The design and implementation of VM/370 attached processor support is discussed from the point of view of adding radical new function to an existing operating system. Three major design decisions are described, and performance is analyzed as it relates to those decisions.

VM/370 asymmetric multiprocessing

by L. H. Holley, R. P. Parmelee, C. A. Salisbury, and D. N. Saul

This paper discusses the design and implementation of the attached processor (AP) support first available in Release 4 of VM/370. The term *attached processor* in this context refers to the specific implementation on System/370 Models 158 and 168 and on the 3031 processor of an asymmetric processor configuration. This implementation comprises one central processing unit (CPU) with full execution, channel, and input/output (I/O) capability, and one attached processing unit (APU) which shares main storage with the CPU but has only execution capability.

Described are considerations for adding shared storage multiprocessing to the then existing VM/370 operating system and the tradeoffs that were required to achieve a practical result. We hope to show how significant new function was provided in VM/370 within the context of an original operating system design which did not provide for such function. The major design points are covered, with emphasis on how they affected performance. It is not our intent to present a complete view of VM/370 Release 4, but rather to highlight the overall design, the tools, and the techniques of design and implementation as they relate to attached processing. Toward that end, three central design problems are presented in detail, and others of lesser significance are mentioned briefly.

It is assumed that the reader is familiar with the concepts and facilities of IBM's Virtual Machine Facility/370 (VM/370). If not, the paper by Seawright and MacKinnon elsewhere in this issue[1] provides basic information and an extensive bibliography.

The genesis of this project lay in the rapid growth of large VM/370 systems and the requirement for more processing capability to satisfy that growth. The multiprocessing options of System/370 suggested an obvious and attractive solution to this problem. The additional processing power could be provided, and the single system image of a uniprocessor could be maintained. But many questions concerning the nature and feasibility of software support remained to be answered.

Prior to VM/370 Release 4, the only IBM system control programs that supported shared storage multiprocessing on System/370 were Multiple Virtual Storage (MVS) and Time Sharing System/370 (TSS/370). Both systems supported symmetric (multiprocessor) as well as asymmetric (attached processor) hardware configurations. The control of multiple processors was an inherent part of the design objectives for both MVS[2,3] and TSS.[4] The software architecture of each had basic multiprocessor functions such as locking, signaling, and interprocessor communication. Further, the code in these systems was organized so that parallel execution was possible. These essential primitive operations were not defined in VM/370.

In other words, VM/370 did not have the fundamental building blocks required to support parallel processing. Lacking were functions for locking and unlocking (enqueuing and dequeuing) nonsharable resources, provision for parallel execution of shared code, protocols for communication between processors, and a mechanism for serializing I/O processing. Moreover, much of the control program supervisor provided no design base for parallel execution. A total rewrite of VM/370 was not practical, so an effort was undertaken to selectively rewrite parts of the control program (CP) to fulfill the requirements for integrity and performance while minimizing user disruption. Finally, it was decided to support an attached processor configuration rather than a symmetric multiprocessor configuration.

Review of multiprocessing

The basic characteristic that distinguishes a tightly coupled multiprocessing system from a multicomputer system is shared main storage, with simultaneous operation of the multiple processors under control of a single operating system.[5] In loosely coupled configurations such as ASP, shared spool, and JES3, each processor has its own supervisor, so these configurations are not considered true multiprocessing systems. Also outside the above defini-

tion are multiprogramming systems that support concurrent, rather than simultaneous, execution. A multiprogramming system may appear to have multiple functions operating at the same time, but at any given instant, only a single instruction stream is in execution on the processor. A multiprocessor has two (or more) streams in execution at the same time. All the above systems are sometimes considered parallel processors but this designation is misleading since the degree and level of parallelism can vary.

multiprocessing defined

Enslow[6] defines a true multiprocessor as having four characteristics:

- Two or more processors, each of approximately equal power.
- Shared access to memory.
- Shared access to input and output.
- A single operating system in control.

By this strict definition, the System/370 attached processor does not qualify as a true multiprocessor because the APU has no access to input or output. The consequences of this variant are discussed under *Serialization of I/O control on the CPU*, below.

Given the characteristics listed above, the two major focal points in designing a multiprocessing system are *sharing* and *interaction*. Control of shared resources can be accomplished in any of several ways. The three most basic to current multiprocessing design are serialization of the resource by means of a lock, replication of the shared resource, and restriction to running on a specific processor. Identification of the resource owner is also important. The processors interact at several levels of communication (such as memory, I/O bus, cache, and CPU signaling instruction).

The System/370 multiprocessing feature[7] provides for coordination of multiple processors. Prefixing hardware permits the first 4096 bytes of each processor's storage to be replicated. This area contains processor dependent locations such as those for program status words, logout, and general processor and program status. Locking is accomplished by the COMPARE AND SWAP (CS) instruction, which ensures serialization during its execution and provides a field for ownership identification.[8] Processor coordination is provided explicitly by the SIGNAL PROCESSOR (SIGP) instruction, and at several implicit levels (such as storage protect and cache). System/370 multiprocessing always deals with two processors of equal power.

software architecture

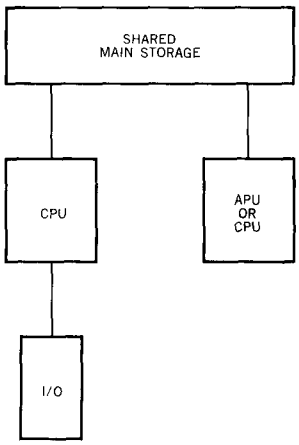
There are three alternative software architectures for multiprocessing hardware. There can be a master-slave relationship, there can be a separate executive for each processor, or each processor can be treated as a symmetric resource. The third is the

most useful and also the most difficult to implement. Its usefulness derives from its ability to support a general set of parallel functions without restricting what can run at any time on any processor. This generality requires a higher level of supervisor code and more care in implementation. A variation of it is the basis for attached processor support in VM/370.

Attached processor

The justification for multiprocessing support in VM/370 derived from the need for additional instruction processing power on large systems. The first design decision concerned the forms of System/370 multiprocessors that should be supported. Fully symmetrical multiprocessors[5] would have required parallelism throughout the control program. But by restricting I/O operations to one processor, as in an attached processor or asymmetric multiprocessor configuration, many fewer changes were required in VM/370. In making the choice to support the attached processor concept, it was clear that the system's applicability would be restricted to users with a requirement for additional CPU power only. These users, however, are a significant subset of all VM users. Typically they have many CMS systems or mixed CMS-production virtual machines. Compared with a uniprocessor, an attached processor does not provide for any additional I/O capability.

Figure 1 Logical hardware configuration



In an attached processor configuration, any functions that relate to real I/O (START I/O and interruptions, for example) can run only on the CPU. This limitation reduces the complexity of the design and ensures serialization within the I/O process. The real hardware configuration is shown conceptually in Figure 1.

Project constraints

The design of attached processor support in VM/370 Release 4 was undertaken under the following project constraints:

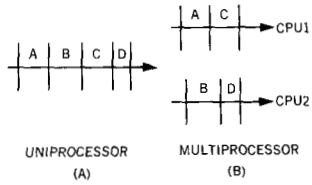
- The flexibility to experiment was constrained by the development process and its schedules.
- Compatibility with all VM/370 functions available in uniprocessor mode was to be maintained without significantly degrading uniprocessor performance.
- The integrity of all serial processes was to be maintained.
- Cost-justified performance was to be ensured.
- A base was to be created for the VM/370 Resource Management Program.[9]
- Control program modifications were to be minimized to reduce the rewriting required for user modifications. The objective was stability and minimal regression.
- The attached processor capability was to be an option which could be added to VM/370 during system generation.

- Any changes made for multiprocessing should have minimal effect on uniprocessor operation.

Fundamental multiprocessing problems and solutions

The central design problem in multiprocessing is selecting the means by which computing tasks are to be divided among the processors so that, for the duration of that unit of work, each processor has logically consistent instructions and data. Consider, for example, the stream of instructions executed by a uniprocessor, and suppose the instructions are divided into computing tasks as depicted in Figure 2(A). A task may be a single machine instruction or several thousand. Deferring for the moment the design issues associated with identifying these tasks, the central design question becomes: Given some division of the uniprocessor instruction stream into tasks, when is it permissible to overlap the execution of these tasks with two or more processors? Figure 2(B) depicts an overlapped relationship.

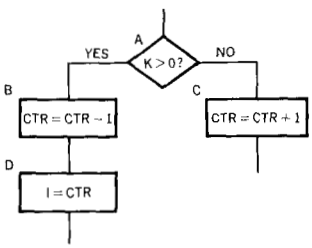
Figure 2 Task relationships



This question is closely related to a design issue in uniprocessing: Given two successive tasks, A and B, as shown in Figure 2(A), when is it permissible to reorder these tasks? For example, if task B is the processing of an I/O interruption and A is a task that normally runs disabled for I/O interruptions, what are the consequences of enabling for interruptions during task A, so that B can be executed before or during A rather than after it?

The reordering of two uniprocessor tasks may be disallowed by either a logical dependence or a data dependence of task B on task A. B is logically dependent on A if B may or may not be executed, depending on the execution of A. Tasks B and C in Figure 3 are logically dependent on task A, since either B or C may be executed, depending on the result of the execution of A.

Figure 3 Logical and data dependencies



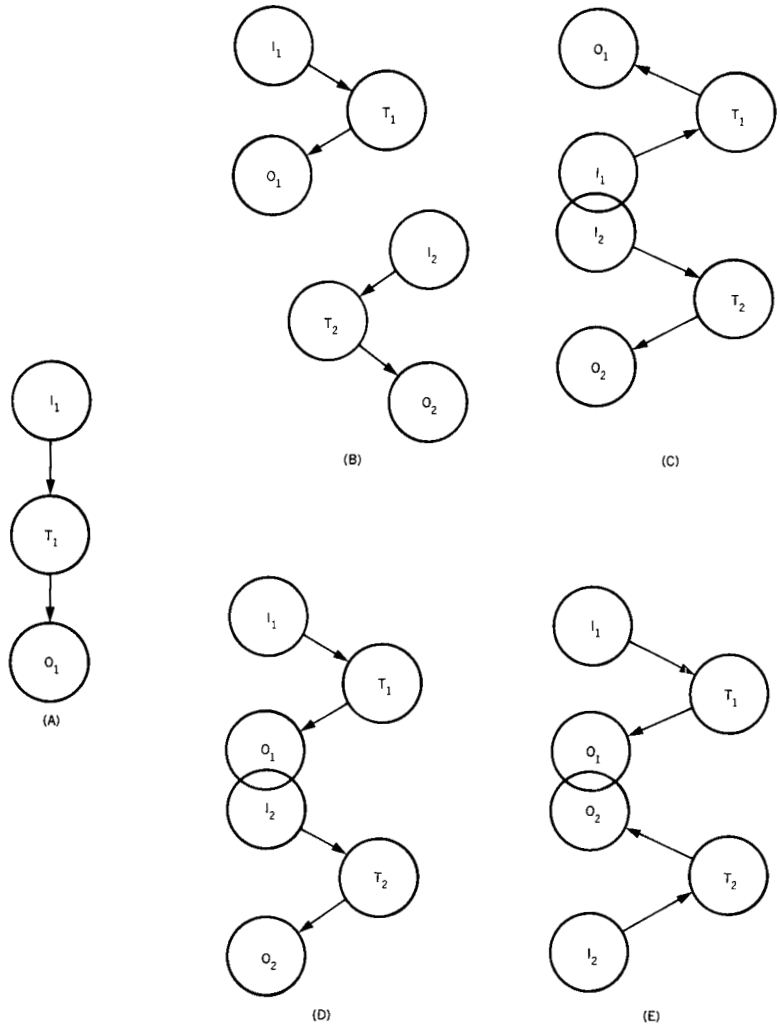
Data dependence is illustrated by tasks B and D in Figure 3. The output value of task B is the input to task D. Data dependence may occur in several other ways if the term *data* is broadened to include any storage location or register, or other task input or output.

If two sequential uniprocessor tasks are logically independent and data independent, they can be reordered. In multiprocessing systems, attention must be paid to both logical and data dependence when tasks are divided among processors.

data dependence

A task T_1 can be viewed as a mapping from input data I_1 to output data O_1 as shown in Figure 4(A). Parts (B), (C), (D), and (E) of Figure 4 illustrate various relationships between input and output data for two tasks. In 4(B) the data elements for the two tasks are

Figure 4 Multiprocessing data dependencies



disjoint. In 4(C) the input data overlaps. In 4(B) and 4(C) there would not be a problem for multiprocessing, but in 4(D) and 4(E) there could be inconsistent results. Therefore special precautions should be taken in instances such as these.

In the uniprocessor version of VM/370, system tasks presented many conflicts like those in 4(D) and 4(E) for any reasonable division of tasks. Indeed, the principal obstacle to design was the diffuse ownership of system data areas in a system not designed for multiprocessing.

Four general strategies are employed to cope with these data dependencies:

- Serialize the use of the shared data with a gatekeeper function or lock.
- Replicate the shared data items, assigning one set to each processor.
- Force all tasks that require a particular set of data items to run on a particular processor.
- Redesign tasks to minimize data conflict.

Aspects of VM/370 design which illustrate these strategies are discussed under *Design strategies*, below.

logical dependence

Much research into the theoretical problems associated with multiprocessing has concentrated on parallel execution within a single program. Baer[10] surveys some of this work. In the context of a single program, the problem of logical dependence is acute. Special language features are required to identify which portions of the program can be executed in parallel and to synchronize execution of the processors at various points in the program. Although this fine-granularity multiprocessing is an important theoretical and practical problem for some computing environments, many large systems do not require such solutions. These systems are characterized by competition for CPU resource among many tasks which, by necessity, have a large measure of logical independence. For example, two batch jobs or two time-sharing users often will be totally independent except when they require supervisor services.

Often a uniprocessor system provides a central function for queuing and dispatching of these independent tasks. A system may take advantage of this independence, for example, to reorder the tasks according to a user- or system-determined priority. The design for multiprocessing in VM/370 exploits the logical independence of the tasks on the central queues. This design choice also reduces the requirement for interprocessor communication. It is part of the new system but is required only rarely.

Design strategies

Locking

This section describes the locking structure of VM/370 attached processor support, the specific types of locks implemented, and their uses.

locking structures

The structure of software locks introduced into the VM/370 supervisor to support multiple processors allows different virtual machines to be in execution simultaneously on each of the processors. It does not, in general, allow simultaneous supervisor execution on behalf of those virtual machines. The design principle

used is that only one processor at a time can execute supervisor functions, except for selected paths. Specific supervisor paths that are used frequently and do not share much data with the bulk of the supervisor were programmed selectively to allow simultaneous execution on multiple processors.

A logical software lock is implemented by designating a word of storage as the physical lock. The unlocked and locked states correspond respectively to the zero and nonzero values of the word. When locked, the specific nonzero value identifies the processor that has acquired the lock. Acquisition of the lock is attempted by trying to replace a zero word value with a nonzero value using the System/370 COMPARE AND SWAP instruction, which is a proper hardware serializing primitive. The replacement occurs only if the current content of the word matches the zero value specified in the instruction. If multiple processors simultaneously perform the operation on a given word of zero value, only one is successful.

If the replacement is successful, the invoking processor can use all the serially reusable resources protected by the lock, and it will be the only user of those resources. Software convention ensures that they cannot be used unless the lock has been acquired. When it has finished using the resources, the processor releases the lock by placing a value of zero in the given word, thus allowing it to be acquired subsequently by any other processor.

If the replacement is unsuccessful, the processor can suspend execution of the unit of work that requires lock acquisition, or spin (that is, loop) on acquisition attempts by continuously testing the state of the lock until it is released and subsequently acquired. The proper course of action for software normally is related to the use of the serially reusable resources protected by a given lock. Thus a lock can be categorized as a *suspend lock* or a *spin lock*. Suspension usually implies the additional work of saving enough information about the current unit of work so that it can be resumed later, ensuring that it will be resumed when the unacquired lock is later released, and switching to some other unit of work. It would appear that spinning on attempted lock acquisition wastes processor power, but if spinning takes less time than the work introduced by suspension, it is the more economical alternative.

specific locks

The VM/370 supervisor's use of most data fields in control blocks is widely scattered throughout the system control program. To protect those fields as serially reusable resources, the processor normally acquires one system lock upon entering the supervisor state at one of the first-level interruption handlers, and it releases the lock upon exiting from the central dispatcher to either virtual machine execution or the wait state. If this lock cannot be acquired, the processor suspends execution by saving the state of

the current task for later resumption and proceeding directly to the dispatcher to perform work that does not require the lock (that is, it puts another virtual machine into execution). Thus this system lock serializes the use of most supervisor resources.

Each virtual machine is serialized by a unique lock, which is acquired before a processor puts a virtual machine into execution or performs supervisor functions on its behalf. Its lock is released, normally in the dispatcher, when the processor stops servicing the machine. Transition of a processor from executing a virtual machine (problem state) to performing supervisor functions for it (supervisor state) does not involve any change in the state of a virtual machine lock. The lock was acquired in the dispatcher before virtual machine execution began and will be held through the supervisor state until deliberately released. Suspension of execution consists merely of bypassing selection of this particular virtual machine in favor of another whose lock can be acquired.

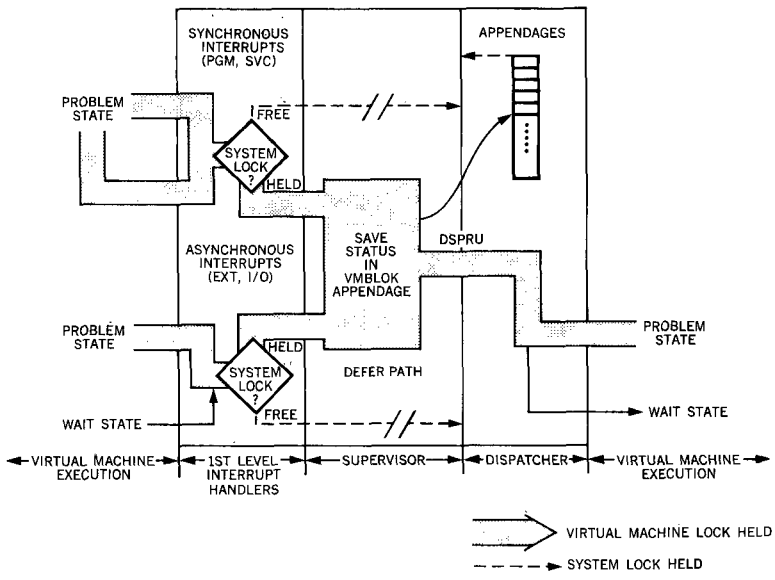
As stated above, most supervisor resources are serialized by the system lock. Resources used before system lock acquisition is attempted, or as part of the suspension mechanism if acquisition fails, are exceptional resources that require separate serialization. Most are queues (free storage blocks, runnable users, timer requests, etc.) which are updated as elements are inserted or deleted. Since updating a queue is a relatively brief operation, these queue locks are spin locks.

Some data items are locked by the data itself. Event counters (to be incremented by one) and the pointer to the current system trace table entry (to be advanced by each trace event) are single storage words which are updated directly by COMPARE AND SWAP instructions. The replacement value of such words is uniquely determined by the current value. If simultaneous replacement is attempted by two processors, one will fail, causing a redetermination of the replacement value and another attempt. This process is a form of spin lock.

the selected paths

Detailed description of a selected path should clarify the interplay among locks. Figure 5 provides an overview of the relationships among various portions of the supervisor. Assume that a processor has been directly executing a virtual machine which has just attempted to execute a privileged instruction (for example, START I/O). Since virtual machines are managed by forcing direct execution in problem state and simulating privileged operations, the hardware causes a privileged-operation program interruption. Thus supervisor software is invoked at the first-level program interruption handler. The processor holds the virtual machine lock of the virtual machine that just caused the interruption; it does not hold any other locks. The contents of the processor's registers and of the program-interruption old program status word that

Figure 5 Supervisor path overview



identifies the virtual state can be saved in control block areas serialized by the virtual machine lock. The virtual machine is flagged as nonrunnable until simulation by supervisor functions is complete.

When supervisor functions determine that system resources other than those specific to the virtual machine are necessary to complete the simulation, an attempt is made to acquire the system lock. If the attempt is successful, the processor proceeds through the balance of the supervisor while holding both the system lock and the virtual machine lock. It may also temporarily hold one of the spin locks on specific queues. When the processor eventually reaches the dispatcher, it still holds the system and virtual machine locks. At this point the processor either redispatches the virtual machine or performs some other unit of work. Redispatching the virtual machine means that the system lock is released, and the processor returns to problem state still holding the virtual machine lock. To perform other system work, the virtual machine lock must be released and the system lock retained.

If the system lock cannot be acquired when needed, the processor follows the defer path, as shown in Figure 5. Deferred service requires that the logical point in the simulation and any interruption information be saved. General register and instruction counter contents are stored in a control block appendage that defines the virtual machine. This appendage is then queued on a chain of system requests to be processed when the system lock has been acquired. (A spin lock on this chain is held briefly while the queuing is performed.) Control is then passed to a special

DSPRU entry point in the dispatcher, where the current virtual machine lock is released and acquisition of another is attempted. Exit from the dispatcher is either to problem state (holding a virtual machine lock), or to wait state (holding no locks). The DSPRU entry point is the start of a special path through the dispatcher which has no need of the system lock. System-wide resources such as free storage queues, which must be used by this path, are serialized by separate spin locks.

As can be seen from the above discussion, most state changes in suspend locks occur in the dispatcher. Entry into the dispatcher normally represents the termination of some unit of work, and exit represents the initiation of another. One way of exiting from the dispatcher is to execute a system request that has been queued by another supervisor function. Such requests, of which the defer path appendage is one instance, are invoked while the system lock and the pertinent virtual machine lock are held.

Some selected paths in the synchronous first-level interruption handlers (supervisor call and program check interruptions from problem state) were programmed to operate without acquiring the system lock if they ended by returning to problem state execution of the same virtual machine. Such paths were constrained to use only those system resources serialized by the virtual machine lock held on entry or by specific spin locks. System performance was enhanced by these paths because of their frequency of use.

Serialization of I/O control on the CPU

As discussed above, an attached processor system is a multiprocessor with all I/O devices attached to one processor, so that all I/O activity is serialized on that processor. Only the CPU can initiate input or output (the APU responds to all I/O instructions with a condition code that indicates that the addressed device is not operational) and only the CPU can respond to an I/O interruption. In other words, only the CPU can execute the code in the system control program's first-level I/O interruption handler.

Reviewing briefly the I/O handling logic of CP, recall that the I/O supervisor performs the following three functions:

- Accepts I/O requests made by other components of CP. These requests are enqueued from the appropriate I/O control block, and an I/O operation is started if a path to the requested device is free.
- Accepts I/O interruptions. If an interruption indicates completion of an I/O task, the supervisor passes that task to the dispatching module of the system.
- Dequeues and starts one or more new I/O tasks if, as a result of an I/O interruption, a path to a requested device becomes free.

The I/O supervisor module in CP is essentially a self-contained package of code which contains all of CP's I/O interruption handling logic. Although its internal logic is somewhat complex, its interactions with the remainder of the system are well defined. However, it does not control all the initiation of I/O in CP. There are many code modules that initiate I/O, and the interactions of these modules with the remainder of the system are complex. In particular, at some point, many of the various terminal handling routines attempt to initiate I/O to a terminal.

I/O design

In view of the I/O handling considerations outlined above, the design problem for input and output has three aspects:

- Ensuring that only the CPU will initiate I/O.
- Ensuring that the I/O control blocks are not changed by the attached processor in such a way as to cause the CPU to make an error.
- Ensuring that the interactions between I/O handling and the remainder of the system are suitably interlocked.

The major decision in designing I/O support for VM/370 was that only the CPU would execute code that made use of the dynamically changed real I/O control structure. The intent of this decision was to ensure that all I/O interruption processing could be done without the system lock, for, in effect, there is no multiprocessing of the I/O logic. All other paths that share data with this unlocked path must be serialized on the CPU. The result is that all paths, other than the I/O interruption handler, that refer to the I/O control blocks must operate with the double constraint of being controlled by the system lock and running on the CPU. The I/O interruption path shares, with other paths in the system, the following resources: fields in the I/O control blocks, fields in the other system control blocks, and system free storage and queue pointers.

I/O control block considerations

Consider all dynamically changed fields in the real I/O control blocks that were used (that is, modified or referred to). All paths that use these fields, other than the interruption handler, must be switched to the CPU. This was accomplished by defining a SWITCH macroinstruction and adding an AFFINITY option to the CALL macroinstruction, so that one routine could call another with assurance that the called routine would return to the same processor.

Next, all sections of the system that used the I/O control structure, other than the I/O interruption handling path, were examined to find (or create) closed paths of execution that could be switched onto the CPU by the SWITCH macroinstruction. The macroinstruction was invoked at the beginning of the path and, if necessary, an AFFINITY option was added to each CALL within each path.

The design rules for restricting all I/O control to the CPU have one exception, in that the page device manager manipulates the queue of drum-storage page requests so that they are slot sorted. This is done only for the paging drums so that one I/O task will cause several page I/O operations, without the delays inherent in separate START I/O instructions for each paging operation. In particular, if an I/O task is already active on the paging drum, the page device manager places the current request in its proper place in the queue of tasks that will be started when the current I/O task finishes. On the other hand, if the drum is idle, the page device manager passes its I/O request by calling the I/O supervisor.

The design problem caused by this exception is that the page device manager manipulates the I/O control structure, and to conform to the design rule stated above, this portion of the code would have to be switched to the CPU. As a result, performance could be severely degraded. Consider the following situation:

The APU, operating with the system lock, has finished processing a page fault. All that remains to be done is to insert a request into a queue. But since the processing is being done by the APU (not the CPU), a SWITCH must be performed. The delay caused in switching to the CPU more than offsets the performance advantage of placing the I/O request in the queue.

The solution chosen was to define a spin lock for the task queue associated with each real device. This lock ensured mutual exclusion of the dequeuing of tasks by the I/O interruption handler and the enqueuing of tasks by the page I/O manager, while allowing the high performance advantage of slot sorting page I/O requests.

system control block and system queuing considerations

The remaining major concern in this part of the design was the requirement that access to system resources be shared by the interruption path and the rest of the system. Since the major design decision for I/O was that only the CPU would refer to I/O control blocks, the I/O interruption handler did not need the system lock. Two other resources are shared by the interruption path and the rest of the system. They are the scheduler code (protected by the system lock) and fields in the virtual machine control block (protected by virtual machine lock). The following example illustrates the problem.

An I/O interruption is received, indicating completion of one I/O task and freeing a path to the I/O hardware. Therefore one or more other I/O tasks can be started. Performance considerations require that restarting of the I/O hardware not be delayed. At least three virtual machines are involved in the processing:

- The machine that was executing when the interruption was received.
- The machine for which the interruption is destined.
- One or more machines waiting for a START I/O instruction to be issued; as a result of this interruption, the machines can have their I/O started and become candidates for being run.

The existing logic of the interruption handler was to place the completed I/O task in the queue of completed tasks. Next, the interruption handler restarted the I/O hardware and made one or more virtual machines dispatchable by turning off a bit in their virtual machine control blocks and calling the scheduler to make them eligible for dispatching. For attached processor support, this logic presents two problems: the field in the virtual machine control block is protected by a suspend lock, and the scheduler has to operate while holding the system lock (also a suspend lock). In other words, the I/O interruption handling path contained two points at which suspend locks were required, yet the design goal was to have an interruption handling path that was unlocked.

The solution was to replicate a portion of the scheduler function in the I/O interruption handling path, thereby removing the requirement for the system suspend lock. The other suspend lock requirement cannot be avoided. Thus, an attempt is made to lock the virtual machine. If the attempt is successful, the replicated portion of the scheduler function is executed, and the virtual machine is unlocked. If the lock cannot be obtained, a CP execution request block is built and stacked. As discussed under *Locking*, above, the dispatcher ensures that this request is executed with the associated virtual machine control block locked.

free storage and system queuing considerations

The I/O interruption path required access to the two task execution queues (for CP execution request and I/O task processing). These queues are locked, and access to them is controlled (serialized) by a central routine. Note that the locks are spin locks, so control is never lost as a result of queuing an element on them.

Finally, the I/O interruption path requires free storage to handle I/O errors. By design, free storage is protected by a spin lock. Thus, on receipt of an I/O interruption that indicates an error, the CPU calls for free storage, and only if the APU is currently getting or returning free storage will the CPU spin momentarily before continuing.

an illustrative example

The discussion above concentrates more on what the I/O control logic does than on how it treats any one task. It is perhaps helpful to step through the sequence of events when the APU is running a virtual machine and encounters a START I/O instruction. When START I/O is executed, the APU executes the logic in the program interruption handler. This includes a sequence of checks to deter-

mine that a virtual machine was running, a privileged instruction execution was attempted, and the virtual machine was in supervisor state.

The APU executes the logic to decode the instruction. When it has been determined that the instruction is a START I/O, the processor must acquire the system lock before proceeding further. If the system lock cannot be acquired, the virtual machine is blocked, a deferred execution task is stacked, and the APU finds and runs another virtual machine.

As an example, assume that the system lock is free. The APU acquires the lock and proceeds with START I/O simulation for the virtual machine. The APU checks the virtual device address, builds an I/O task block, translates the addresses in the channel command words from the address space of the virtual machine to the real address space of the system, and finally passes the I/O task to the I/O supervisor for processing.

It is important to note that until this point, no reference has been made to the real I/O control blocks. It is in the I/O supervisor that the SWITCH macroinstruction determines whether it is the CPU or APU that is executing the code. When it is the APU, as in this example, supervisor call (SVC) 24 (new to APU support in CP) is executed. Processing of this supervisor call entails building a task execution block and stacking it for execution by the CPU. After the task block has been stacked, the APU goes to the dispatch routine, where it tries to run another virtual machine. No signal is sent to the CPU; rather, the CPU encounters this switched I/O task in its normal course of events (that is, on its next trip through the locked supervisor).

When this switched task is unstacked by the CPU, control passes to the instruction that follows SVC 24. Now the CPU is executing the logic, and the supervisor lock is held, permitting access to the real device blocks and the real I/O hardware. The CPU queues the I/O task and, when the real device is available, starts the I/O operation. Simulation of the virtual machine's START I/O is now complete (although the I/O itself has not yet been completed). The CPU marks the virtual machine as runnable and either proceeds with other I/O supervision or enters the dispatcher. In any case, the virtual machine is run by the CPU or the APU according to which gets to it first.

Shared segment support

VM/370 supports the sharing of read-only virtual address space among several virtual machines. The units of read-only sharing are 64K-byte segments, which can be discontiguous. The principal benefits of sharing segments are that less main storage is re-

quired, less space is required on secondary paging devices, and interactive responsiveness is improved because fewer paging operations are needed to complete a command. The major design and implementation problem is not the sharing of address space, but ensuring its read-only integrity and properly treating the virtual machine that has changed it (or attempted to change it).

Historically, two approaches to ensuring integrity have been employed in the various releases of VM/370. They are key-based and change-bit-based storage protection.

key-based protection

Releases 1 and 2 of VM/370 utilized storage protection keys so that System/370 hardware prevented any change to the shared segments. Read-only pages were placed in key 0, and the virtual machine was never allowed to run with key 0 or change the keys of the shared segments. Since CMS was using the keys to protect other parts of storage, however, CP and CMS were, in effect, sharing the storage protection keys. CP cannot ensure that the virtual machine (or its user) will abide by any convention for key use, so CP had to simulate key 0 execution for virtual machines with shared segments. This requirement led to the use of a key-flipping algorithm, which increased CP overhead by approximately ten percent. Further, it precluded the use of virtual machine assist hardware[11] because that hardware allows a virtual machine to switch its execution key without CP's intervention. For shared segment integrity, however, the virtual machine must never execute in key 0.

change-bit-based protection

Release 3 of VM altered the manner of ensuring integrity by using a change bit to detect (after the fact) a change to a read-only page. CP scans these change bits each time it dispatches another virtual machine. Any modification of a read-only page can thus be identified, and the virtual machine that caused the change can be given a private copy of the segment. The result is that no other virtual machine is affected by the change. The cost of this technique is that about ten percent of CP's execution time is devoted to scanning the change bits. Note that this cost increases with the number of active shared segments. The significant benefit of this approach, compared with the key-based technique, is that virtual machine assist hardware can be used in running virtual machines with shared segments.

multiprocessing considerations

As with uniprocessing, the multiprocessor design problem is to ensure the integrity of shared segments. That is, no CP design problems are created by allowing segment sharing. The technical issue becomes whether the shared segments are to be protected by keys (that is, read-only) or checked for change after the fact.

The most important distinction between these approaches is whether the segments can be shared simultaneously. Only if the

shared segments are protected by keys can the CPU and APU simultaneously run virtual machines that share access to a single copy of one (or more) segments. The following example illustrates the problem.

If the APU and CPU both have been running virtual machines that have simultaneous access to a segment of virtual storage, and if a change bit is found to be on, then two questions arise: Which virtual machine is responsible for the change? Which virtual machine gets a private copy of the shared segment? In short, usually it is unacceptable to have simultaneous sharing without read-only protection. The alternative is not to share segments simultaneously. The need for simultaneous sharing can be avoided by duplicating the resource—that is, by providing one copy (or partial copy) of each segment for each processor. An alternative is to establish a lock structure to ensure serialization of the resource.

The design choice that was made was to provide change bit scanning and to duplicate the shared segments. As a result, both APU and CPU have segments (or portions thereof) available for sharing by the virtual machines that each dispatches, and upon switching from one virtual machine to another, each processor can identify the virtual machine that has modified its copy of the shared segment.

Once this design decision was made, the remaining problems were restricted to implementation. The most important was what to do when there was a changed page in a shared segment. The design constraints were such that upon detecting a changed page in a shared segment, CP would give the offending virtual machine a private copy of the now changed segment and allow that machine to continue execution. An implementation problem arises because the scan for changed shared segments is part of the defer path (that is, the unlocked supervisor code, as discussed under *Locking*, above). The unsharing of a changed shared segment must be part of the locked supervisor code because unsharing requires extensive changes to the page and segment tables.

The implementation problem was how to make the unsharing of shared segments a completely deferrable task. The problem was solved by dividing a single process (scanning for an unsharing of changed segments) into disjoint processes of scanning and unsharing.

scanning

Scanning for changed pages in shared segments is performed in one step (with no loss of control) without requiring the system lock. After scanning has been completed, the system (virtual machines, core tables, etc.), is in a state that allows continued access to the shared segments (except for the changed pages).

When changed pages are detected, each is marked *invalid* and suitable changes are made to core-, swap-, and page-table entries. These changes allow for full reconstruction of the offending user's address space, even after other virtual machines that have run with the shared segment have changed pages in it.

Finally, the system acquires and stacks a CP execution request that this user's address space be unshared from the shared copy to a private copy. This task can be deferred for an arbitrary period.

unsharing

The unsharing process is a separate and deferrable task that runs with the system lock. That is, at some point, the CP execution request that was stacked when the change to a shared segment was detected will be unstacked. Execution will then be on the same processor that owned the shared segment and will be controlled by the system lock. The unsharing process constructs new segment and page tables for each shared segment changed by the offending virtual machine. Finally, the unsharing process places in these new segments the pages that were changed by that virtual machine (thereby removing them from the shared segment).

This approach is in contrast to that of VM/370 Release 3, in which the offending virtual machine was given the shared-segment page tables, and a new shared-segment page table was constructed.

Performance

performance measurement and the design process

Performance is always an important design consideration in software systems, and in attached processor support for VM/370 it is the central issue. The principal aim of that support is to increase the processing capability of VM/370 systems. Consequently, during the development cycle, it was considered essential to have detailed performance data to aid in making design choices and to refine implementation details in line with overall performance objectives.

Performance measurements were obtained in two ways. First, a prototype system was constructed and measured so that the main design approach could be validated. In addition, a benchmark was designed to duplicate key elements of the CPU-bound environments in which attached processors would be required. The benchmark was run repeatedly with interim versions of the final system. In each case, the system was thoroughly instrumented with software and hardware monitors. These measurements included the detailed distribution of supervisor state time across the various modules of the system. The measurements were used primarily to refine implementation details. However, as illustrated below, the measurements also provided the basis for some design alterations.

The following sections describe some of the performance insights gleaned during the development cycle.

quality measures and load environments

It is customary to rate the effectiveness of multiprocessing software by comparison with uniprocessor performance. Frequently the comparison is summarized with a single ratio or range, but this simplification masks four major dependencies:

- The performance variables or figures of merit used.
- Load dependencies.
- Hardware effects.
- The software system itself.

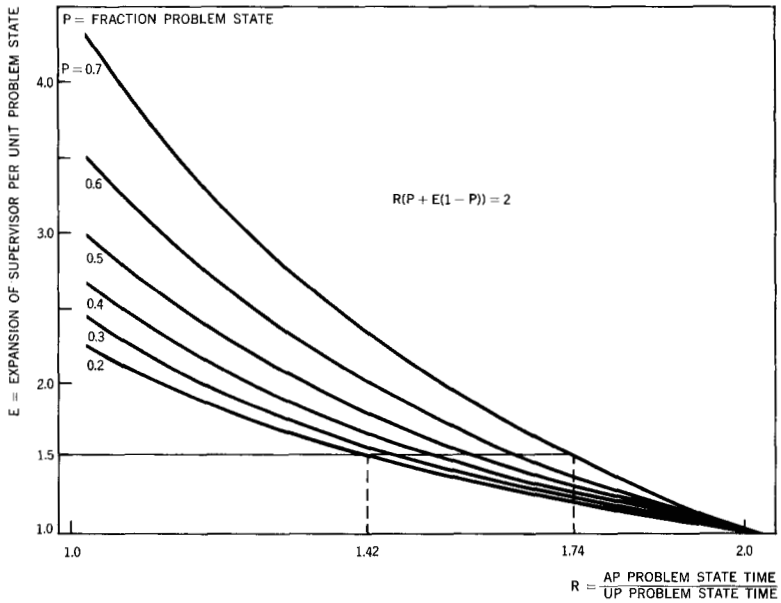
Consider the problem of selecting key performance measures. VM/370 can support diverse computing environments, including both batch processing and time sharing. Response time and throughput are suggested as key variables. Response time generally is measured in terms of the total elapsed time required to process commands or transactions entered at a terminal. Typically in VM/370 systems, most such commands make only a modest demand on the CPU. In addition, the scheduler attempts to order these interactive tasks ahead of longer running tasks. Consequently, average response time tends to reflect paging delays and I/O delays rather than CPU use or contention, even though the CPU may be saturated. Of course there are exceptions.

On the other hand, the aggregate throughput of the system may well be constrained by the available CPU resource. Bard[12] suggests means to detect and measure the degree of this constraint. A convenient measure of throughput is total problem state time, since all virtual machine execution is in problem state. The ratio of this quantity between multiprocessor and uniprocessor systems is a reasonable figure of merit with which to evaluate multiprocessing software designs. A poor design increases supervisor state time and thus decreases this number.

This quality measure has two disadvantages, however. First, supervisor state should not be considered pure overhead. The system provides storage management, I/O service, command processing, etc. Thus the supervisor state also contains useful work. Second, an attempt to isolate the increase in supervisor state time due to multiprocessing support shows that there may be an equivalent increase in supervisor overhead for widely varying values of the problem state ratio, depending on the relative distribution between problem and supervisor states.

For example, let P be the problem state fraction for a uniprocessor benchmark. Then, if the CPU is saturated, $1 - P$ is the supervisor state fraction. Supervisor state per unit problem state is then $(1 - P) \div P$. The expansion, E, of this quantity in a multi-

Figure 6 Problem state ratio versus supervisor expansion



processing system is a first-order measure of increased overhead. If the ratio, R, of multiprocessing to uniprocessing total problem state time is taken as the figure of merit, then the best-case value of R can be computed from the following equation:

$$R(P + E(1 - P)) = 2$$

The right-hand side of this equation reflects the fact that maximum throughput occurs when both processors are saturated.

This equation defines a family of curves for various distributions of uniprocessor problem state and supervisor state, as shown in Figure 6. Thus, as a first-order effect, a supervisor expansion of 1.5 yields a problem state ratio ranging from 1.42 to 1.74 as the problem state fraction varies from 0.2 to 0.7. As discussed below, other factors prevent high supervisor state environments from achieving maximal throughput.

Both supervisor expansion and the problem state ratio from our measurements are given in Table 1. Taking hardware effects into account, these quality measures are further refined.

effects of design decisions

The most obvious effect of any additional function is to increase path lengths in the supervisor. In the attached processor implementation, this effect comes primarily from the additional work necessary to defer a virtual machine when the system lock is required but unavailable. Additional work also is required to re-

Table 1 Illustrative performance comparisons

Ratio of problem state time	1.77
Ratio of problem state instructions	1.63
Problem state instruction rate uniprocessor (MIPs)	1.18
Problem state instruction rate multiprocessor (average MIPs)	1.09
Expansion in supervisor time per unit problem state	1.46
Expansion in supervisor instructions per unit problem state	1.27
Supervisor instruction rate uniprocessor (MIPs)	0.92
Supervisor instruction rate multiprocessor (average MIPs)	0.80

sume that deferred task. Path lengths also increase somewhat because of the need to acquire and release local locks, but generally this requirement has had only a modest effect on the path lengths of most modules. One outstanding exception is the free storage handler, in which the original paths were so short that just the addition of the LOCK macroinstructions is significant. Measurements thus far have shown that spin lock contention is quite small, accounting for less than one percent of supervisor time. This result, which had been obtained also on the prototype system, served to confirm the expected low contention for the spin locks.

As noted above, a processor that performs either supervisor or problem state execution for a virtual machine must hold the virtual machine's lock. As a consequence, there is no performance gain if only one virtual machine is in the system, since there would be no overlapped execution.

A portion of the supervisor executes without acquiring the system lock. This portion is concentrated primarily in the first-level interruption handlers and the dispatcher. Most other supervisor work, such as storage management, I/O simulation, and command processing, operates under the system lock. As a consequence, load environments characterized by high supervisor state execution may perform less well under this design than load environments characterized by a higher percentage of problem state execution. In such environments, the attached processor tends to become idle as the main processor handles stacked supervisor tasks that require the system lock.

Generally the APU becomes idle, rather than the CPU, because some of the stacked supervisor requests involve I/O that can be executed only on the CPU, and these tasks are shifted to the CPU. When the CPU holds the system lock, on the other hand, there are few tasks that can be executed only on the APU. Consequently, once the CPU obtains the system lock, it tends to hold it much longer than the APU. This tendency also accounts for a substantial

migration of supervisor state execution to the CPU. The lock design, therefore, has biased the system in favor of multiprogramming environments characterized by a higher percentage of problem state than supervisor state.

Replication of shared resources as a design strategy can make for slower handling of shared segments. The possible effect of this replication on storage should be evaluated on a case-by-case basis.

As noted above, design changes occasionally were dictated by feedback from measurements made during the development cycle. Accounting provides a good example. Supervisor time is accumulated for users in a data field in the virtual machine control block (VMBLOK). Ordinarily it would be guarded by the virtual machine lock. However, in a few places in the system it is desirable to be able to charge supervisor time to a virtual machine without having to acquire its lock. For example, I/O interruption processing is charged to the owner of the I/O task, even though that user may be executing on the other processor at the time of the interruption. The original design attempted to serialize the use of the supervisor time field by using the synchronizing primitive COMPARE DOUBLE AND SWAP in a common subroutine. This approach was found to cost approximately eight percent of supervisor time for this function alone. Replication of this accounting field, at some small cost in storage, virtually eliminated this overhead. The two separate accounting fields simply are combined when the total is required for an accounting record.

effects of multiprocessing hardware

Aggregate problem state time has been suggested as one measure of the throughput of a multiprocessing system. A more accurate measure would take into account any change in the instruction rate of the machine. Because of hardware memory interference effects, instruction rates of processors in a multiprocessing configuration are somewhat lower than the equivalent uniprocessor rate. For this reason, the aggregate number of problem state instructions is a more accurate measure of throughput. Similarly, the expansion in supervisor state time per unit problem state is caused in part by increased path length and in part by hardware slowdown.

When comparing uniprocessor and multiprocessor performance, it is reasonable to assume that the mix of problem state instructions is constant. On the other hand, the mix of supervisor instructions could change between uniprocessor and multiprocessor measurements. Thus the difference between supervisor time and the number of supervisor instructions as a measure of additional overhead is caused partly by multiprocessing hardware effects and partly by a possible change in the instruction mix. Measurements of the effect of various instruction streams on the

performance of multiprocessing hardware should provide a fertile area for further experimentation. For a discussion of some additional hardware effects in the context of MVS measurements, see White.[13]

illustrative results

The performance effects discussed above are illustrated by the data in Table 1. Caution should be observed in any attempt to extrapolate the data. All measurements were made on a three-megabyte System/370 Model 158. The load consisted of 80 CMS users executing various scripts repetitively. The uniprocessor problem state was approximately 65 percent. The load was adequate to saturate both processors in the multiprocessor runs. All comparisons are between the multiprocessor system and the uniprocessor system prior to implementation of multiprocessor support. Somewhat different results would be obtained in a comparison based on uniprocessor measurements of the system level that supports attached processors.

As Table 1 shows, the attached processor system produced 1.77 times the problem state time of the uniprocessor system. This result is consistent with other measurements, which have been in the range 1.5 to 1.8. Because of hardware slowdown, these measurements yielded a ratio of 1.63 in terms of problem state instructions. These numbers also indicate the order of magnitude of the increase in path lengths in the supervisor as measured by supervisor instruction-count inflation.

The magnitude of these hardware effects clearly illustrates the need for data from hardware monitors in evaluating multiprocessor systems.

Concluding remarks

In undertaking an effort like the one described herein, there are many decisions to be made and pitfalls to be avoided. It is hoped that some insight has been gained into the problem of adding a major new function to an operating system that did not provide for that function originally. It is gratifying to see the results of that labor operating in production environments and performing up to its objectives.

ACKNOWLEDGMENTS

The authors thank all the people involved in this effort. In particular, we thank Charles Weagle for his attached processor prototype, which provided the basis for this work, and Marjorie Schong for the professional work of her development group. We also recognize Ronald Reynolds for his reworking of the VM/370 Resource Manager PRPQ base code in Release 4. Finally, we acknowledge the efforts of R. A. MacKinnon, who provided encouragement and direction for the VM/370 attached processor.

CITED REFERENCES

1. L. H. Seawright and R. A. MacKinnon, "VM/370—a study of multiplicity and usefulness," *IBM Systems Journal* **18,** No. 1, 4–17 (1979, this issue).
2. A. L. Scherr, "Functional structure of IBM virtual storage operating systems—Part II: OS/VS2-2 concepts and philosophies," *IBM Systems Journal* **12,** No. 4, 382–400 (1973).
3. J. S. Arnold, D. P. Casey, and R. H. McKinstry, "Design of tightly-coupled multiprocessing programming," *IBM Systems Journal* **13,** No. 1, 60–87 (1974).
4. *IBM Time Sharing System Concepts and Facilities*, IBM Systems Library, order number GC28-2003, IBM Corporation, Department 80M, 1133 Westchester Avenue, White Plains, New York 10604.
5. H. Lorin, *Parallelism in Hardware and Software: Real and Apparent Concurrency*, Prentice Hall Inc., Englewood Cliffs, New Jersey (1972).
6. P. H. Enslow Jr., "Multiprocessor organization—a survey," *ACM Computing Surveys* **9,** No. 1, 103–129 (March 1977).
7. R. P. Case and A. Padegs, "Architecture of the IBM System/370," *Communications of the ACM* **21,** No. 1, 73–96 (January 1978).
8. *IBM System/370 Principles of Operation*, IBM Systems Library, order number GA22-7000, IBM Corporation, Department D58, P.O. Box 390, Poughkeepsie, New York 12602.
9. *VM/370 Resource Management Programming RPQ PO-9006 Installation Guide*, IBM Systems Library, order number SH20-1906, IBM Corporation, Department 825, 1133 Westchester Avenue, White Plains, New York 10604.
10. J. L. Baer, "A survey of some theoretical aspects of multiprocessing," *ACM Computing Surveys* **5,** No. 1, 31–80 (March 1973).
11. R. A. MacKinnon, "The changing virtual machine environment: Interfaces to real hardware, virtual hardware, and other virtual machines," *IBM Systems Journal* **18,** No. 1, 18–46 (1979, this issue).
12. Y. Bard, "Performance analysis of virtual memory time-sharing systems," *IBM Systems Journal* **14,** No. 4, 366–384 (1975).
13. W. White, *Attached Processing (AP) System Performance Characteristics and Considerations*, IBM Washington Systems Center Technical Bulletin No. GG22-9004, IBM Corporation, Building 2, 18100 Frederick Pike, Gaithersburg, Maryland 20760 (May 1977).

Reprint Order No. G321-5086.