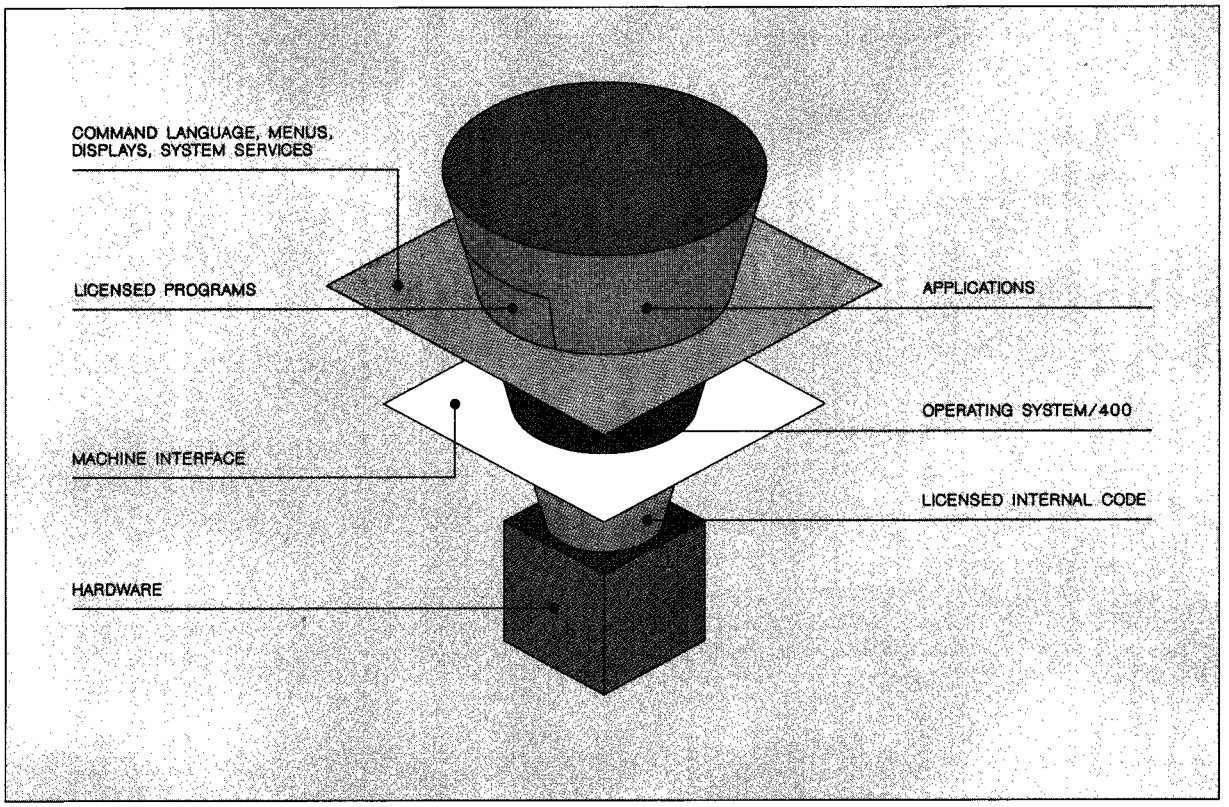
Basic hardware structure

The AS/400 family of computers is a system, made up of several processors, including the main processor, a service processor, one or more storage control processors, one or more local workstation processors, and optional communications processors. The storage control, local workstation, and communications processors offload functions from the main processor.

The AS/400 main processor hardware provides control storage, main storage, a set of internal registers, and an address translation unit.

The most highly used parts of the licensed internal code execute in the high-speed control storage,

Figure 1 System structure overview



whereas the rest of the licensed internal code executes IMPI instructions in main storage. The IMPI instruction set provides 16 general-purpose registers, a condition-code register, and an instruction-address register. This instruction set is used by the licensed internal code to implement the MI instruction set. The high-level MI instruction set is not interpreted but is translated by the licensed internal code to the IMPI instruction set before execution.

The IMPI instruction set is similar to the System/360–System/370 instruction set, but with many extensions. It provides one-, two-, four-, and six-byte registers with the ability to do arithmetic on one-, two-, and four-byte integers. It provides a binary floating point implementation and decimal arithmetic on integers up to 15 digits.

The IMPI has a large number of register-immediate and storage-immediate instructions. These instructions provide faster execution than their register-storage and storage-storage counterparts.

The IMPI provides instructions that allow a fast implementation of many of the MI instructions. The IMPI also provides many instructions to implement common sequences of more basic instructions. For example, there are test-and-branch instructions which can be used to test a bit and branch, depending on instruction contents.

Low-level system services

High-level IMPI instructions. The IMPI instruction set, made available by the hardware and licensed internal code, includes some functions which, on most machines, would be implemented from more primitive instructions by the operating system. Because the functions are implemented in AS/400 hardware and licensed internal code, they perform much faster than if built from primitive instructions. These functions include:

- Task dispatching—The IMPI provides a fast, prioritized, pre-emptive task dispatcher.

- Queuing—The IMPI provides a set of instructions and data structures that allow tasks to communicate via messages. The queuing functions are integrated with the task dispatching functions such that the receive message functions place a task in a wait state until an appropriate message is available on the queue. This allows the licensed internal code layer to be implemented as a multitasking, message-passing system.
- Serialization—The IMPI provides instructions that allow tasks to have a very fast serialization mechanism.
- Locking—The IMPI includes a set of instructions for the management of lock conflict. These instructions make available a fast hashing function for accessing symbolic locks and for automatic conflict detection.
- Data compression—The IMPI has a set of instructions that perform SNA and Multileaving Remote Job Entry (MRJE) data compression. These instructions perform the CPU-intensive compression algorithms much faster than the equivalent algorithm implemented by general-purpose, low-level IMPI instructions.
- Data scanning—The IMPI provides instructions that perform complex operations on character string data, including scanning for specific characters and trailing blank truncation.
- Array subscripting—In support of high-level languages, the IMPI provides a set of instructions that compute array element addresses from array indexes.
- Supervisor link—The IMPI provides a set of instructions used to route requests from user programs (MI programs) into the licensed internal code layer. These instructions automatically allocate a save area, save the registers of the process, and route execution to the proper function. A complementary instruction is used to restore registers, free the save area, and return to the user program.
- Implicit instructions—The IMPI provides that any unimplemented instructions will be executed as if they were supervisor link instructions. The licensed internal code can implement complex functions as if they were IMPI instructions.

The IMPI also provides an attribute bit for each quadword (16 bytes on a 16-byte boundary) within main storage. This bit is not addressable by the normal IMPI instructions used to address storage. The bit specifically identifies quadwords in storage containing MI pointers. MI pointers are addresses that MI programs may use and manipulate. MI programs have no direct access to the tag bit. The tag bit is turned on by the licensed internal code when a pointer is set and turned off by hardware anytime the quadword is modified (except through a controlled set of IMPI pointer manipulation instructions). This procedure allows the system to detect invalid pointers. It is not possible for an MI program to counterfeit a pointer or to modify a pointer in an invalid way.

The attribute bit implementation allows the validation of pointers in an extremely efficient way and is the basis for system integrity at the MI layer.

An error detected during the execution of an IMPI instruction is routed to the licensed internal code using the same technique used for the supervisor link instructions. The IMPI identifies many exceptional conditions in this way, allowing the licensed internal code layer to take appropriate action.

Index support. The licensed internal code layer implements a general balanced binary tree with front compression. The binary tree function is used extensively for fast, keyed information retrieval within the licensed internal code and OS/400 components. This implementation is highly optimized to minimize the number of disk operations required to retrieve an entry. The tree is balanced at a page level, providing a very broad, short tree.

Binary tree indexes are used within the licensed internal code by:

- Storage management, for permanent, temporary, and free-space directories
- Database, for indexed file support
- Libraries, for object name to address resolution
- Security, as a fast mechanism to check a user's authority to perform object operations
- Event management, to provide a fast way for finding processes that act as monitors for specific events

The binary tree function is also made available to OS/400 with support for an MI object, an index, which contains a binary tree. This is used within OS/400 for:

- Message files, so that the text of a message can be found quickly
- Job queues, so that jobs may be ordered by priority and status
- Output queues, so that spool files may be ordered by priority and status

Measurements on customer systems with heavy database applications show 10 to 15 percent of the total CPU being used by the index support code. See Reference 2 for more information on the implementation of the binary tree function.

Storage management. The AS/400 hardware and licensed internal code provide a "single-level storage" addressing architecture. A better term might be "uniform addressable storage." As objects (files, programs, control blocks, directories, and so on) are created, they are allocated disk space and are assigned a range of virtual addresses. These virtual addresses are used by the IMPI instructions to address the object data directly. The storage management licensed internal code reads the object data from disk into main storage on demand, as required by instruction access. This is known as "demand paging." Essentially all of main storage is used as a cache for objects stored on disk.

Auxiliary storage management uses a binary buddy system to manage free disk space.

Storage management is divided into two parts: auxiliary storage management and main storage management. Auxiliary storage management allocates disk space to objects, whereas main storage management handles the demand paging.

Figure 2 shows the following relationships:

- Auxiliary storage management assigns disk space to the virtual addresses of an object.
- Main storage management moves the pages of an object between disk storage and main storage.
- The CPU addressing hardware translates the virtual address of an object into the corresponding main storage address.

Auxiliary storage management. Auxiliary storage management uses a binary buddy system to manage free disk space. The binary buddy system only allows disk space blocks (extents) whose sizes are a power of two. Thus, one sector, two sectors, four sectors, eight sectors, and so on, are valid free-space block sizes. This scheme has several performance advantages:

- Garbage collection (the recombination of small blocks of free space into larger blocks) is very simple and fast. When a disk block is freed, a simple check can be made to see if its "buddy" is also free. If it is, the two buddies are combined, and the process is repeated until no buddy is found.
- External free-space fragmentation is nearly eliminated in most real-world cases.

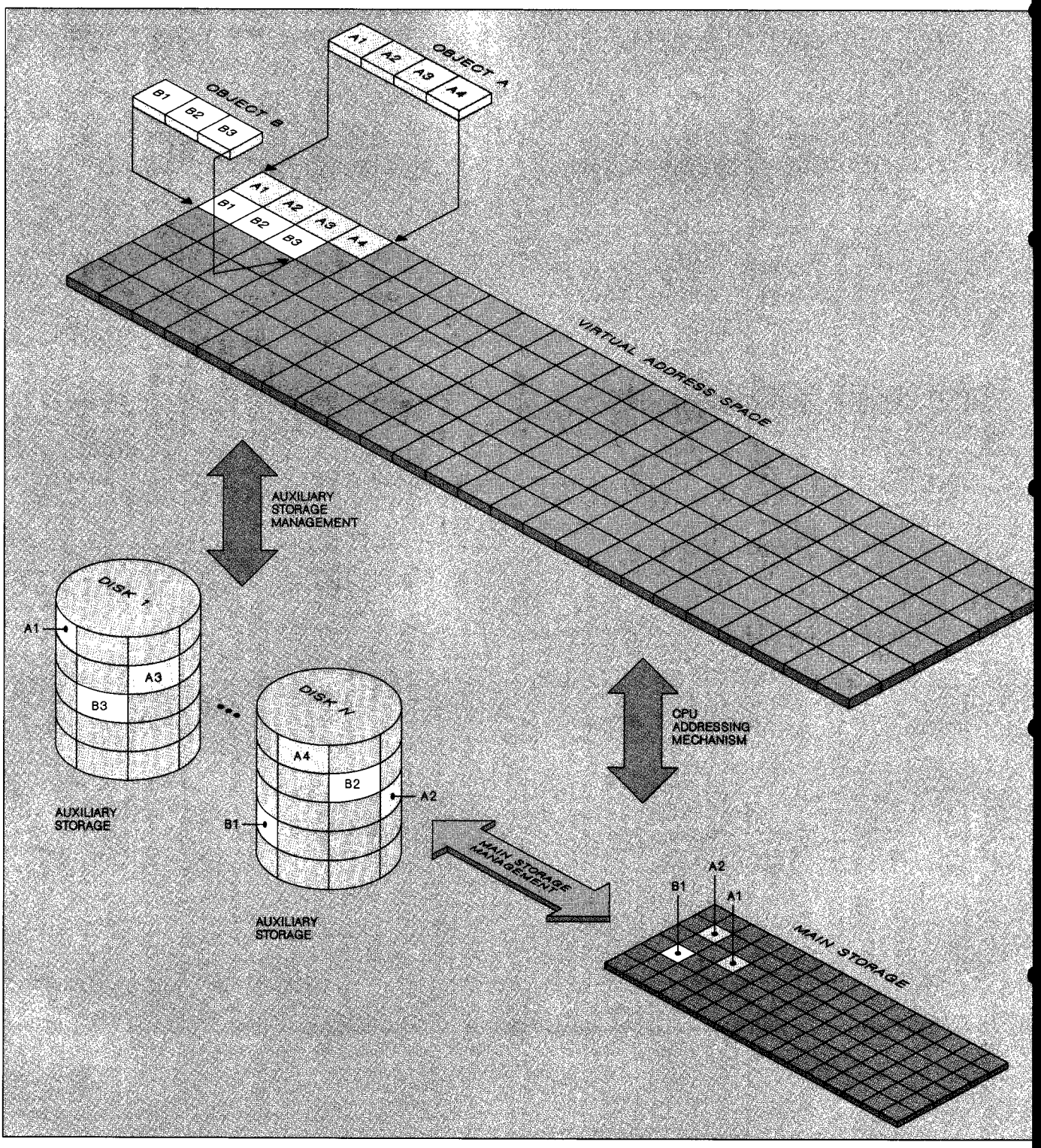
Auxiliary storage management uses binary tree indexes to maintain allocated and free-disk-space directories. These indexes are organized so that most operations (allocation, deallocation, and translation between virtual addresses and disk addresses) can be performed with a single index operation.

Auxiliary storage management uses one of two techniques to select the disk unit (actuator) from which the space will be allocated. If the request is small (less than or equal to 32K bytes), a randomized round-robin scheme is used. If the request is large, the disk unit with the greatest percentage of free space is selected. Data on the system is fairly well spread out among the disk units and provides reasonable disk-access balancing.

Storage management forces newly created objects to contain binary zeros on first reference. This action guarantees that a new object never contains old data from a deleted object that occupied the same disk space. No performance penalty occurs because the virtual address assigned to the object is stored in a "header" associated with each sector on disk. When a page of an object is read into main storage, its virtual address is compared with the address stored in the header. If they do not match, the contents of the disk sector are not part of this object and the page is "zeroed." This technique eliminates the need to "zero" disk space when it is allocated (or freed) or to maintain a large table containing an entry for each virtual page indicating whether it had ever been referenced.

Optionally, auxiliary storage management may divide the disk units into auxiliary storage pools (ASPs). Most user data (files, programs, and so on) are stored

Figure 2 Relationship of an object and storage



in the "system" ASP. Certain objects, such as journals and saved files, may be created and stored into other "user" ASPs. This process provides a physical separation between the active data files and the on-line backup, which improves performance by avoiding disk arm contention.

Main storage management. The basis for main storage management is a simple, demand paging scheme with an LRU (least recently used) page replacement algorithm. Performance would not be adequate with this simple approach in most environments. Main storage management provides functions that allow other components of the licensed internal code and the operating system to improve the paging performance of the machine. Some of these functions are:

- Requesting that large blocks of virtual pages are read into storage prior to any reference to them. This can be performed either synchronously with the requestor or asynchronously.
- Requesting that blocks of virtual pages are "cleared." This allocates zeroed pages of main storage to the virtual pages without doing any I/O operations.
- Identifying blocks of virtual pages not likely to be referenced in the near future. These pages are written to disk (if changed) and put at the head of the LRU list.
- Dividing main storage into "pools." A customer may divide main storage into pools. Each user and system task is assigned to one of the pools. All task paging requests are satisfied only from their assigned pool. In this way, the customer may ensure that a batch job, for example, will not steal the pages of a higher-priority interactive user.

The integrated database licensed internal code is highly optimized to reduce both I/O requests and main storage requirements.

When handling a request to read a virtual page into main storage, main storage management must determine the disk address assigned to the given virtual address. Determination is made by finding the entry in the binary tree index, which is built by auxiliary storage management. Main storage management maintains a "lookaside directory" of recently used virtual addresses, which can be examined very quickly. The index operation can be avoided by using the lookaside directory.

With the single-level addressing structure of the AS/400, main storage can be thought of as a cache for virtual storage. In this way, little within the system is sensitive to the main storage size. As more main storage is added to a system, the amount of disk I/O activity is reduced, since more data is automatically cached in main storage.

The access group gathers many small objects associated with a process into a few large blocks of disk space.

Access groups. Storage management uses an MI object, the access group, as a container where other objects may be suballocated. The access group gathers many small objects associated with a process into a few large blocks of disk space. When a process enters a long wait (for terminal response), its access group is written to disk in the fewest possible I/O operations. The main storage pages are then placed at the top of the LRU list. When the process executes again, the pages of the objects in its access group (that were in main storage before the long wait) are read back into main storage.

If the demand for main storage pages is small, storage management determines dynamically that the access groups of a process need not be written to or read from disk at all. This determination is based on a number of factors which are dynamically monitored. These include: the general faulting rate in the pool, the number of pages of the access group which were still resident in the pool at the start of the last few transactions, the number of faults that occurred on the access group during the last transaction, and any simple patterns detected in the read and write decisions over the last few transactions (both for the pool and the specific access group). The amount of data and history gathering done is directly tied to the general faulting rate in the pool so that this overhead is also minimized as demand in the pool decreases. With this enhancement, response time for machines with a large amount of main storage is fast and the CPU resource is substantially reduced. Access group

swapping in a highly memory-constrained system can consume 30 percent of the CPU resource of the system. Swapping decreases to 1–2 percent of the CPU as the paging demand in the pool decreases.

Because main storage is used as a cache for virtual storage and has the ability to dynamically turn swapping on and off, there is a strong and direct relationship between main storage size and the amount of disk I/O operations required on a system. If the main storage size is increased, the amount of disk I/O operations decreases. The system shifts smoothly from an environment of heavy swapping and faulting, to one where I/O activity is required only for randomly accessed data when main storage is added.

See Reference 3 for more information on the implementation of storage management.

Resource and process management. Resource and process management are the licensed internal code components that control the execution of user and system tasks within the system.

Although the IMPI instructions supply a task dispatcher, its pre-emptive, priority scheduler is not adequate for a system with other resource constraints. For example, allowing all processes to compete for the CPU could quickly force the working set (the number of main storage pages required to run without excessive page faults) of the system to exceed the available main storage.

The process management component implements a scheduler, limiting the number of processes that may actively compete for pages in a storage pool to a number set for that pool by the user. An active process may become ineligible to compete when it has used a certain amount of CPU time, known as a *timeslice*. An active process that becomes ineligible or that reaches a long wait for terminal response has its access group "purged." When an access group is purged, any changed pages are written to disk, and the pages are forced to the top of the LRU list. This action amounts to swapping the process out. A process eligible to compete has its access group swapped in.

Relational database support

The key AS/400 system component, from the performance standpoint of commercial applications, is its relational database management system (RDBMS). Because of the performance-critical nature of this component, the majority of the run-time support and management of the RDBMS (including journaling and commitment control support) is implemented in the licensed internal code layer (below the MI layer). Run-time support is closely integrated with two other key performance areas of licensed internal code support, index support and storage management. Index support is heavily used to implement the logical views of the database in the most performance-efficient manner possible. See References 4, 5, and 6 for more detail on the RDBMS design and implementation.

Storage management services are extensively used by the RDBMS.

Storage management services are extensively used by the RDBMS to maximize and overlap disk I/O operations and minimize working set size. Anticipatory asynchronous reads and writes on database record segment pages and indexes are done based on expected or historical reference patterns. Blocking of multiple data pages to and from disk are done automatically when sequential processing patterns are detected or at the request of the application. Journals can also be placed in an auxiliary storage pool, separate from the rest of the system, to eliminate contention for the disk arm.

One important consequence of the single-level store as it relates to the database is the cost of ensuring that all changed pages associated with a file have been forced to disk when the file is deactivated (closed). Because of the implicit sharing (or caching) provided by main storage management, finding all changed pages of an object currently in memory requires either examining all of the pages in the main store or checking each page of the object to determine if it is in main store. This technique becomes prohibitively expensive as the size of the main store and object increases. On the System/38 Model 700 with 32M bytes of main store, this approach was consuming up to 30 percent of the system CPU in the RAMP-C™ benchmark. A bit-map technique was

implemented to resolve this problem so that at file close time a bit map associated with the file identifies which pages were modified, thus restricting the number of page examinations required.

Sophisticated search features, based on estimates made with incomplete information, or "guesstimates," of the number of selected keys in specified indexes, minimize processing time for dynamic queries. This key range "guesstimate" technique is unique to the System/38 and AS/400 in that it is done dynamically, without requiring any additional index management at update time or static key counting routines run at the user's request. See Reference 7 for a detailed description of this technique.

Implicit index sharing by multiple logical views is done when equivalent sequencing is specified in the logical view definition, avoiding the maintenance of multiple indexes at execution. Such sharing is particularly important on the AS/400 because of the serialization protocols currently used in the RDBMS. These protocols result in all of the indexes involved in an update being locked concurrently while the update is in progress. Therefore, the potential for contention on a file increases with the number of concurrently updated logical views over it. This potential can be a serious bottleneck on a large system with a heavily updated file that has a large number of logical views over it. This design will need to be changed to provide for more granular serialization as the system size and number of supported users grows.

Combining the characteristics of implicitly cached main storage, automatically balanced disk arm utilization, high-function horizontal IMPI primitives, and the low-level, integrated implementation of the RDBMS results in unusually good performance characteristics for a relational database. This result is a key contributor to the good price/performance characteristics of the AS/400.

Machine interface

Abstract machine. From a performance standpoint, perhaps the most important architectural feature of the AS/400 is the machine interface (MI) layer. The MI layer is an enforced boundary (a set of instructions) formally structured in accordance with the architecture between the licensed internal code layer of the system and the OS/400 layer. The MI instruction set, although giving the architectural appearance and function of direct execution, is actually compiled. The OS/400, and all code above it (licensed programs, applications), is implemented entirely on the MI. The MI instruction set can be categorized into several logical groupings: computational instructions; specific objects (over 15 different object types are supported); locking, exceptions, events; and machine resource observation and management.

All of the systems compilers are targeted to this MI instruction set, producing a "program template" which is then used as input to the MI instruction "Create Program." This process invokes a translator component in the licensed internal code layer that "translates" this program template into a program object containing an IMPI instruction stream. Generating the instruction stream involves normal code generation chores (such as performing register optimization, temporary operand management, and so on) followed by the final step of encapsulating all of the generated pieces into a new program object. A system pointer is returned to the program object, which can then be used as the operand of a call or transfer MI instruction.

MI instructions are characterized by being high-level, generic, and machine-independent. There is no concept of registers, physical storage locations, or other hardware-specific characteristics in the instruction syntax. For example, the computational instructions consist of generic arithmetic operations and string manipulation operations. To add two numbers together, a single add numeric instruction exists that accepts any combination of numeric operand types and precisions. At translate time, if the type and precision of the operand is known, an appropriate set of IMPI instructions is generated to perform the operation, performing type conversions and precision adjustments as required. If the operand attributes are not determined at translate time (i.e., late binding was used via data pointers), a Supervisor Link (SVL) to the appropriate licensed internal code routine is generated, performing the operation in an interpretive manner when executed.

Along with the traditional numeric and string manipulation instructions supported in the computational class, a number of higher-function instructions for performing common string-handling operations exist. Besides generalized versions of the System/370-like translate instructions, there are instructions in support of parsing (scanning for the occurrences of a particular string in another string or array) and string compression and decompression (MRJE and SNA flavors). Special support for double-byte character strings (DBCS for ideographic character sets)

is also provided in the scan instruction. Character string operands can be up to 32K bytes in length, and arrays of up to 16 megabytes are supported.

Each class of object supported by the MI layer has its own unique set of instructions appropriate for the class of object (i.e., a program object supports "create," "delete," "call," "transfer," and "materialize" instructions). In general, these instructions (at execution) result in an SVL operation to invoke the appropriate licensed internal code routine to perform the function. It is also true for most of the other instructions in the remaining two categories.

A dominant characteristic of the AS/400 is its object-oriented architecture.

A program object contains an instruction stream that is a mixture of:

- IMPI instructions, corresponding to early-bound computational MI instructions
- SVLs to licensed internal code routines, to perform more complex and late-bound operations, such as object management, database access, authorization management, and so on

This mixture results in a machine interface that is high-level, abstract, late-bound, and interpretive in nature. The machine interface is translated, however, into an instruction stream, where the performance-critical computational and string-handling operations are handled in line with compiled, early-bound performance characteristics (where possible). Furthermore, since there is a single translator for a single MI instruction set targeted by all compilers on the system, it is comparatively easy to enhance the IMPI support (i.e., to provide additional high-function primitives) and quickly take advantage of the enhancement because only one code generator must be modified.

See References 8 and 9 for more detail on the MI and on object-oriented architecture.

MI objects. A dominant characteristic of the AS/400, both externally (to the user) and internally (in the OS/400 design and implementation) is its object-oriented architecture.

The basic object handlers are implemented in the licensed internal code layer, providing the support for the set of objects at the MI. These objects are interfaced by the OS/400 and licensed programs (LPs) via the respective object-specific MI instructions. These MI objects present a set of common functions (via MI instructions) to all of the system code built on top of the MI layer, thus providing the benefits of improved integrity and reliability, functional and interface consistency, optimized performance, and reduced operating system code redundancy.

These benefits come from formally encapsulated function and data structures that are centralized, carefully implemented, and easily accessed. The structures are widely used throughout the operating system and LPs as basic building blocks for the functions and objects they provide. This formalized and rigidly enforced data abstraction model is a key contributor to the integrity, reliability, and usability characteristics of OS/400. It also contributes significantly to its performance characteristics by providing a highly shared implementation of common constructs which can then be highly optimized.

Several MI objects are used in support of the RDBMS of the system. These include cursor, data space, data space index, journal port, journal space, and commit block. These MI objects provide the basic, supporting building blocks for the OS/400 RDBMS.

Most of the fundamental areas of the functions of the operating system are supported through appropriate MI objects. Other objects that have a key influence on the performance of the system include contexts (libraries), user profiles (authorization), and programs.

Contexts and address resolution. The context object maps the symbolic identification (type and name) of an MI object to its virtual address. Above the MI layer, this virtual address is embodied in a 16-byte pointer, which can only be produced and manipulated through MI instructions (such as object creates and resolve pointer) that are designed in the architecture. Pointers are hardware-tagged so that they cannot be counterfeited or manipulated through interfaces not conforming to the architecture. Since pointers are the primary mechanism for identifying object oper-

ands on MI instructions, context objects serve as the mechanism for mapping the symbolic object identification, of an object provided by a user, to the virtual address needed to access the object on the system.

Context objects are used by OS/400 to support what is presented to the user as a library. A user-specified (and modifiable) list of libraries is associated with each job on the system, and objects can be referenced by the user explicitly qualified to a specific library. If not explicitly qualified to a library, the library list of the job resolves the reference by searching each library on the list in order until a matching entry is found. Context objects are implemented as indexes (keyed by object type and name) to provide optimum performance for this address resolution.

System authorization management is based on user profiles.

User applications refer to all of the objects making up an application symbolically, and everything is represented as an object in the system (including the user's job itself, over 40 different external object types are on the system). This representation combined with the late-bound nature of the system (no link-editing, late-bound calls, each CL command represented by an object, and so on) results in this address resolution operation occurring very frequently in the system, often accounting for 5 percent of the CPU usage in interactive applications.

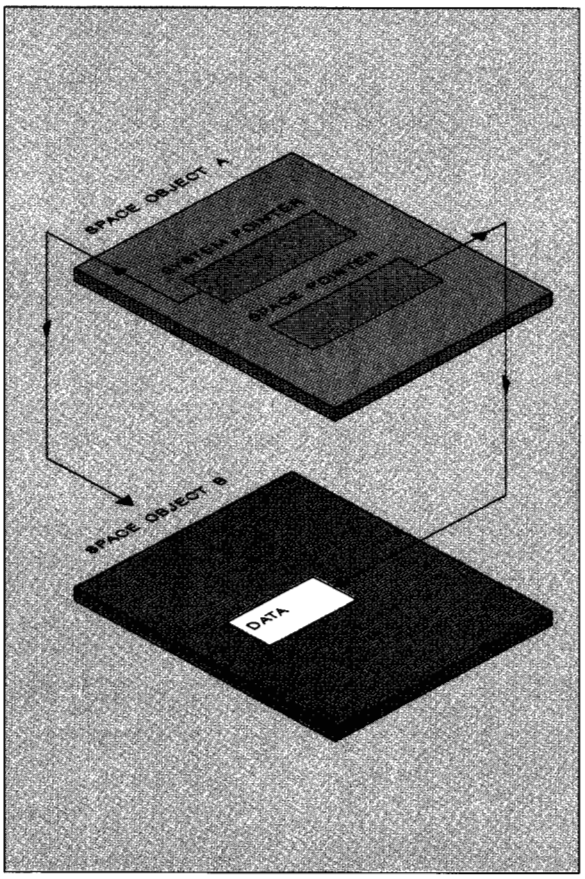
User profiles and authority management. System authorization management is based on user profiles. Each system user is represented by a user profile object, which serves as the repository for all authorization information related to that user. All objects created on the system are owned by a specific user, and authorization to use, modify, and manage that object (and the data within it, in some cases) can be controlled on an individual user basis. At creation time, the object is given a default level of access authority that applies to all users. The authority level can be overridden on an individual user basis to give that user more or less authority to access each object. Each operation or access to an object must be verified by the system to ensure the user's authority. This level of authority checking in combination with the granularity of objects typically used in an application (data and device files, programs, libraries, data areas, commands, spool files, data queues, device and controller descriptions, output queues, message queues, menus, and so on) implies the potential for a great deal of execution overhead, and a number of optimizations exist to minimize this overhead. The enhancements include:

- All object authority user profile attribute—When the attribute is present in the user profile attempting an operation, no further checking is required. This mechanism is used when the user configures the system to run without resource authorization checking. It can also be granted to selected profiles when resource authorization checking is active.
- Default authority in the object—The object default level of authority is stored in the object itself, along with a bit that specifies whether any specific (private) authorities have been granted to specific users. This default avoids doing any user profile lookup if no private authorities exist for the object.
- Pointer authority—A user's authority to access an object can be stored in a resolved object pointer as part of the address resolution operation. An example is the database file open processing, which performs an address resolution, storing authority in the pointer used for subsequent operations against the file (within the same job), and avoids authorization checking on the data accesses to the file.

A number of additional constructs exist for controlling object authorizations (such as group profiles, adopted authorities, and authorization lists). A complete authorization verification can result in several user profiles being accessed. The user profile object itself is implemented as an index (using the virtual address of the object as the key), thus providing optimum performance for random lookup operations when they do have to be made.

The most expensive part of this authority resolution are the index operations against the user profiles. These operations have been observed in some customer systems to be consuming 15 to 20 percent of the total processor when the various optimizations described above were disabled. The authority verification algorithm has been optimized to perform the checks in an ascending order of cost, attempting to

Figure 3 Space objects and space pointers



avoid the index operations if possible. For example, the authority in the pointer is checked first, next the user profile(s) is checked for all object authority, then the object is checked for no private authorities and sufficient default authority. This order typically results in less than 5 percent of the authority verifications performing an index operation (0 percent if resource authorization checking is not active).

See References 10 and 11 for detail on the authorization support.

Space objects. A space object is an MI object that is essentially a free-format byte string (up to 16 megabytes in length), which can be freely accessed and manipulated using MI computational instructions. Access to this byte string is gained through a special-purpose pointer called a space pointer (SPP). Figure 3 depicts space objects and space pointers. An SPP identifies the space object and an offset within it. The SPP can be used as the operand for many of the MI instructions. The offset within an SPP can be manipulated via specific MI instructions that are provided for this purpose. A high-performance form of a space pointer, called a machine space pointer (MSPP), is supported with limitations on its use, such that its actual storage location cannot be accessed directly from an MI program. The pointer can be optimized to and manipulated as a six-byte virtual address, potentially being optimized into a register across MI instructions, without compromising program debug support. A specific authority to access the object is required in order to set a space pointer (from a system pointer) to the space object, but once it has been initially set, its offset within the space can be manipulated without any authorization checking.

At the time the space object is created, 16 megabytes of address space are reserved for the object, with the actual disk allocations being made only upon explicit request or, optionally, automatically on first reference to an offset.

The space object provides a high-performance free-format construct for use when the frequency of reference or unpredictability of use would make more formal encapsulation of the object impractical. It often serves the function of "GETMAIN" type of support in more conventional systems without the space-management (chaining and so on) problems normally associated with these older mechanisms. Space objects are extensively used for control blocks within OS/400 as well as for many of the external objects (commands, job descriptions, menus, device files, data areas, and so on) presented to the user.

MI program architecture. All MI programs are reentrant—that is, the instruction stream and other constant execution entities are nonmodifiable and shared among multiple users. Only one copy ever exists in main storage, regardless of the number of concurrent users. Storage for program variables and other process-specific pieces of the program are allocated and managed in process-specific storage by the MI on appropriate Call and Return boundaries. (The external call is implemented as an SVL routine in the licensed internal code.) In addition to allocating and managing this storage in a manner consistent with its attributes (static or automatic), program variables can be initialized to specific values by the MI at the time the program is called, by specifying the initial values in the declarations of the variable. This feature, plus other services such as exception-

description management, invocation-tracing support, event management, and so on, provide a very rich, productive programming model at the MI level.

From a performance standpoint, this rich support can make external program calls expensive. The minimum path length is on the order of 60 instructions, with much longer path lengths being incurred, depending on features used (such as the number of variables initialized). Path length has not been a significant problem in the commercial application arena as it is characterized by large programs and relatively infrequent external calls.

Since MI programs are re-entrant (do not have to be loaded or relocated), they have their program variable storage automatically allocated (in separate segments) at call time and can be identified either symbolically (late-bound call) or by virtual address pointer (early-bound call). Since all other external references are resolved at execution time, there is no concept of a link loader at the MI level. Program "linkage" is dynamic, implicitly occurring at external call time. If the called program is symbolically identified, an implicit address resolution is performed using an explicitly specified context or an implicitly specified list of contexts (an address resolution list associated with the process). This resolution maps the symbolic program name to a virtual address. The address can optionally replace the symbolic specification (in the processes, program variable storage area) so that subsequent call executions do not incur the overhead of the address resolution. This option is commonly used in application programs to provide dynamic binding to the programs on the first call; then subsequent calls in the "run unit" of the language reuse the resolved address. Similar techniques are also applied to other external references by the program.

This linkage technique has been further refined for the OS/400 system code by building a "system entry point table" containing the addresses of all of the system programs (built at the time that the system code is installed). All external calls within the system code and from application code to system code are done via these preresolved pointers.

Similar techniques are heavily used within the OS/400 system code to early-bind other external references. Numerous control blocks and structures are built and initialized at different points in time (install, initial program load, job initiation, first use, and so on), binding external addressability at the most appropriate point based on functional and performance considerations and on tradeoffs. A vast majority of external references are early-bound without losing the flexibility of late binding. Late binding is still used freely when functional considerations make it desirable.

The MI supports a set of common program debug functions.

Program debug. The MI also supports a set of common program debug functions, including the ability to set breakpoints on MI instructions as well as displaying and modifying program variables while at a breakpoint. Breakpoint support is implemented through licensed internal code support and designates an address range within an instruction stream (specific to a process) where interrupts will be presented on instruction execution. This designation allows supporting breakpoints to be on the program anytime (in a process-specific manner), without incurring any extra overhead in the instruction stream when running without breakpoints being set.

The program variable display and modification support is provided via a table generated by the translator that maps program variables into their storage locations at execution. Currently this support is automatically provided, so a recompile is not needed to perform program debugging operations. To make this support as predictable as possible, the MI architecture guarantees that the storage locations associated with variables are always current at MI instruction boundaries (the only place where breakpoints are serviced) and that changes made to variables while at a breakpoint will be reflected immediately in the execution of the program. Ensuring this predictability places some constraints on register optimization. Although addresses are currently optimized into registers across MI instructions, data items are not. This restriction can result in poor performance for tightly coded loops where the loop control code and array index values cannot be optimized into registers. For the typical commercial application

environment, this condition is generally not a problem because of the existence of the high-function string manipulation of MI instructions, which usually eliminates the need for tightly coded loops at the MI. As newer languages and engineering and scientific languages (Pascal, C, FORTRAN) are supported on the system, this performance shortcoming of the MI may become more serious, requiring a relaxation of this aspect of the architecture as a program option.

Transaction processing model

The AS/400 IMPI supports a basic tasking model represented by a task dispatching element (512-byte memory-resident control block). The licensed internal code layer of the system combines this tasking model with several other constructs to provide an MI "process model." Constructs include:

- User profile
- Process access group
- Program variable storage—Program automatic storage area (PASA) and program static storage area (PSSA)
- MI exception-handling support
- Event-handling support
- Object-locking support

The OS/400 combines an MI process with additional structures and support to present a "job" to the user. The additional structures include:

- Job message Q
- Output Q
- QTEMP library
- Local data area
- MI response Q (I/O interface to the MI)
- Data management communications Q (manages file opens and dynamic file redirection)

All this system function, available in support of a user's "job," in combination with the previously discussed support (re-entrant programs, dynamic address resolution, storage management, RDBMS, and so on) results in a transaction processing model based in each user's job. This model results in a dramatically simplified, flexible, and dynamic application development environment. Application control flow is single thread and free of conventional resource bottleneck constraints that confront more conventional transaction processing environments. Each user's job contends for and accesses resources dynamically and independently of other users, using shared copies of the permanent objects in main storage (code, data, indexes, control blocks) and their own job-specific program variable and file-access buffer areas. Thus, the performance benefits of shared system resources are achieved without the drawbacks of restricted address space, complex, inflexible resource management problems, and rigid early-bound requirements which come from transaction processing models servicing multiple users under a single task.

System compatibility results in performance characteristics similar to native applications.

Execution environment support

AS/400 System/36 Environment. One of the major challenges in the development of AS/400 was providing a platform to support the execution of the System/36 application family with equivalent or improved price and performance. Given the radical differences in the architectures, designs, and heritages of the two systems, the conventional solution would have been to support an emulation mode (based on hardware) on the new system. This choice would have had the advantage of providing object code compatibility but would not have achieved the objective of immediately providing a wide range of new functions, productivity, and capacity to System/36 applications. An alternative solution was implemented, based on software.

The AS/400 System/36 Environment (S36E) provides source code compatibility for System/36 applications on the AS/400. Compatibility is accomplished by providing a "mapping" layer of support and structures in OS/400 to map the System/36 Application Programming Interfaces (APIs) to the underlying native support in OS/400. As a result, the S36E is an integrated extension to OS/400 rather than an emulator or a "mode." There is no concept of "hot keying" between the environments. Applications running in the S36E share the same system facilities and support that an AS/400 application does. The S36E

language compilers generate code that runs directly on the AS/400 hardware, and the System/36 command language invokes the appropriate OS/400 services directly. The database, spool, security, message handler, display facilities, and so on used by applications running in the S36E are the same as and are fully shared with the native AS/400 applications. See Reference 12 for a more complete description of the System/36 Environment design.

System compatibility results in performance characteristics similar to native applications. Although there is some performance overhead incurred in mapping some System/36 functions to the appropriate native services, these functions are generally in the 5 to 15 percent range. When a migrated System/36 application does experience significantly degraded performance (compared to the equivalent-sized System/36), it is usually caused by the design of the application. That is, it is making unusually heavy use of a system service, which is significantly more expensive on the AS/400 than it was on the System/36.

For example, the creation and deletion of a file on the System/36 is relatively cheap (fast) since it primarily involves a Volume Table of Contents (VTOC) update (the System/36 had a simple flat file system). On AS/400, all files are part of a full-function RDBMS, and the creation of one file involves creating and linking a number of complex control blocks as well as the updating of the data dictionary. Creation and deletion of a database file on AS/400 is much more costly (and slow) than on System/36. However, a System/36 application executing in the S36E on AS/400 is using a full-function RDBMS file instead of the limited-function flat file on the System/36, making much of the function (and performance) of the integrated RDBMS immediately available to the System/36 application.

System/36 Environment applications that make heavy use of those system functions which perform comparatively poorly on the AS/400 have been observed to sometimes require 50 percent more system resource than they did on the System/36 and may have to be modified (usually in relatively simple ways) to achieve acceptable performance. See Reference 13 for a more detailed description.

Save/restore (backup and recovery)

One implication of the auxiliary storage management scheme of the AS/400 (distributing the disk extents associated with an object across multiple DASD units) is that a simple sector-by-sector copy of the contents of a single device to a backup medium is of no value in the event of a future device failure. Since any single device, in general, contains only some of the pieces of any specific object, a backup of those pieces is out of synchronization with the other pieces residing on other devices. Short of doing a sector image backup of all of the relevant DASD on the system and then restoring all of these devices (essentially reloading the entire system), a sector image backup has little value. Thus the system save and restore strategy is based on a higher-level, object-oriented premise: essentially collecting and copying complete images of objects to the backup media on the basis of an object, group of objects, library, or group of libraries. This process is clearly more complex, requiring significantly more system processing for organization and management, particularly for complex database networks where many files (physical and logical views) may be interconnected so that they must be backed up together. For smaller objects, the result is that significantly more disk I/O activity is required since smaller disk I/O operations must be used to collect the small extents associated with these objects. Other object-related information, such as authorizations which are not physically stored with the object but must be recoverable, also add complication to this kind of a scheme.

To maximize the save and restore processing performance, a number of different strategies and support have been developed both to improve the performance of the processing itself and to reduce the volume of objects that must be backed up. The save and restore design employs extensive multitasking and main storage buffering, achieving the maximum possible amount of concurrent disk I/O operations and overlap with media I/O activity. The multitasking and buffering can be easily restricted by tuning parameters (CPU priority and buffer sizes which directly affect the amount of concurrent disk I/O operations) so that the impact of this activity on the rest of the system can be controlled when running in a nondedicated environment.

To reduce the volume of data to be backed up, the system supports a save changed objects scheme, whereby only those objects that have changed in a library (since the last time the entire library was backed up) are saved. Database files being journaled can be exempted from this procedure since a journal save achieves the same result (in less time if the file is large and the activity comparatively low).

The system also supports the concept of a save file. This file, residing on DASD, is a simulated tape file which can be used as a substitute for removable media on save and restore operations. If the save file is placed in an auxiliary storage pool (ASP) separate from the rest of the system, it provides the following benefits:

- Operatorless backup. For example, unattended backup overnight.
- Improved performance; i.e., a simulated tape device that runs at DASD speed.
- Improved flexibility. The backup can be done unattended when the system and objects are not in use, then optionally copied with low overhead (at device speed) to removable media during prime shift without interfering with normal operations and use of the objects. Or, if the save file is in a separate ASP, it can be left on line. If a disk unit in the system ASP is lost, the system ASP can be reloaded (after appropriate repair actions). Then the ASP containing the save file can be logically reattached to the system and used as the source for restore operation as appropriate.

Checksums. Probably the most innovative feature of the AS/400 system in this area is the facility known as checksums. This facility provides data redundancy on the DASD of the system using an exclusive ORing technique such that the contents of any disk drive on the system can be reconstructed from the contents of several other disk drives (from three to seven, depending on the systems configuration). Although the implementation of the concept on the AS/400 does not allow continued operation of the system while a disk device is inoperable, it does provide data recovery characteristics similar to DASD mirroring at a fraction of the DASD cost (13 to 33 percent additional DASD required, depending on the configuration). Although there is a CPU cost for the support (about 5 to 10 percent for interactive workloads) and an increase in disk I/O operations (about 25 percent for interactive workloads), it provides a cost-effective solution for many users desiring "no data loss" from DASD failure characteristic to the system.

The checksumming concept that was implemented implies that for every write of changed data to disk, the corresponding data locations on all of the other DASD in the checksum set must be read into memory and a checksum calculated and then written out to the checksum disk. Three key optimizations were adopted in the AS/400 implementation of checksumming which allow its performance to be acceptable.

First, when a changed page is written to DASD, the old data in the location is read into memory along with the old version of the checksum for that data. By exclusive ORing of the new data, old data, and old checksum, the new checksum value can be derived. This method avoids having to read all of the DASD locations that correspond to the checksum, reducing the required disk I/O operations from N (where N is the number of DASDs in the checksum set) to four when writing changed data to disk.

Storage on the system is segregated into two classes.

Second, the checksum data for a checksum set is spread evenly across the DASDs in a checksum set, thus spreading the I/O activity required to maintain it evenly among all of the members of the set and avoiding over-utilization of one DASD arm in the set.

Third, the storage on the system is segregated into two classes: temporary objects, whose existence does not span IPLs, and permanent objects. Since the temporary objects normally represent 5 percent or less of the DASD space on a system but account for 40 to 60 percent of the DASD writes on a typical customer system, segregating these two classes of storage and providing the checksum protection only for the permanent objects significantly reduces the number of DASD operations that incur checksumming overhead. The negative implication of this operation is that the system cannot continue to run when a DASD fails, as the portions of temporary objects stored on that device are no longer available and cannot be recovered. System operation cannot be resumed until the failing device has been repaired (and the permanent data on it reconstructed if lost during the repair action).

For systems that are not CPU-bound and which add an appropriate amount of DASD and/or main mem-

ory (adding memory almost always results in a significant reduction in total disk I/O activity), interactive performance with the checksum support active is usually equivalent to that prior to activating checksums (and adding the appropriate hardware). Very disk-write-intensive batch performance can degrade significantly, in some extreme cases by as much as a factor of three. This performance can usually be improved by fixing problems in the application such as blocking factors or changing file placement to get better overlap between DASD controllers or adding DASD controllers/buses. See Reference 14 for more details on this support.

Concluding remarks

The architecture of AS/400 is characterized by a number of features normally associated with poor performance, such as a hardware-independent operating system, a relational database, pervasive late binding, and a broad range of functions.

However, through extensive use of techniques such as low-level implementations of highly used primitives, an innovative storage management system, careful scoping of early- and late-bound features based on function and performance tradeoffs, and many other optimization techniques, the AS/400 exhibits competitive price and performance characteristics in the commercial application (as typified by the RAMP-C benchmark) marketplace.

Cited references

1. "System processor architecture," *IBM Application System/400 Technology*, SA21-9540-0, IBM Corporation (1988); available through IBM branch offices.
2. "System/38 machine indexing support," *IBM SYSTEM/38 Technical Developments*, ISBN O-933186-03-7, IBM Corporation (1978); available through IBM branch offices.
3. "System/38 machine storage management," *IBM SYSTEM/38 Technical Developments*, ISBN O-933186-03-7, IBM Corporation (1978); available through IBM branch offices.
4. "System/38 machine data base support," *IBM SYSTEM/38 Technical Developments*, ISBN O-933186-03-7, IBM Corporation (1978); available through IBM branch offices.
5. "System/38 data base concepts," *IBM SYSTEM/38 Technical Developments*, ISBN O-933186-03-7, IBM Corporation (1978); available through IBM branch offices.
6. "An integrated data base," *IBM Application System/400 Technology*, SA21-9540-0, IBM Corporation (1988); available through IBM branch offices.
7. *Index Key Range Estimator*, U.S. Patent 4,774,657, IBM Corporation (September 27, 1988).
8. "System/38—A high-level machine," *IBM SYSTEM/38 Technical Developments*, ISBN O-933186-03-7, IBM Corporation (1978); available through IBM branch offices.
9. "System/38 object-oriented architecture," *IBM SYSTEM/38 Technical Developments*, ISBN O-933186-03-7, IBM Corporation (1978); available through IBM branch offices.
10. "System/38 addressing and authorization," *IBM SYSTEM/38 Technical Developments*, ISBN O-933186-03-7, IBM Corporation (1978); available through IBM branch offices.
11. "Security," *IBM Application System/400 Technology*, SA21-9540-0, IBM Corporation (1988); available through IBM branch offices.
12. "The System/36 Environment," *IBM Application System/400 Technology*, SA21-9540-0, IBM Corporation (1988); available through IBM branch offices.
13. "Improving S/36 Environment performance," *News 3X/400* (March 1988).
14. *Parity Spreading to Enhance Storage Access*, U.S. Patent 4,761,785, IBM Corporation (August 2, 1988).

Brian E. Clark *IBM Application Business Systems, Highway 52 & NW 37th Street, Rochester, Minnesota 55901.* Mr. Clark is a senior programmer in system design control for OS/400 in the Rochester Programming Center. He received an M.S. in computer science from New Mexico State University in 1976. He joined IBM at Rochester in 1977 and participated in the development of the System/38. Since then he has worked on development, design, and performance for the System/38 and AS/400. He was a member of the original design and development team for AS/400 36 Environment support. Mr. Clark received several Division Awards and Outstanding Technical Achievement Awards for work on System/38 and AS/400. In June 1989, he was the recipient of a Corporate Award for his work on the AS/400 software design.

Michael J. Corrigan *IBM Application Business Systems, Highway 52 & NW 37th Street, Rochester, Minnesota 55901.* Mr. Corrigan is an advisory programmer in data management design control for AS/400 in the Rochester Programming Center. In 1974, he received a B.S. in physics from the University of Notre Dame and joined IBM at Endicott, New York. In 1977, he transferred to Rochester where he has worked on microcode development for the System/38 and AS/400. Mr. Corrigan received an Outstanding Technical Achievement Award for his work on the AS/400.

Reprint Order No. G321-5367.