Architecture, design, and performance of Application System/400 (AS/400) multiprocessors

by J. E. Bahr
S. B. Levenstein
L. A. McMahon
T. J. Mullins
A. H. Wottreng

The architecture, design, and performance of multiprocessors in the Application System/400® (AS/400®) family are discussed. The paper describes how this multitasking system, originally designed as a uniprocessor system, was modified to form a multiprocessor system. The unique approach, using *relatively atomic instructions*, required a minimum of change while providing significant performance gains.

Introduction

The Application System/400® (AS/400®) system is a general-purpose, mid-range family of computers which was first introduced in 1988. It provides batch and interactive capability for commercial and office applications. Among the software and hardware features of the AS/400 system is the layered machine architecture [1] shown in **Figure 1**. The user and parts of the operating system are provided a high-level machine interface (MI). Below this, vertical licensed internal code (VLIC) implements the remainder of the operating system functions. VLIC uses internal microprogrammed interface (IMPI) instructions. Horizontal licensed internal code (HLIC) performs the operations specified by the IMPI instructions. The HLIC is likely to change significantly over time as newer processors are designed, even if the IMPI instructions do not. This constant change of the HLIC is consistent with many other microcoded systems, such as System/370™; however, the IMPI interface can also change without necessarily requiring significant changes in application code or operating system code (above the MI). The layered architecture of the AS/400 system allows the underlying hardware and software interface and functions to change in order to take advantage of technology advances without affecting the end user.

Many of the high-level functions performed by software in other systems are provided below the IMPI in the AS/400 system [2]. Complex functions, such as dispatching

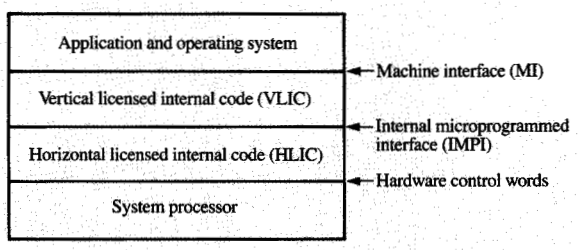


Figure 1

Layered machine architecture of AS/400 system—elements and interfaces.

tasks, queuing, and input/output (I/O) operations, are provided at the IMPI by means of hardware and HLIC. These are some of the functions that an operating system must often alter significantly when multiprocessors are introduced into an architecture. Often the uniprocessor assumption that only one task at a time runs and alters data structures that are shared by other tasks is no longer true for multiprocessors.

The unique architecture of the AS/400 system made it an ideal candidate for design as a shared-memory multiprocessor. The layered architecture made it easier to introduce multiprocessing without requiring changes to existing applications, because multiprocessor changes could be made below the MI. In addition, the AS/400 system was already a multitasking system. Task-to-task communications are controlled by built-in instructions supported by HLIC. Other functions are also written in HLIC—for example, the task dispatcher. This type of high-level support at the IMPI level makes it possible to incorporate multiprocessor architecture while restricting most of the changes to the hardware and HLIC levels.

One of the major changes made at the IMPI level in order to implement multiprocessing was the redefinition of many instructions as *relatively atomic instructions*. Only one instruction that operated on shared IMPI data structures was allowed to run at a time. The instruction execution on one processor was suspended by the processor hardware if another instruction of the same type was already running on another processor. A unique hardware lock was used by the HLIC for each type of relatively atomic instruction. Other hardware changes were also made to support multiprocessors: A common shared bus allowed multiple processors to access main storage. A cache was attached to each processor to reduce the main-storage-access bandwidth needs of the processors. (Earlier versions of AS/400 did not have caches.) Another addition was a mechanism to send messages between processors.

Hardware and software modeling to evaluate performance was done while the hardware was being designed. As a result of this modeling, improvements were made early in the design of the hardware and in the HLIC task dispatching.

Synchronization and serialization

• *Conventional synchronization mechanisms*

In the System/370 architecture, several instructions, such as TEST AND SET, COMPARE AND SWAP, and COMPARE DOUBLE AND SWAP, are defined to provide atomic (i.e., indivisible from start to finish) operations for multiprocessor operation [3]. Other architectures provide similar instructions. For example, the Digital Equipment Corporation VAX™ architecture provides atomic test-and-set-main-storage functions via the BBSSI (Branch on Bit Set and Set Interlocked) and BBCCI (Branch on Bit Clear and Clear Interlocked) instructions [4]. In the COMPARE AND SWAP instruction of the System/370 architecture, fetching an operand (for the purpose of the compare) and storing (for the purpose of the swap) into the location of this operand appear to be an interlocked operation as observed by other processors. (In this paper, we use "interlocked" and "atomic" interchangeably, to refer to single, noninterruptible operations.) It is worth noting that the interlocked fetch–store operation only "appears" to other processors to be a single operation (i.e., other processors, no matter what sequence of fetch and store operations the other processors may perform, cannot get results inconsistent with those that would have resulted from a single interlocked fetch–store operation). Many hardware implementations of this architecture are possible. Some of these are

- Preventing any accesses to the interlocked location during the time between the interlocked fetch and store operations, and delaying accesses to this location by any other processor.
- Allowing only fetches that are not part of an interlocked fetch–store operation (and no stores) to occur between the interlocked fetch and store, since the outcome is the same as if the noninterlocked fetch had occurred before the interlocked fetch.
- Allowing any accesses to occur between the interlocked fetch and store, but repeating the interlocked fetch if a store to the same location occurs between the interlocked fetch and store, since the outcome is the same as if the store had not occurred between the interlocked fetch and store.

The atomicity mechanism for the interlocked fetch–store operation is, in general, different from the mechanism that

can be used in a uniprocessor multitasking environment, where it is sufficient to guarantee that interrupts and task switches do not occur in the middle of this type of instruction. The atomicity mechanism for a multiprocessor requires more than the uniprocessor instruction-level atomicity, since it involves the prevention or repetition of specific types of storage accesses.

Other systems provide hardware locks that can be tested and set in one atomic operation. Each such TEST AND SET lock represents a shared resource and provides the synchronization mechanism so that only one processor operates on the shared resource at a time. The Balance™ multiprocessor system of Sequent Computer Systems [5], for example, provides 64 semaphores that can be TESTed AND SET by any of the 30 processors in the system. The Sequoia Systems fault-tolerant computers provide 1024 TEST AND SET locks in storage [6]. The processors in the system (up to 64) contend for these locks. Other synchronization primitives [7] exist, but these have less in common with the AS/400 multiprocessor architecture than TEST AND SET mechanisms.

Given an elementary, atomic fetch and store sequence operating on one storage location, such as COMPARE AND SWAP or TEST AND SET, software can build more elaborate structures for managing the sharing of storage data and other resources.

• *Relatively atomic instructions*

The AS/400 multiprocessor provides a synchronization solution by means of *relatively atomic instructions*. These instructions are divided into classes. Instructions in each class are atomic relative only to instructions within the same class executing on other processors in the system. (AS/400 instructions, with a few exceptions, are treated as atomic operations on a uniprocessor; i.e., interrupts and task switches are not allowed in the middle of a partially completed instruction. Thus, it is not necessary to consider relative atomicity on a uniprocessor.) *Absolutely atomic instructions* are atomic relative to all instructions executing on all processors. For example, COMPARE AND SWAP in System/370 multiprocessors is absolutely atomic, while COMPARE AND SWAP WORD in the AS/400 system is atomic relative only to instructions in the COMPARE AND SWAP WORD class. While one AS/400 processor is executing an instruction in this class, other processors are permitted by the hardware architecture to access the same storage location simultaneously with relatively atomic instructions not in this class or with regular load or store instructions; however, the operating system avoids this type of simultaneous access by using only relatively atomic instructions in the same class for potentially simultaneous accesses to the same data (except when software ensures via other mechanisms that there is no simultaneous, conflicting access to the data used by relatively atomic instructions). Since these data are internal to the operating system, only the operating system accesses them, and it follows this convention.

In System/370, for the most part, absolutely atomic instructions are used by the software to set software locks that provide protection for storage accesses to shared data. After setting such a lock and completing one or more general-purpose instructions that perform the accesses, software must then use another instruction to release the software lock.

In AS/400, many accesses to the operating system VLIC shared-data structures are combined into a single complex instruction that is not interruptible, or is interruptible only at certain defined points. For example, a single instruction searches a message queue, enqueues a new message in the appropriate position, and moves a waiting task from the wait list of the queue to the task-dispatching queue. Another characteristic of AS/400, mentioned previously, is that, by convention, general-purpose instructions are not used to access the shared-data structures. These characteristics were used to advantage to support multitasking in AS/400 prior to the introduction of multiprocessors. Because of these characteristics, it is possible to define some instructions that access the VLIC shared-data structures as relatively atomic rather than have the VLIC obtain software locks before using these instructions. In some cases, VLIC accesses shared data without using relatively atomic instructions. For example, when task dispatching is disabled on all but one processor, VLIC on the remaining processor can use instructions that are not relatively atomic.

Relatively atomic instructions are divided into classes based on the type of shared-data structure, called IMPI objects, that the instructions access. Instructions that are not in the same class do not access the same shared-data structures. Hardware and HLIC incorporate a lock mechanism into these complex instructions that locks out only instructions from the same class. (This lock mechanism, called instruction-class locks, is described in more detail in the section on hardware locks.) Once a processor has set such a lock, all other processors in the system are prevented from executing instructions from the same class until the first processor's instruction is complete. Instructions in this class in other processors are simply delayed by the hardware until the relatively atomic instruction completes. Instructions in different classes are allowed to execute simultaneously, since they do not have conflicting accesses to the same data. Multiple classes of relatively atomic instructions are defined in order to reduce the likelihood that more than one instruction from a class will attempt to execute simultaneously and thus degrade performance.

The instruction classes follow:

- *Compare and swap* instructions, such as COMPARE AND SWAP WORD.
- *Hold record* instructions, which provide symbolic software locks.
- *I/O*, which includes both IMPI I/O instructions and operations performed by the HLIC when handling I/O interrupts.
- *System timer* instructions, which provide time-of-day, time interval, and clock comparator functions.
- *SRC* instructions, which provide semaphores.
- *SRQ* instructions, which pass messages between tasks.
- *TDQ* accesses, which include updates to the task-dispatching queue and accesses by the HLIC task dispatcher.
- *Primary Directory* accesses, which include accesses to address-translation tables by the translation hardware and by IMPI instructions.

There are several other significant characteristics of relatively atomic instructions. First, I/O interrupts that occur in the middle of a relatively atomic instruction are held pending until the instruction execution is completed. Second, HLIC implements relatively atomic instructions in a manner that prevents other interrupts from causing a partially completed interlocked operation. For example, all main-storage pages that must be used to complete a relatively atomic instruction are referenced before any of the store operations of the instruction are performed. Another processor cannot invalidate a main-storage page that has already been referenced until the completion of the relatively atomic instruction.

- *Serialization*

Some architectures use "strong ordering" for storage accesses. (That is, store operations and fetch operations are performed in the order given by the program. This is not the case with "weak ordering.") In other architectures, the program order of storage accesses is not necessarily the order in which the hardware performs the operations. For performance reasons, the hardware may be designed to carry out the storage accesses in an order that differs from the program order. At times, when the software must ensure that previous storage operations have been completed, a serialization operation is performed. This means that all previous storage operations that are initiated by the processor doing the serialization are completed or have the appearance, to other processors, of being completed. In System/370, the COMPARE AND SWAP instruction and the interlocked fetch–store operation provide serialization mechanisms [3]. While weak ordering allows the hardware to reorder storage accesses for maximum performance, it also places additional requirements on the software to perform serialization operations at times when they would not be needed if the program order of storage accesses had been implemented by the hardware. To avoid this impact on software, AS/400 architecture uses a modified strong ordering for its storage accesses. That is, store operations are done in program order, and fetch operations are done in program order; however, store operations can be done late relative to fetch operations. This type of ordering is also used in System/370 [3]. This architecture allows the hardware to buffer stores in the processor, continue with subsequent fetches, and propagate stores to main storage

PROGRAM ORDER

Processor 1	Processor 2
(1) store A	(1) store B
(2) fetch B	(2) fetch A

REAL TIME ORDER, CASE 1

Processor 1	Processor 2	Time
store A		t_1
fetch B		t_2
	store B	t_3
	fetch A	t_4

Processor 1 fetches old value of B.
Processor 2 fetches new value of A.

REAL TIME ORDER, CASE 2

Processor 1	Processor 2	Time
fetch B		t_1
	fetch A	t_2
store A		t_3
	store B	t_4

Processor 1 fetches old value of B.
Processor 2 fetches old value of A.

REAL TIME ORDER, CASE 3

Processor 1	Processor 2	Time
	store B	t_1
store A		t_2
	fetch A	t_3
fetch B		t_4

Processor 1 fetches new value of B.
Processor 2 fetches new value of A.

REAL TIME ORDER, CASE 4

Processor 1	Processor 2	Time
	fetch A	t_1
	store B	t_2
fetch B		t_3
store A		t_4

Processor 1 fetches new value of B.
Processor 2 fetches old value of A.

Figure 2

Storage accesses with weak or modified strong ordering.

PROGRAM ORDER

Processor 1	Processor 2
(1) store A	(1) store B
(2) serialize	(2) serialize
(3) fetch B	(3) fetch A

REAL TIME ORDER, CASE 1

Processor 1	Processor 2	Time
store A		t_1
serialize		t_2
fetch B		t_3
	store B	t_4
	serialize	t_5
	fetch A	t_6

Processor 1 fetches old value of B.
Processor 2 fetches new value of A.

REAL TIME ORDER, CASE 2

Processor 1	Processor 2	Time
store A		t_1
serialize		t_2
	store B	t_3
fetch B		t_4
	serialize	t_5
	fetch A	t_6

Processor 1 fetches new value of B.
Processor 2 fetches new value of A.

REAL TIME ORDER, CASE 3

Processor 1	Processor 2	Time
	store B	t_1
	serialize	t_2
	fetch A	t_3
store A		t_4
serialize		t_5
fetch B		t_6

Processor 1 fetches new value of B.
Processor 2 fetches old value of A.

Figure 3

Storage accesses with serialization.

(i.e., complete the storage operations) or to the caches of other processors whenever the main-storage bus is available. Software can guarantee that fetches return the latest data by using the serialization mechanism described below.

Consider the example in **Figure 2**. With weak ordering or with modified strong ordering, the store operations may be completed after the fetches. Four of the possible sequences are shown. One can observe that if processor 1 fetches the old value of B, processor 2 may or may not fetch the new value of A. With strong ordering rules, if processor 1 fetches B and obtains the old value of B (i.e., the value that exists before processor 2 stores to B),

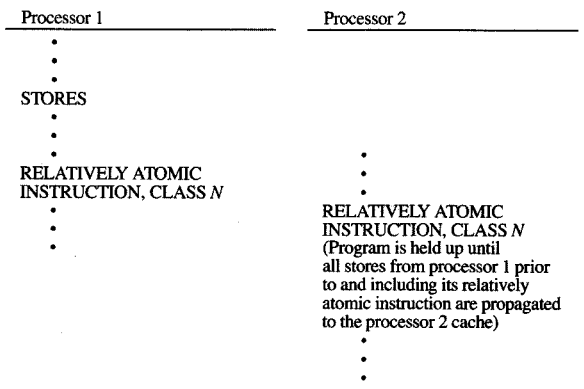


Figure 4

Storage accesses with relatively atomic instructions.

it follows logically that processor 2 will fetch the new value of A (i.e., the value that processor 1 stores into A). In **Figure 3**, a serialization operation is added between the stores and fetches, and some of the possible sequences are shown. With serialization, software can ensure that if processor 1 fetches the old value of B, processor 2 will fetch the new value of A.

Relatively atomic instructions provide serialization as well as interlocking on shared data structures. Serialization in an AS/400 system means that all storage accesses that occur prior to or are part of a relatively atomic instruction have been performed such that processors subsequently executing the same class of relatively atomic instruction will fetch the latest stored data and will not alter data already logically fetched before the relatively atomic instruction was executed. Serialization is achieved by propagating store operations to the common main-storage bus. Bus snooping, a commonly used cache-consistency mechanism for computers [7], is used in the AS/400 system. Bus snooping ensures that once a store operation is propagated to the storage bus, all processor caches are updated or invalidated. Bus snooping is described in the following section. Thus, if two processors execute a sequence as shown in **Figure 4**, processor 1 store operations may be buffered so that they do not update storage or the processor 2 cache; however, should processor 1 and processor 2 perform relatively atomic instructions from the same class in the order shown, the processor 1 stores must be propagated to the processor 2 cache before processor 2 can continue.

Multiprocessor hardware support

Hardware design goals for the AS/400 multiprocessor system included the following:

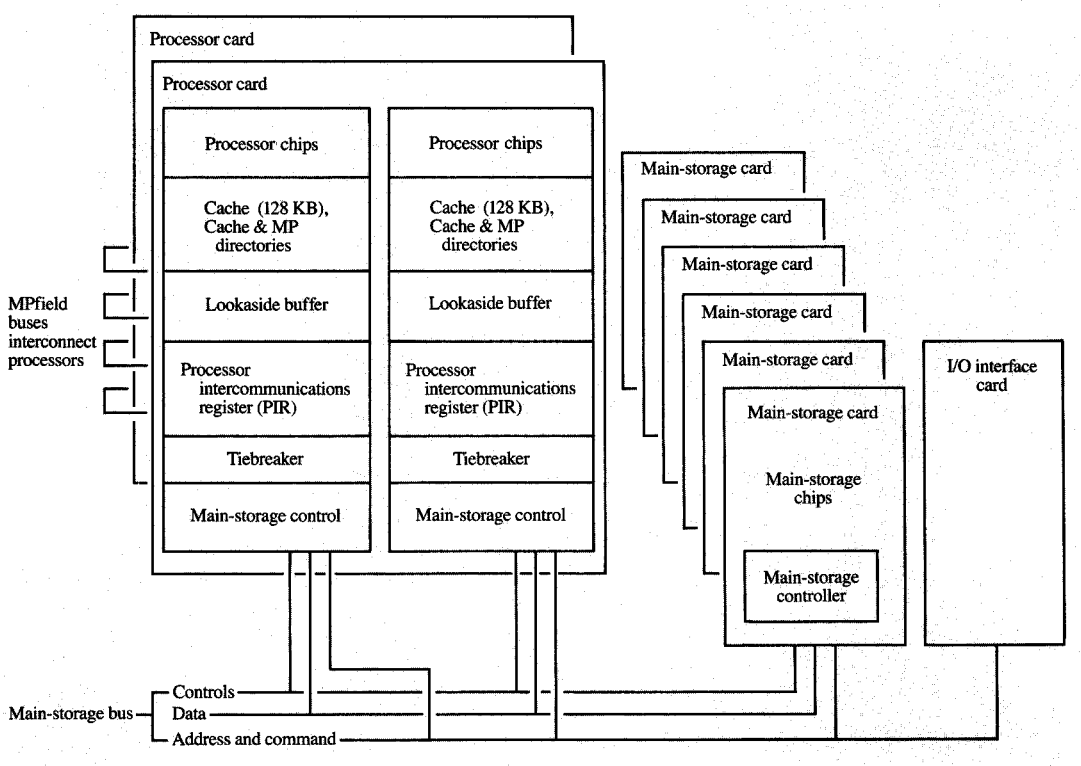


Figure 5

AS/400 system. (The system board supports from one to four processors plugged into two card slots.)

- Providing a shared-main-storage multiprocessor system.
- Providing a bus structure to connect two processors, extendable in the future to four processors.
- Incorporating a high-performance protocol for main-storage bus arbitration.
- Providing HLIC with simple interlock and serialization mechanisms called hardware locks.
- Handling all cache-consistency problems (keeping all cache copies of a main storage location consistent).
- Providing mechanisms to handle lookaside-buffer and primary-directory consistency.
- Minimizing cost and design overhead in converting from uniprocessor to multiprocessors.

Early in the design of the AS/400 multiprocessor, it was decided that a single design should be used for both uniprocessor and *N*-way multiprocessor. The goal was to create a single processor design that would require minor enhancements to produce a two-way processor. To help debug the system and further test the design, a four-way multiprocessor was constructed—even though a two-way processor would be the actual product. Building a four-way system in the laboratory helped expose bugs in the system more quickly by increasing the probability that software and hardware design flaws would be found. For example, consider a noninterlocking instruction such as ADD LOGICAL HALFWORD IMMEDIATE, which fetches the contents of a storage location, adds an immediate value to the contents, and stores the sum in the same storage location. If the operating system uses this instruction to increment a shared-storage location, a problem occurs if two processors simultaneously increment the storage location, since one of the increments will be overwritten by the other processor. The probability of two processors simultaneously performing this instruction is greater with a four-way processor than with a two-way processor (assuming that the frequency of executing the instruction is the same).

Figure 5 shows the processor and main-storage components of the AS/400 system.

Another early design choice was bus snooping for cache consistency. Because all processors are on the same main-

storage bus and all processors perform system-bus arbitration in parallel, it was fairly straightforward to implement main-storage bus snooping [7]. The caches are implemented as store-through; i.e., all store operations are propagated to main storage on the common main-storage bus. Store operations by one processor result in other processors invalidating their cache lines corresponding to the main storage location being accessed. The design contains two identical copies of the cache-directory array: the "cache directory" and the "multiprocessor directory" (both shown in Figure 5). Internal fetches and stores can read the cache directory, while store operations by outside processors are checked against the multiprocessor directory, to determine whether a cache line-invalidation is required. Whenever an invalidation is required, the cache is "stolen" for one cycle to invalidate both directory arrays. Cache consistency is thus maintained by snooping on stores and invalidating the corresponding cache lines, which no longer contain the most current copy of the storage data. This prevents the processor from fetching old data from the cache, since a fetch from one of these invalidated cache lines results in a cache miss, and the cache line is fetched from storage. Invalidations have a small effect on system performance.

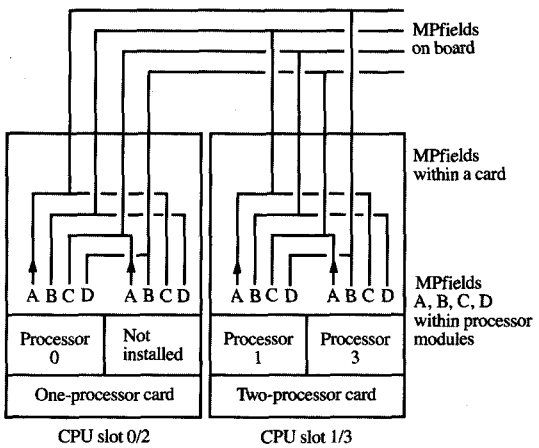


Figure 6

MPfield board wiring for a three-way processor system consisting of one AS/400 card with one processor and one AS/400 card with two processors.

- *The AS/400 interconnection scheme*

The AS/400 multiprocessor interconnection system does not have a central hub. Control information that would normally be passed to a central hub is transmitted between processors on buses called multiprocessor fields, or *MPfields*. This control information shows each processor's store operations, fetch requests, hardware lock operations, and some other operations. The store and fetch operations that require the use of the storage bus are buffered and held pending in each processor until arbitration for the storage bus is complete and the bus is available for the operation. Each processor transmits information pertaining to its operations on its own MPfield output bus, which is sent to all the other processors, and each processor receives information pertaining to its peers' operations on one input bus for each peer. Thus, in a four-way processor system, each processor has one output and three inputs. **Figure 6** shows the MPfield wiring among processors. Each processor MPfield output is designated "A" by that processor (indicated by arrows in the figure). The output of processor 0 is input D of processor 1 and input B of processor 3. Were processor 2 installed, the signals would be input C. In this way, all processors have identical logic, any processor may be placed in any slot, a processor is identified with a different "name" by each of the other processors, yet the processors have consistent views of the overall system state. The control information sent on the MPfield buses is used by the processors to track all pending operations and to arbitrate the storage bus and hardware locks. Bus hardware for this tracking and arbitration must be duplicated in all processors. The state of this bus hardware in one processor is kept consistent with the bus hardware state in the other processors, in the sense that all of them reflect the same set of pending store, fetch, and hardware lock operations. Also, each processor must arrive at the same conclusion relative to which processor can use the storage bus or hardware locks, since arbitration for these is resolved in parallel in all of the processors.

This interconnection scheme yields an interesting result. A processor that includes the extra logic required for multiprocessors can run in a multiprocessor configuration or by itself. If one of the processors in a multiprocessor system fails, the system can be reinitialized with the failing processor disabled. If diagnostic software that executes during the initial program load (IPL) detects a failing processor, it uses multiprocessor configuration registers to disable the processor and its corresponding MPfield bus.

The bus arbitration algorithm requires that bus hardware be duplicated in all processors; this speeds up arbitration by tracking all pending store operations that are buffered in each processor until the storage bus is available. In addition, the main-storage-card interface had to be designed to allow 100% usage of the main-storage bus. A tie-breaker mechanism is implemented; processor priority changes in a round-robin fashion.

• *Main-storage interface*

The storage bus consists of three parts: main-storage controls, the main-storage address and command bus, and an 8-byte-wide main-storage data bus. Up to six main-storage cards can be installed. If four or more cards are installed, addresses are interleaved among the first four cards. Main-storage cards independently process different commands at the same time.

Since the main-storage card takes three cycles to retrieve fetched data from its internal dynamic random access memory (DRAM) arrays, the data transfer for a fetch is done no less than three cycles after the fetch address and command are sent to the main-storage card. Fetches usually involve fetching all 32 bytes for a cache line, so the data transfer for a fetch requires four cycles (bus and storage cycle times are the same) on the 8-byte main-storage data bus.

All processors arbitrate for control of the shared bus, monitoring the main-storage interface and MPfield buses each cycle to determine, in parallel, which processor should use the main-storage bus in the next cycle. This allows the main-storage bus to be used every cycle. (Control of the bus can switch from processor to processor every cycle.) This maximum utilization would be difficult to achieve if bus arbitration were done at one central point, since it would take a cycle just to communicate the arbitration result to each processor.

Because the cache greatly reduces the number of main-storage fetches, most storage-bus commands are for store operations. However, a store-through cache creates a greater load on the storage bus than a store-in cache, especially with many single-byte store operations. As a result, effort was expended to reduce the time required for store operations. Many ideas were combined to allow one store command to be issued each cycle on the main-storage bus, when multiple main-storage cards are available to service these stores. (Bursts of fetch commands can be issued at this rate, but the rate of fetches is limited by the four-cycle data transfer for a cache-line fetch.) AS/400 multiprocessor architecture requires interlocked stores to shared storage but allows noninterlocked stores to nonshared *bytes* within an 8-byte word. Thus, main storage must be byte-writable (i.e., processors can issue single-byte store commands on the main-storage bus). However, the main-storage card implementation, in order to handle the error-correction code (ECC), turns single-byte store operations into read-modify-write sequences on the main-storage card in the following way: The main-storage card fetches the word plus check bits from its internal DRAM chips, corrects errors, modifies the selected byte, generates new check bits for the new set of 8 bytes, and stores the new word and ECC into the DRAMs. A main-storage card can perform such a read-modify-write operation in five cycles; a direct 32-byte store operation requires four cycles.

The bus design involved a number of trade-offs. A single, shared bus has limitations: It has a limited bandwidth; all processors must arbitrate for control of it; it presents a heavier electrical load than point-to-point buses, thus requiring a longer time to drive the bus signals. Increasing this time to accommodate more processors reduces the bus bandwidth and, in turn, limits the number of processors that can be supported. On the other hand, its simplicity is a compelling argument in its favor. Furthermore, enhancements in the main-storage card and processors help to lessen the effects of a single main-storage bus. As a result, having a single shared bus does not significantly degrade total system performance.

• *Hardware locks*

A highly sophisticated locking structure is incorporated into the AS/400 system hardware. In multiprocessor systems, an extension is provided to every HLIC word (in control store). This extension is for HLIC to issue commands to request or release hardware locks. These lock commands can be coded in parallel with any other HLIC function that can be coded in the normal HLIC word, and are only to serialize storage accesses and to interlock HLIC code sequences, executing in different processors, that implement IMPI relatively atomic instructions. The hardware implements ten different locks. Eight such locks are used by HLIC to support the eight classes of relatively atomic instructions, one lock is used to interlock HLIC accesses to shared resources internal to HLIC, and one lock is used for hardware debugging.

An HLIC sequence for using a hardware lock might consist of the following HLIC words:

- Requesting a hardware lock (subsequent HLIC operations in this processor are suspended by the hardware if the hardware lock has already been obtained by another processor).
- Fetching shared data.
- Operating on the data (optional).
- Storing data into the shared location.
- Releasing the lock.

Some HLIC sequences perform multiple fetches, stores, and other operations between requesting and releasing the lock.

Hardware locks are preferable to other interlock mechanisms because hardware locks do not involve main-storage accesses. One alternative would be a "test-and-set" that would use main storage and operate more slowly. The hardware-lock scheme is simpler for HLIC, since the test-and-set scheme would require HLIC to handle situations in which the test-and-set variable is already set,

and HLIC would have to loop on the test-and-set command until the test found that the variable had been reset. Also, test-and-set variables in shared storage increase the cache miss ratio in many cache designs. Another advantage of hardware locks is that multiple lock requests for different locks can be granted in parallel in one cycle, whereas test-and-set storage accesses with a shared main-storage bus require multiple bus cycles.

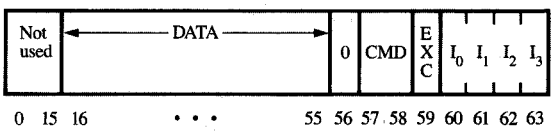
Because of the small number of locks and processors, the HLIC lock commands to obtain or release hardware locks can be broadcast between processors in one cycle across unique wires in the MPfield. If a lock is requested and has not been obtained by any other processor, execution of the HLIC is not delayed. Multiple processors can collectively obtain more than one hardware lock in a single cycle unless conflicts occur. Releasing a hardware lock is also normally a single-cycle operation.

In contrast, the Balance system of Sequent Computer Systems [5] supports 30 processors and 64 semaphores, whose values are communicated between processors across a serial interface.

The ten hardware locks are independent, and a processor can own more than one lock at a time. A deadlock could occur if one processor obtained one lock and requested another lock that was already owned by a second processor, while the second processor requested the lock already owned by the first processor. To prevent this type of deadlock situation, the locks are numbered, and HLIC for a processor always acquires multiple hardware locks in order, from lowest to highest.

The serialization provided by relatively atomic instructions (as described earlier in the section on serialization) is accomplished at the hardware level when a lock is released. If previously issued store operations are buffered in the processor, they are propagated to storage whenever the storage bus is available. No other processor can obtain the hardware lock that has been released until these buffered store operations are propagated to the storage bus and snooping has invalidated all copies of these storage locations in the other processor caches, since an HLIC sequence in another processor that requests such a hardware lock is suspended by the hardware until this occurs. There is no delay due to the buffered store operations in the HLIC sequence that released the hardware lock.

Simultaneous lock requests in the same cycle are rare but must be handled. More often, a lock collision results when one processor requests a lock when another processor is in the middle of a locked sequence. Simultaneous lock requests are resolved by the hardware through the use of priority bits in each processor that define which processor has the highest priority and therefore obtains the hardware lock. The priority bits are kept in a register in each processor, along with the hardware to detect and handle collisions. The priority changes pseudorandomly over time to prevent any one processor from having a constant disadvantage. Since all processors implement the same pseudorandom algorithm, the priority bits in all processors are consistent.



DATA: 40 bits, enough for a virtual address segment identifier

CMD: 2-bit command

EXC: 0 = Reset acknowledgment bits only
1 = Write command, data, and ID bits to all processor PIRs, and cause a PIR interrupt on processors with ID bit = 1

I_0–I_3 = ID (processor identification) bits for processors 0–3, respectively

Figure 7

Processor intercommunications register (PIR) format.

- *Processor intercommunications register*

All multiprocessor systems require a message-handling scheme to send messages between processors. The AS/400 multiprocessor design uses a hardware register, called the processor intercommunications register (PIR), and an efficient set of controls to send messages between processors. Using a register instead of main storage permits processor intercommunication without causing cache misses. The PIR, duplicated in all processors, acts as a main-storage location and can be protected with one of the hardware locks. When a write to the PIR occurs, the PIRs of all processors are set identically. A message-handling scheme must be capable of causing an interrupt in the receiving processor in order to have the message handled quickly. In the AS/400 design, that interrupt is handled by HLIC.

Figure 7 shows the PIR format. The ID (processor identification) mask is included in the PIR, each bit corresponding to one of the possible processors. A unique main-store-interface command allows messages to be sent to the PIR register. An additional PIR bit is used to generate interrupts: When a message with the exception bit on (EXC in Figure 7) is sent to the PIR, an interrupt is caused in each processor whose ID bit is 1. To acknowledge a message, a processor need only reset its ID bit by sending a message to the PIR with the exception bit off. No hardware lock is required to reset an ID bit, because the hardware can reset the bit atomically.

All copies of that PIR ID bit in all processors are updated simultaneously to keep the ID fields in all processors consistent. HLIC conventions allow using the PIR ID bits as busy indicators. The HLIC can release the PIR lock and process other IMPI instructions after sending a message, rather than spinning in a loop until the ID bits are reset. Hardware resets the PIR ID bits as other processors respond to the PIR message. The HLIC does not send out PIR messages when the PIR ID bits indicate messages still outstanding.

The data field in the PIR (bits 16–55) is large enough to send an AS/400 virtual-address segment identifier from one processor to another. This field width allows the PIR to be used to send to other processors the virtual address of a segment to be purged from the lookaside buffers of all processors. The data field is also used as a command extender, since the CMD field contains only 2 bits.

- *Hardware primary-directory lock*

The AS/400 system uses virtual addresses, which must be translated to real addresses before main storage is accessed. High-speed lookaside buffers in the processor, as shown in Figure 5, contain the addresses of the most recently translated page addresses. Special hardware maintains the lookaside buffers and the primary directory (the main table used in translating virtual addresses, which resides in main storage and is shared by all processors and all tasks). Lookaside buffer misses are handled by the hardware; however, one of the hardware locks interlocks hardware primary directory searches in order to allow maintenance of the primary directory.

HLIC follows special rules in turning off the valid bit associated with a virtual address in the primary directory. When the valid bit is 0, any attempt to translate the address causes a page fault. Also associated with the primary directory entry for each virtual page are a reference bit and a change bit. The reference bit is set by the hardware if the bit is 0 when a virtual address is translated using the primary directory. Subsequent virtual address translation may use the high-speed lookaside buffers. The change bit is set by the hardware if the change bit is 0 when a store operation into the page is performed. The reference bit acts as an indication to HLIC that a processor lookaside buffer may hold a copy of a primary directory entry. The reference bit is used by HLIC in the following manner when implementing an IMPI instruction that requires resetting the valid bit: If the primary-directory lock is obtained and the reference bit is 0, the valid bit may be set to 0. If the reference bit is 1, a PIR message is required to purge the page from all of the lookaside buffers in order to keep the lookaside buffers from being used subsequently to translate virtual addresses that are no longer valid.

- *Set bit function*

The main-storage card provides a set-bit and a reset-bit function, which is a special command that is executed by the main-storage card as a read–modify–write of its internal DRAMs. Therefore, the processor hardware does not have to do a read–modify–write to set the reference and change bits during a virtual-address translation using the primary directory. This set-bit function avoids having to interlock primary directory translations and allows multiple concurrent translations.

Task dispatching

AS/400 multiprocessor task dispatching provides automatic workload balancing among processors and, by using the same objects and methods as were used by previous uniprocessors, avoids the need for significant VLIC changes. As with a uniprocessor, all ready-to-run tasks are enqueued in priority order on a single, system-wide task-dispatching queue in shared storage. The task dispatcher can be invoked on any processor by an IMPI instruction. It can also be implicitly invoked after the task-dispatching queue is altered by a built-in function or IMPI instruction. (The I/O interrupt handler is an example of a built-in function that can alter the task-dispatching queue. IMPI instructions that can implicitly invoke the task dispatcher include those supporting semaphores and message passing.) Task dispatching remains a built-in processor function performed by HLIC between IMPI instructions. The VLIC is not directly involved in either the decision to switch tasks or the actual saving and loading of task state.

The only functional change caused by multiprocessing that is visible to VLIC is that task selection is not based solely on priority. "Processor eligibility" and "cache affinity" also affect task selection. *Eligibility* can be used by VLIC to restrict a task to a subset of the available processors. *Cache affinity* identifies the processor on which a task has most recently executed and, therefore, the processor on which the task is likely to have the highest cache hit rate and experience the best performance. Other than initializing the new task-state fields for processor eligibility, cache affinity, and current processor, no VLIC changes were required for multiprocessor task dispatching.

When invoked, the task dispatcher checks the task-dispatching queue for changes since the last task-dispatcher call executed on any processor. If the task dispatcher finds changes, it searches the queue to determine what task should be running on each processor. If it is determined that processors other than the one executing the task dispatcher must perform task switches, a list of required task switches is stored in an HLIC object in main storage, and the first processor required to switch is signaled to run the task dispatcher. If the task dispatcher, when running on this processor, finds that the

task-dispatching queue has not changed since the previous invocation, the HLIC performs the task switch using the information stored in the HLIC object rather than repeating the queue search. (A *task switch* consists of storing the state of the current task and loading state information for the new task.) If the HLIC object indicates that additional task switches are required, the next processor on the list is signaled. This continues until all required task switches have been completed or until a task dispatcher finds the task-dispatching queue altered, the latter causing the queue to be searched again and a new list of required task switches to be generated.

Task selection is based on a combination of priority, eligibility, and cache affinity. The task dispatcher selects the highest-priority tasks, unless they are prevented from being dispatched because of eligibility or cache affinity. Eligibility is never overridden by the task dispatcher. If all processors for which a task is eligible are assigned to higher-priority tasks, the task is not dispatched.

If the processor for which a task has cache affinity is assigned to a higher-priority task, the task is skipped (not dispatched), unless doing so would result in a processor remaining idle or an excessive number (as defined below) of tasks with cache affinity being skipped. The *skip threshold* is the limit on the number of tasks that may be skipped because of cache affinity and is specified by the VLIC. If the number of skipped tasks reaches the skip threshold, cache affinity is ignored and the task is assigned to any processor for which it is eligible. If tasks are skipped and the end of the task-dispatching queue is reached before each processor is assigned a task, cache affinity is ignored and skipped tasks are assigned to the remaining processors. When affinity is ignored, either because of the skip threshold or in order to assign a task to an otherwise idle processor, tasks that are closest to the front of the queue are not selected, because they are presumed to have stronger affinity than those farther back in the queue. Tasks that are nearer the front will usually have a shorter wait for their preferred processor and have more data remaining in cache when they run, so system performance will benefit more from skipping them than from skipping tasks farther down the queue.

Initially, a task has equal affinity for all processors. When a task is initially dispatched, processor selection is based only on priority and eligibility. HLIC sets the cache affinity of a task to a specific processor when the task is switched to that processor. Certain IMPI instructions that can result in a task being removed from the task-dispatching queue for a long time can specify that the cache affinity of the task be reset to the initial state.

Performance

Early in the development of the two-way multiprocessor (Model D80), the performance objective was 1.7 times the throughput of the Model D70 uniprocessor. This was a 15% degradation from the ideal performance (twice that of a single processor). This factor was deemed a realistic expectation for an initial multiprocessor scheme.

An interactive workload called RAMP-C (Repeatable And Measurable Performance–COBOL) was used to model and measure the performance of the AS/400 multiprocessor system. The workload simulates the activity of workstations in commercial data processing. Clearly, other environments might have caused a significant departure from the performance values given. The performance analysis effort for the multiprocessor machine included modeling the hardware and software design prior to completion of the hardware implementation and of the system performance measurement.

- *Hardware and software design modeling*

Various analysis techniques assessed aspects of AS/400 multiprocessor performance. The two major classes of performance factors that required evaluation were 1) hardware and HLIC effects and 2) software implications. Performance issues related to hardware and HLIC included

- Cache consistency.
- Cache miss rate, both aggregate and dynamic.
- Main-storage contention, which is related to main-storage card and main-storage bus utilization.
- Synchronization of instruction-class locks.
- Lookaside buffer synchronization.
- Queue-structure manipulation.
- Task-dispatching algorithms.

Several models provided for assessment of the performance questions. The first was a simulation model that provided an understanding of main-storage contention and the effect of cache-consistency maintenance. This model was a low-level abstraction of the processor design. It encompassed all aspects of the hardware design that are shared facilities (e.g., cache data bus and memory-card controller) and enough detail of other parts of the architecture to cause all significant queuing effects to occur in the simulation. The input to the model consisted of a sequence of IMPI instructions generated on the basis of frequencies observed for uniprocessors. Code that emulated the HLIC for the IMPI instructions was used to simulate the detailed interaction of machine facilities.

The second category of models consisted of analytic approaches that evaluated the performance degradation due to the instruction-class-lock implementations. The frequency of occurrence of instructions in uniprocessors was used in the calculations. Various instruction-class-lock schemes, which hold class locks for different durations, were evaluated. Degradation of throughput was determined,

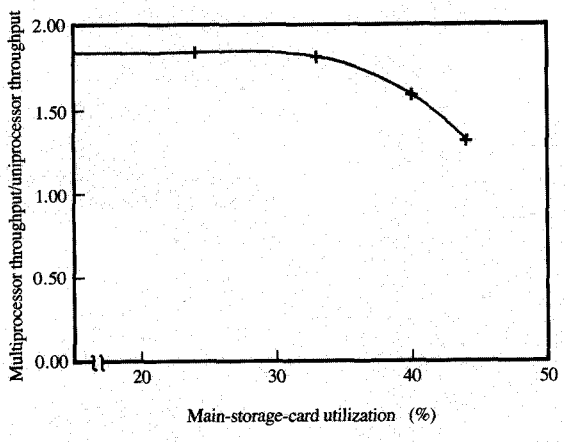


Figure 8

Processor sensitivity to main-storage-card utilization.

and trade-offs were made on instruction-class-lock design schemes.

The third category of model was address-trace analysis, which is important in helping one to understand cache performance. Many real-address main-storage traces taken from the earlier uniprocessor (AS/400 Model B70) machines were used as input to models of a cache directory. Reference patterns and overall behavior in terms of cache miss rate were outputs of these models.

A fourth category of model was main-storage-card utilization, one factor that is critical to multiprocessor throughput. This parameter represents the percentage of all machine cycles that a card controller is busy. The simulation model used represented the processors simply as originators of memory requests of various types. Memory cards were defined to the model with parameters describing the time for processing requests. For a given configuration and set of parameters, simulation projected the processor throughput.

A study of model results showed significant sensitivity to the number of main-storage cards installed and the speed of the DRAM chips used on the main-storage cards. Main-storage-card utilization was greater with slower DRAMs and with fewer main-storage cards. As card utilization increased over 30%, throughput of the multiprocessor dropped off, as shown in **Figure 8** for a two-way multiprocessor. (The uniprocessor throughput did not vary significantly in the range of main-storage-card utilization shown in Figure 8.) The AS/400 Model D80 has DRAM chips with an 80-ns data access time and can have a maximum of six main-storage cards. With this configuration, main-storage-card utilization averaged 22% for the two-way multiprocessor.

Results of the modeling work showed that ideal throughput was reduced by 7.5% after consideration of all hardware and HLIC factors. The contributors to this degradation broke down roughly as follows:

- Instruction-class-lock contention (3.0%). This degradation was a queuing effect that depended upon the class-lock synchronization scheme.
- Cache hit rate reduction (2.3%). This resulted from processors interacting and changing data in shared memory. The store-through cache design caused directory entries that would not be affected on a uniprocessor to be purged in a multiprocessor. This resulted in a reduced cache hit rate.
- Main-storage contention (1.6%). This effect was due to contention for the shared main-storage controllers and for the main-storage bus.
- Task dispatcher multiprocessor overhead (0.6%). Additional function provided in the HLIC task dispatcher for multiprocessors requires additional microcode path length compared with that of the uniprocessor dispatcher.

A simulation model of the software was used to provide an understanding of multiprocessor operating system effects. The performance model simulated the flow of user tasks and operating system tasks when the RAMP-C benchmark was executed. The effects of cache affinity (the ability to maintain tasks on the same processor to maximize the cache hit rate) were observed by the model. The effects of different algorithms for allocating tasks to processors was a key element of this area of the analysis. In particular, the skip threshold provides a means of extending the search of ready-to-run tasks. Modeling portrayed the effects of the skip-threshold value on the cache affinity of tasks, as shown in **Figure 9**. Tasks with affinity had a high initial cache hit rate. When the first 100 cache accesses had an 86% cache hit rate, the cache was considered "hot." Increasing the skip threshold provided a greater probability of finding a task with affinity for an available processor. As can be seen in Figure 9, a skip threshold of 1 provided the preponderance of the benefit. Little additional probability of finding a task with affinity was gained for increased searches. Having a skip threshold larger than 1 reduced the likelihood that the highest-priority task would complete quickly.

Modeling results showed a multiprocessor degradation of 2.2% due to all software effects. Thus, the total hardware and software degradation was 9.7%. Based on an ideal multiprocessor ratio of two times a single processor, the modeled degradation implied a multiprocessor performance factor of 1.81 for the two-way multiprocessor machine.

• *Performance measurement*

Two sets of internal performance counters are employed in the hardware to measure multiprocessor performance. The first set of counters accumulates data for each of the 21 unique states of the multiprocessor, including application-run-time state, VLIC-run-time state, wait states, exception states, and the suspended state. The suspended state, unique to multiprocessors, is entered by one or more processors when another processor executes an instruction that suspends their IMPI instruction execution and task dispatching. For each state, the counters accumulate the numbers of a) instructions executed, b) processor cycles, and c) times the state is entered. A second set of internal counters monitors cache hits and misses, main-storage bus use, and multiprocessor-lock conflicts. Measurements undertaken on systems running the RAMP-C benchmark show a degradation of 10.7% from the ideal multiprocessor, or multiprocessor throughput of 1.79 times that of a single uniprocessor.

Conclusion

The AS/400 multiprocessor effort represents a significant step forward for the AS/400 system. The intent of the multiprocessor architecture was to minimize the software changes required to support multiple processors. The uniprocessor operating system was not significantly changed to handle shared data objects, because either the code already handled them for a multitasking environment, or the relatively atomic instructions handle them for a multiprocessor environment. The operating system already used the IMPI instructions, which accessed IMPI-level shared data structures and which could be redefined as relatively atomic to provide interlocked accesses. The relatively-atomic-instruction approach had much less of an effect on the software than if the conventional approach to interlocking instructions had been used. The design includes a four-way processor used in the development laboratory (since multiprocessor problems surfaced more quickly in the four-way system).* The design provides HLIC with the necessary mechanisms to efficiently implement AS/400 multiprocessors and maintains both simplicity and performance with minimal overhead.

Performance modeling of the AS/400 multiprocessor helped us gain a better understanding of the task-dispatching characteristics of the system and hardware effects. This modeling enabled us to modify the HLIC task-dispatching algorithm and ensure that the multiprocessor met its performance objectives. The performance models predicted a two-processor system throughput of 1.81 times that of a uniprocessor. System performance measurements indicated a factor of 1.79, a 3% difference between model results and measurements.

*This four-way multiprocessor was recently announced as an IBM product (AS/400 Model E95).

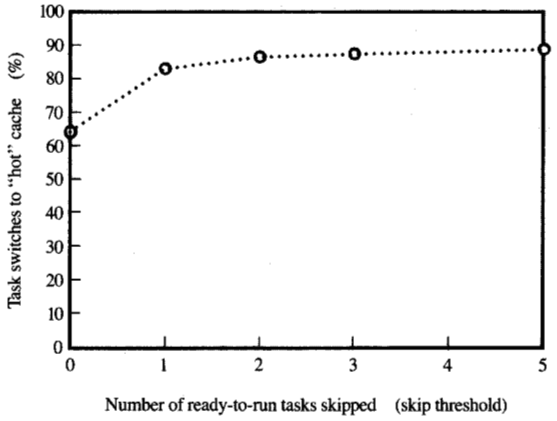


Figure 9

Skip threshold summary.

References

1. M. R. Funk, Q. G. Schmierer, and D. J. Thomforde, "System Processor Architecture," *IBM Application System/400 Technology*, Order No. SA21-9540, pp. 100–103, June 1988; available through IBM branch offices.
2. B. E. Clark and M. J. Corrigan, "Application System/400 Performance Characteristics," *IBM Syst. J.* **28**, No. 3, 407–423 (1989).
3. *IBM System/370 Extended Architecture, Principles of Operation*, Order No. SA22-7085-0; available through IBM branch offices.
4. R. N. Gamache and K. D. Morse, "VMS Symmetric Multiprocessing," *Digital Tech. J.* **1**, No. 7, 57–63 (1988).
5. S. Thakkar, P. Gifford, and G. Fielland, "The Balance Multiprocessor," *IEEE MICRO* **8**, No. 2, 57–69 (1988).
6. P. A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing," *Computer* **21**, No. 2, 37–45 (1988).
7. M. Dubois, C. Scheurick, and F. A. Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors," *Computer* **21**, No. 2, 9–21 (1988).

Received October 1, 1990; accepted for publication January 2, 1992

James E. Bahr *IBM Application Business Systems, Hwy. 52 & 37th St. NW, Rochester, Minnesota 59901 (JBAHR at RCHVMW2).* Mr. Bahr is a Senior Planner, responsible for system price/performance analysis of the AS/400 system. Prior to this assignment, he was a performance analyst in the Rochester Programming Laboratory, focusing on multiprocessor performance. Past assignments include technical and managerial positions in the Rochester Engineering Laboratory for processor development. Mr. Bahr holds one patent and has published two papers on multiprocessor performance. He joined IBM in 1973 after receiving a B.S.E.E. degree from the University of Minnesota.

Sheldon B. Levenstein *IBM Application Business Systems, Hwy. 52 & 37th St. NW, Rochester, Minnesota 59901 (SHEL at RCHVMX).* Mr. Levenstein is a Staff Engineer in AS/400 High End Processor Development. He joined IBM in 1980 as a junior engineer and has been involved in the development of the AS/400 processor chips. He has had several assignments dealing with processors, storage control, and multiprocessor design. Mr. Levenstein has applied for four patents; he was awarded an IBM Outstanding Technical Achievement Award for his contributions to the AS/400 multiprocessor effort. He received a B.S.E.E. from the University of Illinois in 1980.

Lynn A. McMahon *IBM Application Business Systems, Hwy. 52 & 37th St. NW, Rochester, Minnesota 59901 (MCMAHON at RCHVMX).* Mr. McMahon is an Advisory Programmer in the AS/400 HMC development group. He joined IBM in 1977 as an associate programmer and worked on various emulation and device-control microcode projects before becoming involved in horizontal-microcode development for the System/38™ and AS/400 processors. Mr. McMahon has received two patents and applied for four others; he has received an IBM Outstanding Technical Achievement Award for his work on the AS/400 multiprocessor IMPI definition. He received a B.S. degree in computer science in 1975 and an M.S. degree in computer science in 1977, both from Iowa State University.

Timothy J. Mullins *IBM Application Business Systems, Hwy. 52 & 37th St. NW, Rochester, Minnesota 59901 (MULLINS at RCHVM3).* Mr. Mullins joined the Rochester Development Laboratory after receiving a B.S.E.E. degree from the University of California at Berkeley in 1977. He has done work in the design and development of input/output controllers for System/38 user-terminal devices, such as workstations and operator consoles. Mr. Mullins later became involved in CPU development in the areas of logic design and timing analysis. In 1982, he received an M.S.E.E. degree from the University of Minnesota. Mr. Mullins is currently an Advisory Engineer in the Rochester Laboratory Hardware Design Center and is involved in AS/400 system design and performance analysis.

Andrew H. Wottreng *IBM Application Business Systems, Hwy. 52 & 37th St. NW, Rochester, Minnesota 59901 (AHW at RCHVMX).* Mr. Wottreng is a Senior Engineer in the AS/400 High End Processor Architecture and Definition Department. He joined IBM in 1974 as a Junior Engineer, and has been involved in the development of System/34, System/36™, and AS/400 processors. He has had several assignments dealing with chip designs, performance modeling, maintenance procedures, and architecture. Mr. Wottreng has received one patent and applied for another; he received an IBM Outstanding Technical Achievement Award and an IBM Corporate Award for his work on the AS/400 multiprocessor architecture. He received a B.S.E.E. from Marquette University in 1971 and an M.S.E.E. from the University of Minnesota in 1980.