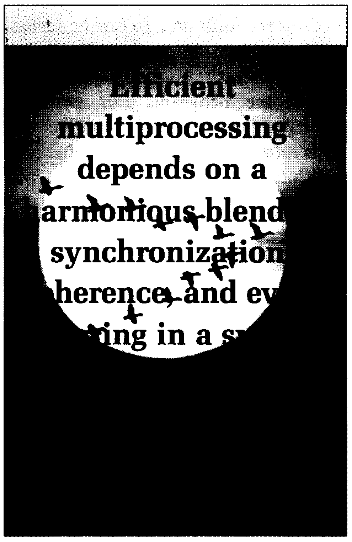
SURVEY & TUTORIAL SERIES

Synchronization, Coherence, and Event Ordering in Multiprocessors

Michel Dubois and Christoph Scheurich
Computer Research Institute, University of Southern California

Fayé A. Briggs
Sun Microsystems

Multiprocessors, especially those constructed of relatively low-cost microprocessors, offer a cost-effective solution to the continually increasing need for computing power and speed. These systems can be designed either to maximize the throughput of many jobs or to speed up the execution of a single job; they are respectively called *throughput-oriented* and *speedup-oriented multiprocessors.* In the first type of system, jobs are distinct from each other and execute as if they were running on different uniprocessors. In the second type an application is partitioned into a set of cooperating processes, and these processes interact while executing concurrently on different processors. The partitioning of a job into cooperating processes is called *multitasking*[1]* or *multithreading*. In both systems global resources must be managed correctly and efficiently by the operating system. The problems addressed in this article apply to both throughput- and speedup-oriented multiprocessor systems, either at the user level or the operating-system level.



Multitasked multiprocessors are capable of efficiently executing the many cooperating numerical or nonnumerical tasks that comprise a large application. In general, the speedup provided by multitasking reduces the turnaround time of a job and therefore ultimately improves the user's productivity. For applications such as real-time processing, CAD/CAM, and simulations, multitasking is crucial because the multiprocessor structure improves the execution speed of a given algorithm within a time constraint that is ordinarily impossible to meet on a single processor employing available technology.

Designing and programming multiprocessor systems correctly and efficiently pose complex problems. Synchronizing processes, maintaining data coherence, and ordering events in a multiprocessor are issues that must be addressed from the hardware design level up to the programming language level. The goal of this article is not only to review these problems in some depth but also to show that in the design of multiprocessors these problems are intricately related. The definitions and concepts presented here provide a solid foundation on which to reason about the logical properties of a specific multiproces-

*Multitasking is not restricted to multiprocessor systems; in this article, however, we confine our discussion, with no loss of generality, to multitasking multiprocessors.

0018-9162/88/0200-0009$01.00 © 1988 IEEE

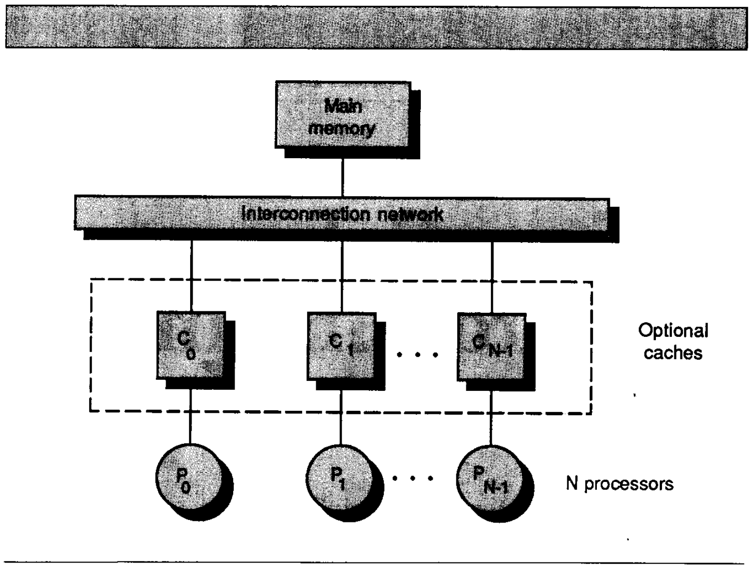


Figure 1. A shared-memory multiprocessor with optional private caches. The interconnection network may be either a simple bus or a complex network.

sor and to demonstrate that the hardware adheres to the logical model expected by the programmer. This foundation aids in understanding complex but useful architectures such as multiprocessors with private caches or with recombining interconnection networks (Figure 1).[2] Other important issues, such as scheduling and partitioning, have been addressed in a previous survey article.[3] Readers who are not familiar with the concept of cache memory should consult the survey by Smith.[4]

Basic definitions

The instruction set of a multiprocessor usually contains basic instructions that are used to implement synchronization and communication between cooperating processes. These instructions are usually supported by special-purpose hardware. Some primary hardware functions are necessary to guarantee correct interprocess communication and synchronization, while other, secondary hardware functions simplify the design of parallel applications and operating systems. The notions of synchronization and communication are difficult to separate because communication primitives can be used to implement synchronization protocols, and vice versa. In general, *communication* refers to the exchange of data between different processes. Usually, one or several sender processes transmit data to one or several receiver processes. Interprocess communication is mostly the result of explicit directives in the program. For example, parameters passed to a coroutine and results returned by such a coroutine constitute interprocess communications. *Synchronization* is a special form of communication, in which the data are control information. Synchronization serves the dual purpose of enforcing the correct sequencing of processes and ensuring the mutually exclusive access to certain shared writable data. For example, synchronization primitives can be used to

(1) Control a producer process and a consumer process such that the consumer process never reads stale data and the producer process never overwrites data that have not yet been read by the consumer process.

(2) Protect the data in a database such that concurrent write accesses to the same record in the database are not allowed. (Such accesses can lead to the loss of one or more updates if two processes first read the data in sequence and then write the updated data back to memory in sequence.)

In shared-memory multiprocessor systems, communication and synchronization are usually implemented through the controlled sharing of data in memory.

A second issue addressed in this article is *memory coherence*, a system's ability to execute memory operations correctly. Censier and Feautrier define a coherent memory scheme as follows: "A memory scheme is coherent if the value returned on a Load instruction is always the value given by the latest Store instruction with the same address."[5] This definition has been useful in the design of cache coherence mechanisms.[4] As it stands, however, the definition is difficult to interpret in the context of a multiprocessor, in which data accesses may be buffered and may not be atomic. Accesses are buffered if multiple accesses can be queued before reaching their destination, such as main memory or caches. An access by processor i on a variable X is atomic if no other processor is allowed to access any copy of X while the access by processor i is in progress. It has been shown that memory accesses need not be atomic at the hardware level for correct execution of concurrent programs.[6,7] Correctness of execution depends on the expected behavior of the machine. Two major classes of logical machine behavior have been identified because they are common in existing multiprocessor systems: the *strongly ordered* and the *weakly ordered* models of behavior.[7] The hardware of the machine must enforce these models by proper ordering of storage accesses and execution of synchronization and communication primitives. This leads to the third issue, the *ordering of events*.

The strictest logical model for the ordering of events is called *sequential consistency*, defined by Lamport. In a multiprocessor *sequential consistency* refers to the allowable sequence of execution of instructions within the same process and among different concurrent processes. Lamport defines the term more rigorously: "[A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."[8]

Since the only way that two concurrent processors can affect each other's execution is through the sharing of writable data and the sending of interrupt signals, it is the order of these events that really matters. In systems that are sequentially consistent we say that events are strongly ordered.

However, if we look at many systems (transaction systems, for example), it becomes clear that sequential consistency is often violated in favor of a weaker condition. In many machines it is often implicitly assumed that the programmer should make no assumption about the order in which the events that a process generates are observed by other processes between two explicit synchronization points. Accesses to shared writable data should be executed in a mutually exclusive manner, controlled by synchronizing variables. Accesses to synchronizing variables can be detected by the machine hardware at execution time. Strong ordering of accesses to these synchronizing variables and restoration of coherence at synchronization points are therefore the only restrictions that must be upheld. In such systems we say that events are weakly ordered. Weak ordering may result in more efficient systems, but the implementation problems remain the same as for strong ordering: strong ordering must still be enforced for synchronizing variables (rather than for all shared writable data).

We can infer from this discussion that synchronization, coherence, and ordering of events are closely related issues in the design of multiprocessors.

```
        {Processor 1:}
        A:=0
        .
        .
        A:=1                           /* event S1(A) */
LAB1:   If (B=1) goto LAB1             /* event L1(B) */
        <critical section>
        A:=0

        {Processor 2:}
        B:=0
        .
        .
        B:=1                           /* event S2(B) */
LAB2:   If (A=1) goto LAB2             /* event L2(A) */
        <critical section>
        B:=0
```

Figure 2. Synchronization protocol using two shared variables, *A* and *B*.

Communication and synchronization

Communication and synchronization are two facets of the same basic problem: how to design concurrent software that is correct and reliable, especially when the processes interact by exchanging control and data information. Multiprocessor systems usually include various mechanisms to deal with the various granules of synchronizable resources. Usually, low-level and simple primitives are implemented directly by the hardware. These primitives are the basic mechanisms that enforce mutual exclusion for more complex mechanisms implemented in microcode or software.

Hardware-level synchronization mechanisms. All multiprocessors include hardware mechanisms to enforce atomic operations. The most primitive memory operations in a machine are Loads and Stores. With atomic Loads and Stores complex synchronization protocols can be built. Figure 2 depicts a simple protocol. Before a processor can enter its critical section, it sets its control variable (*A* for processor 1 and *B* for processor 2) to 1. Hence, for both processors to be in their critical sections concurrently, both *A* and *B* must equal 1. But this is not possible, since a processor cannot enter its critical section if the other processor's control variable equals 1. Therefore, the two processors cannot execute their respective critical sections concurrently. This simple protocol can be deadlocked, but the problem can be remedied.[8] Such protocols are hard to design, understand, and prove correct, and in many cases they are inefficient.

More sophisticated synchronization primitives are usually implemented in hardware. If the primitive is simple enough, the controller of the memory bank can execute the primitive at the memory in the same way it executes a Load or a Store, at the added cost of a more complex memory controller. This is typically the case for the Test&Set and the Full/Empty bit primitives described below. Interprocessor interrupts are also possible hardware mechanisms for synchronization and communication. To send a message to another process currently running on a different processor, a process can send an interrupt to that processor to notify the destination process.

A common set of synchronization primitives consists of Test&Set(lock) and Reset(lock). The semantics of Test&Set and Reset are

```
TEST&SET(lock)
   { temp ← lock; lock ← 1;
     return temp; }
RESET(lock)
   { lock ← 0; }
```

The microcode or software will usually repeat the Test&Set until the returned value is 0. Synchronization at this level implies some form of busy waiting, which ties up a processor in an idle loop and increases the memory bus traffic and contention. The type of lock that relies on busy waiting is called a *spin-lock*.

To avoid spinning, interprocessor interrupts are used. A lock that relies on interrupts instead of spinning is called a *suspend-lock* (also called *sleep-lock* in the C.mmp[1]). This lock is similar to the spin-lock in the sense that a process does not relinquish the processor while it is waiting on a suspend-lock. However, whenever it fails to obtain the lock, it records its status in one field of the lock and disables all interrupts except interprocessor interrupts. When a process frees the lock, it signals all waiting processors through an interprocessor interrupt. This mechanism prevents the excessive interconnection traffic caused by busy waiting but still consumes processor cycles. Spin-locks and suspend-locks can be based on primitives similar to Test&Set, such as Compare&Swap.

The Compare&Swap(r1,r2,w) primitive is a synchronization primitive in the IBM 370 architecture; r1 and r2 are two machine registers, and w points to a memory location. The success of the Compare&Swap is indicated by the flag z. The semantics of the Compare&Swap instruction are

```
COMPARE&SWAP(r1,r2,w)
  { temp ← w; if (temp = r1)
    then {w ← r2; z ← 1;}
    else {r1 ← temp; z ← 0;}
  }
```

Test&Set and Compare&Swap are also called *read-modify-write* (RMW) primitives. A common performance problem associated with these basic synchronization primitives is the complexity of locking protocols. If N processes attempt to access a critical section at the same time, the memory system must execute N basic lock operations, one after the other, even if at most one process is successful. The NYU Ultracomputer[2] and the RP3 multiprocessor[9] use the Fetch&Add(x,a) primitive, where x is a shared-memory word and a is an increment. When a single processor executes the Fetch&Add on x, the semantics are

```
FETCH&ADD(x,a)
  { temp ← x; x ← temp + a;
    return temp; }
```

The implementation of the Fetch&Add primitive on the Ultracomputer is such that the complexity of an N-way synchronization on the same memory word is independent of N. The execution of this primitive is distributed in the interconnection network between the processors and the memory module. If N processes attempt to Fetch&Add the same memory word simultaneously, the memory is updated only once, by adding the sum of the N increments, and a unique value is returned to each of the N processes. The returned values correspond to an arbitrary serialization of the N requests. From the processor and memory point of view, the result is similar to a sequential execution of N Fetch&Adds, but it is performed in one operation. Consequently, the Fetch&Add primitive is extremely effective in accessing sequentially allocated queue structures and in the forking of processes with identical code that operate on different data segments. For example, the following high-level parallel Fortran statement[10] can be executed in parallel by P processors if there is no dependency between iterations of the loop:

```
DOALL N = 1 to 100
  <code using N>
ENDDO
```

Each processor executes a Fetch&Add on N before working on a specific iteration of the loop. Each processor will return a unique value of N, which can be used in the code segment. The code for each processor is as follows (N is initially loaded with the value 1):

```
n ← FETCH&ADD (N,1)
while (n ≤ 100) do
  { <code using N>
    n ← FETCH&ADD (N,1);
  }
```

In the HEP (Heterogeneous Element Processor) system, shared-memory words are tagged as *empty* or *full*. Loads of such words succeed only after the word is updated and tagged as full. After a successful Load, the tag can be reset to empty. Similarly, the Store on a full memory word can be prevented until the word has been read and the tag cleared. These mechanisms can be used to synchronize processes, since a process can be made to wait on an empty memory word until some other process fills it. This system also relies on busy waiting, and memory cycles are wasted on each trial. Each processor in the HEP is a multistream pipeline, and several process contexts are present in each processor at any time. A different process can immediately be activated when an attempt to synchronize fails. Very few processor cycles are wasted on synchronization. However, the burden of managing the tags is left to the programmer or the compiler. A more complex tagging scheme is advocated for the Cedar machine.[3]

Software-level synchronization mechanisms. Two approaches to synchronization are popular in multiprocessor operating systems: semaphores and message passing. We will discuss message passing in the next section. Operations on semaphores are P and V. A binary semaphore has the values 0 or 1, which signal acquisition and blocking, respectively. A counting semaphore can take any integer value greater than or equal to 0. The semantics of the P and V operations are

```
P(s)
  { if (s > 0) then
    s ← (s − 1);
        else
    { Block the process and append it
      to the waiting list for s;
      Resume the highest priority pro-
      cess in the READY LIST;}
  }

V(s)
  { if (waiting list for s empty) then
    s ← (s + 1);
        else
    { Remove the highest priority pro-
      cess blocked for s;
      Append it to the READY LIST;}
  }
```

In these two algorithms shared lists are consulted and modified (namely, the Ready List* and the waiting list for s). These accesses as well as the test and modification of s have to be protected by spin-locks, suspend-locks, or Fetch&Adds associated with semaphore s and with the lists. In practice, P and V are processor instructions or microcoded routines, or they are operating system calls to the process manager. The process manager is the part of the system kernel controlling process creation, activation, and deletion, as well as management of the locks. Because the process manager can be called from different processors at the same time, its associated data structures must be protected. Semaphores are particularly well adapted for synchronization. Unlike spin-locks and suspend-locks, semaphores are not wasteful of processor cycles while a process is waiting, but their invocations require more overhead. Note that locks are still necessary to implement semaphores.

Another synchronization primitive implemented in software or microcode is Barrier, used to "join" a number of parallel processes. All processes synchronizing at a barrier must reach the barrier before any one of them can continue. Barriers can be defined as follows after the task counter Count has been initialized to zero:

```
BARRIER(N)
  { count := count + 1;
    if (count ≥ N) then
    { Resume all processes on barrier
      queue;
```

*The Ready List is a data structure containing the descriptors of processes that are runable.

Table 1. **Synchronization, communication, and coherence in various multiprocessors.**

Multiprocessor	Number of processors	CPU architecture	Hardware primitives	Cache	Coherence scheme
IBM 3081	≤4	IBM 370	Compare&Swap (CS, CDS), Test&Set (TS)	Write-back	Central table
Synapse N+1*	≤32	Motorola 68000	Compare&Swap (CAS), Test&Set (TAS)	Write-back	Distributed table/ bus watching
Denelcor HEP*	100s	Custom	Full/empty bit	No cache	
IBM RP3†	100s	IBM ROMP	Fetch&Op (e.g., Fetch&Add)	Write-back	No shared writable data in cache
NYU Ultracomputer†	100s		Fetch&Add	Write-back	No shared writable data in cache
Encore Multimax	≤20	National Semiconductor 32032	Test&Set ("interlocked" instructions)	Write-through (two processors share each cache)	Bus watching
Sequent Balance 8000	≤12	National Semiconductor 32032	Test&Set (spin-lock using lock cache and bus watching)	Write-through	Bus watching

*Commercial machines no longer in production.
†Experimental prototype.

```
        Reset count; }
          else
        Block task and place in barrier
        queue;
    }
```

The first N-1 tasks to execute Barrier would be blocked. Upon execution of Barrier by the Nth task, all N tasks are ready to resume. In the HEP each task that is blocked spin-locks on a Full/Empty bit. The Nth task that crosses the barrier writes into the tagged memory location and thereby wakes up all the blocked tasks. This technique is very efficient for executing parallel, iterative algorithms common in numerical applications.

Interprocess communication. In a shared-memory multiprocessor, interprocess communication can be as simple as one processor writing to a particular memory location and another processor reading that memory location. However, since these activities occur asynchronously, communication is in most cases implemented by synchronization mechanisms. The reading process must be informed at what time the message to be read is valid, and the writing process must know at what time it is allowed to write to a particular memory location without destroying a message yet to be read by another process. Therefore, communication is often implemented by mutually exclusive accesses to mailboxes. Mailboxes are configured and maintained in shared memory by software or microcode.

Message-based communication can be synchronous or asynchronous. In a synchronous system the sender transmits a message to a receiving process and waits until the receiving process responds with an acknowledgment that the message has been received. Symmetrically, the receiver waits for a message and then sends an acknowledgment. The sender resumes execution only when it is confirmed that the message has been received. In asynchronous systems the sending process does not wait for the receiving process to receive the message. If the receiver is not ready to receive the message at its time of arrival, the message may be buffered or simply lost. Buffering can be provided in hardware or, more appropriately, in mailboxes in shared memory.

A summary of synchronization and communication primitives for different processors is given in Table 1.

Coherence in multiprocessors

Coherence problems exist at various levels of multiprocessors. Inconsistencies (i.e., contradictory information) can occur between adjacent levels or within the same level of a memory hierarchy. For example, in a cache-based system with write-back caches, cache and main memory may contain inconsistent copies of data.[4] Multiple caches conceivably could possess different copies of the same memory block because one of the processors has modified its copy. Generally, this condition is not allowable.

In some cases data inconsistencies do not affect the correct execution of a program (for example, inconsistencies between memory and write-back caches may be tolerated). In the following paragraphs we identify the cases for which data

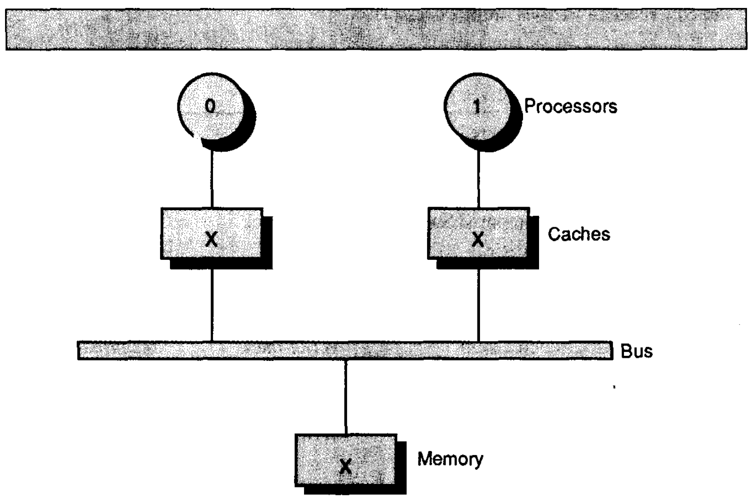


Figure 3. Cache configuration after a Load on X by processors 0 and 1. Copies in both caches are consistent.

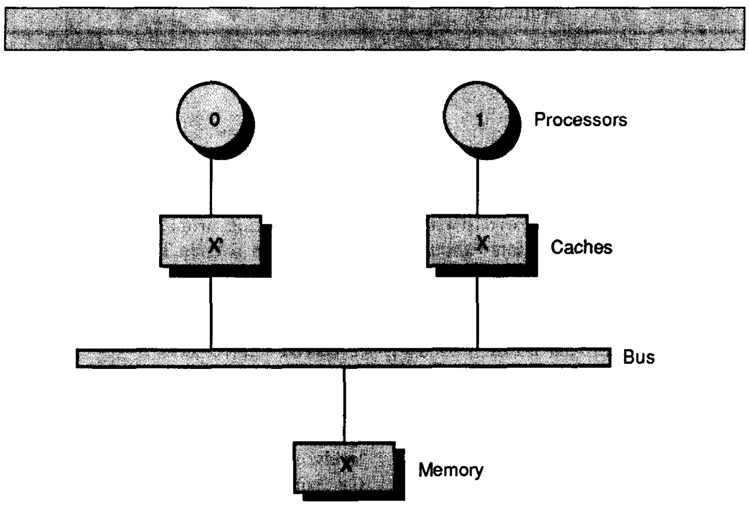


Figure 4. Cache configuration after a Store on X by processor 0 (write-through cache). The copies are inconsistent.

inconsistencies pose a problem and discuss various solutions.

Conditions for coherence. Data coherence problems do not exist in multiprocessors that maintain only a single copy of data. For example, consider a shared-memory multiprocessor in which each CPU does not have a private memory or cache (Figure 1 without optional caches). If Loads, Stores, and RMW cycles are atomic, then data elements are accessed and modified in indivisible operations. Each access to an element applies to the latest copy. Simultaneous accesses to the same element of data are serialized by the hardware.

Cache coherence problems exist in multiprocessors with private caches (Figure 1 with optional caches) and are caused by three factors: sharing of writable data, process migration, and I/O activity. To illustrate the effects of these three factors, we use a two-processor architecture with private caches (Figures 3-5). We assume that an element X is referenced by the CPUs. Let $L_j(X)$ and $S_j(X)$ denote a Load and a Store by processor j for element X in memory, respectively. If the caches do not contain copies of X initially, a Load of X by the two CPUs results in consistent copies of X, as shown in Figure 3. Next, if one of the processors performs a Store to X, then the copies of X in the caches become inconsistent. A Load by the other processor will not return the latest value. Depending on the memory update policy used in the cache, the cache level may also be inconsistent with respect to main memory. A write-through policy maintains consistency between main memory and cache. However, a write-back policy does not maintain such consistency at the time of the Store; memory is updated eventually when the modified data in the cache are replaced or invalidated. Figures 4 and 5 depict the states of the caches and memory for write-through and write-back policies, respectively.

Consistency problems also occur because of the I/O configuration in a system with caches. In Figure 6 the I/O processor (IOP) is attached to the bus, as is most commonly done. If the current state of the system is reached by an $L_0(X)$ and $S_0(X)$ sequence, a modified copy of X in cache 0 and main memory will not have been updated in the case of write-back caches. A subsequent I/O Load of X by the IOP returns a "stale" value of X as contained in memory. To solve the consistency problem in this configuration, the I/O processor must participate in the cache coherence protocol on the bus. The configuration in Figure 7 shows the IOPs sharing the caches with the CPUs. In this case I/O consistency is maintained if cache-to-cache consistency is also maintained; an obvious disadvantage of this scheme is the likely increase of cache perturbations and poor locality of I/O data, which will result in high miss ratios.

Some systems allow processes to migrate—i.e., to be scheduled in different processors during their lifetime—in order to balance the work load among the processors. If this feature is used in conjunction with private caches, data inconsistencies can result. For example, process A, which runs on CPU_0, may alter data contained in its cache by executing $S_0(X)$ before it is suspended. If process A migrates to CPU_1 before memory has been updated with the most recent value of X, process A may subsequently Load the stale value of X contained in memory.

It is obvious that a mere write-through policy will not maintain consistency in the system, since the write does not automatically update the possible copies of the data contained in the other caches. In fact, write-through is neither necessary nor sufficient for coherence.

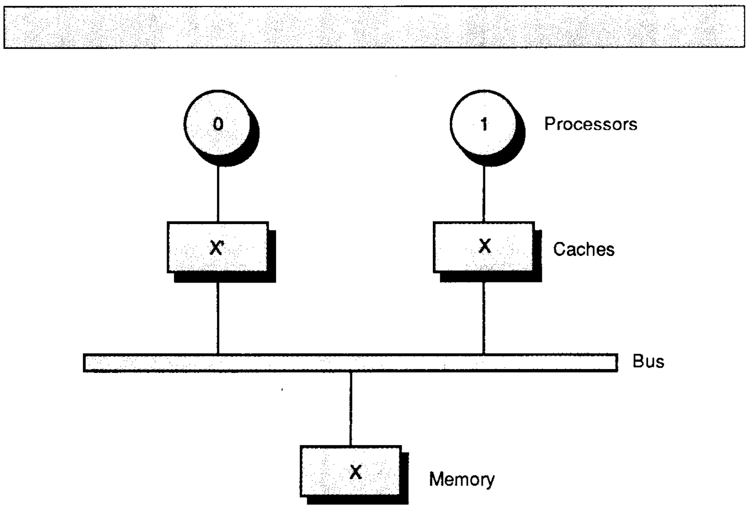


Figure 5. Same as Figure 4 but with write-back cache. The copies are inconsistent.

Solutions to the cache coherence problem. Approaches to maintaining coherence in multiprocessors range from simple architectural principles that make incoherence impossible to complex memory coherence schemes that maintain coherence "on the fly" only when necessary. Here we list these approaches from least to most complex:

(1) A simple architectural technique is to disallow private caches and have only shared caches that are associated with the main memory modules. Every data access is made to the shared cache. A network interconnects the processors to the shared cache modules.

(2) For performance considerations it is desirable to attach a private cache to each CPU. Data inconsistency can be prevented by not caching shared writable data; such data are called *noncachable*. Examples of shared writable data are locks, shared data structures such as process queues, and any other data protected by critical sections. Instructions and other data can be copied into caches as usual. Such items are referred to as *cachable*. The compiler must tag data as either cachable or noncachable. The hardware must adhere to the meaning of the tags. This technique, apparently

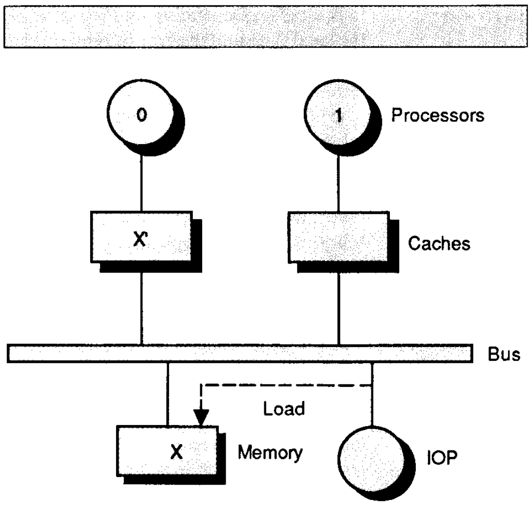


Figure 6. IOPs are attached to the bus and bypass the cache.

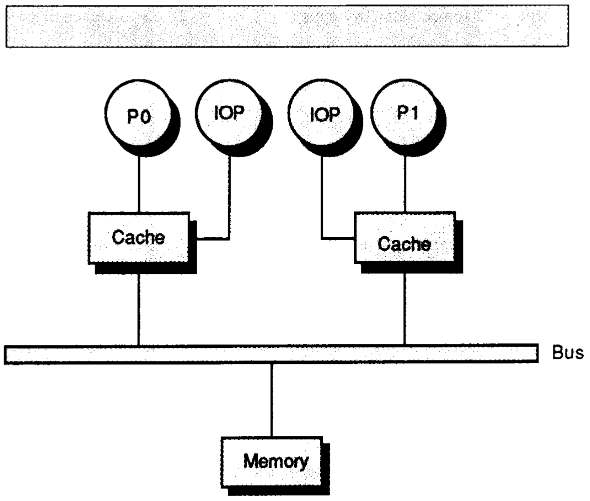
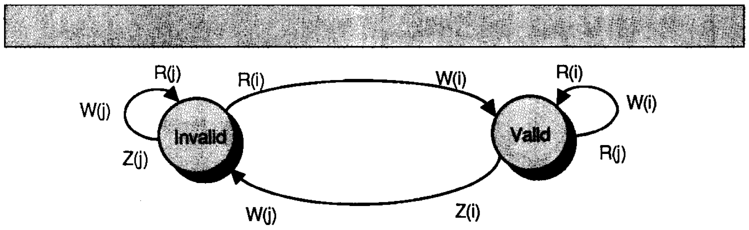


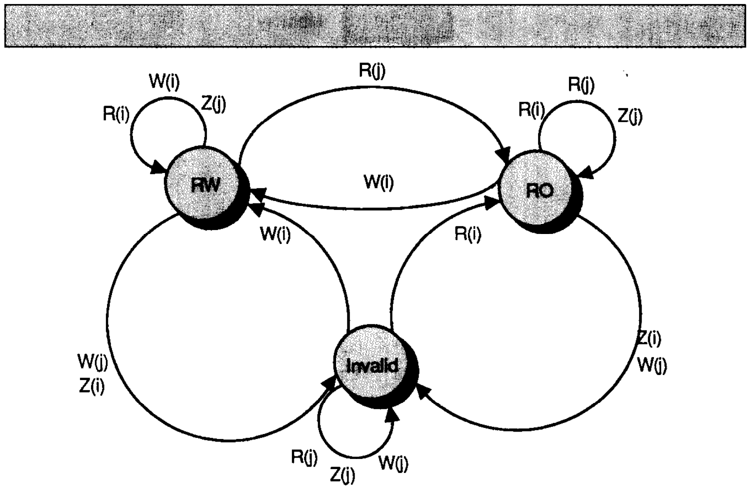
Figure 7. IOPs are attached to the caches.



W(i) = Write to block by processor i.
R(i) = Read block by processor i.
Z(i) = Displace block by cache i.

W(j) = Write to block by processor j ($j \neq i$).
R(j) = Read block by processor j ($j \neq i$).
Z(j) = Displace block by cache j ($j \neq i$).

Figure 8. State diagram for a given block in cache *i* for a write-through coherence protocol.



W(i) = Write to block by processor i.
R(i) = Read block by processor i.
Z(i) = Displace block by cache i.

W(j) = Write to block by processor j ($j \neq i$).
R(j) = Read block by processor j ($j \neq i$).
Z(j) = Displace block by cache j ($j \neq i$).

Figure 9. State diagram for a given block in cache *i* for a write-back coherence protocol.

simple in principle, must rely on the detection within each CPU that a block is cachable or not. Such a detection can be made in a virtual memory environment by tagging each page. The tag is stored in entries in the CPU's translation buffers. Translation buffers (TBs) are similar to caches, but they store virtual-to-physical address translations.

(3) If all shared writable data are declared noncachable, the performance may be degraded appreciably. If accesses to shared writable data always occur in critical sections, then such data can be cached. Only the locks that protect the critical sections must remain noncachable. However, to maintain data consistency, all data modified in the critical section must be invalidated in the cache when the critical section is exited. This operation is often referred to as a *cache flush*. The flushing operation ensures that no stale data remain in the cache at the next access to the critical section. If another cache accesses the data via the acquisition of the lock, consistency is maintained. This scheme is adequate for transaction-processing systems in which a shared record is acquired, updated in a critical section, and subsequently released. It works for write-through caches; for write-back caches, the design is more complex.

(4) A scheme allowing shared writable data to exist in multiple caches employs a centralized global table[5] and is used in many mainframe multiprocessor systems, such as the IBM 308x. The table stores the status of memory blocks so that coherence enforcement signals, called *cache cross-interrogates* (XI), can be generated on the basis of the block status. To maintain consistency, XI signals with the associated block address are propagated to the other caches either to invalidate or to change the state of the copies of the referenced block. An arbitrary number of caches can contain a copy of a block, provided that all the copies are identical. We refer to such a copy as a *read-only copy* (RO). To modify a block present in its cache, the processor must own the block with read and write access. When a block is copied from memory into cache, the block is tagged as exclusive (EX) if the cache is the only cache that has a copy of the block. A block is owned exclusively with read and write (RW) access when it has been modified. Only one processor can own an RW copy of a block at any time. The state IN (invalid) signals that the block has been invalidated.

The centralized table is usually located in the storage control element, which may also incorporate a crossbar switch that connects the CPUs to the main memory. To limit the accesses to the global table, local status flags can be provided in the cache directories for the blocks that reside in the cache. Depending on the status of the local flags and the type of request, the processor is allowed to proceed or is required to consult the global table.

(5) In bus-oriented multiprocessors the table that records the status of each block can be efficiently distributed among processors. The distributed-table scheme takes advantage of the broadcasting capability of the bus. Typically, consistency between the caches is maintained by a bus-watching mechanism, often called a *snoopy cache controller*, which implements a cache coherence protocol on the bus. In a simple scheme for write-through caches, all the snoopy controllers watch the bus for Stores. If a Store is made to a location cached in remote caches, then the copies of the block in remote caches are either invalidated or updated. This scheme also maintains coherence with I/O activity. Figure 8 depicts a state diagram of the block state changes depending on

the access type and the previous state of the block. A similar scheme was applied in the Sequent Balance 8000 multiprocessor, which can be configured with up to 12 processors.

The efficiency of the hardware that maintains coherence on the fly is vital. Recognizing that the Store traffic may contribute to bus congestion in a write-through system, Goodman proposed a scheme called *write-once*, in which the initial Store to a block copy in the cache also updates memory.[11] This Store also invalidates matching entries in remote caches, thereby ensuring that the writing processor has the only cached copy. Furthermore, Stores can be performed in the cache at the cache speed. Subsequent updates of the modified block are made in the cache only. A CPU or IOP Load is serviced by the unit (a cache or the memory) that has the latest copy of the block.

Multiprocessors with write-back caches rely on an ownership protocol. When the memory owns a block, caches can contain only RO copies of the block. Before a block is modified, ownership for exclusive access must first be obtained by a read-private bus transaction, which is broadcast to the caches and memory. If a modified block copy exists in another cache, memory must first be updated, the copy invalidated, and ownership transferred to the requesting cache. Figure 9 diagrams memory block state transitions brought about by processor actions. The first commercial multiprocessor with write-back caches was the Synapse N + 1.

Variants of the cache coherence bus protocols have been proposed. One scheme, proposed for the Spur project at the University of California, Berkeley, combines compile-time tagging of shared and private data and the ownership protocol. In another system, the Xerox Dragon multiprocessor, a write is always broadcast to other caches and main memory is updated only on replacement. These bus protocols are described and their performances compared in an article by Archibald and Baer.[12]

Advantages and disadvantages. Although scheme 1 provides coherence while being transparent to the user and the operating system, it does not reduce memory conflicts but only the memory access latency. Shared caches, by necessity, contradict the rule that processors and caches should be as close together as possible. I/O accesses must be serviced via the shared caches to maintain coherence.

Tagging shared writable data fails to alleviate the coherence problem caused by I/O accesses.

There are a number of disadvantages associated with scheme 2, which tags data as cachable or noncachable. The major one is the nontransparency of the multiprocessor architecture to the user or the compiler. The user must declare data elements as shared or nonshared if a concurrent language such as Ada, Modula-2, or Concurrent Pascal is used.[13] Alternatively, a multiprocessing compiler, such as Parafrase,[10] can classify data as shared or nonshared automatically. The efficiency of these approaches depends respectively on the ability of the language to specify data structures (or parts thereof) that are shared and writable and of the compiler to detect the subset of shared writable data. Since in practical implementations a whole page must be declared as cachable or not, internal fragmentation may result, or more data than the shared writable data may become noncachable.

Tagging shared writable data also fails to alleviate the coherence problem caused by I/O accesses. Either caches must be flushed before I/O is allowed to proceed, or all data subject to I/O must be tagged as noncachable as well. Depending on the frequency of I/O operations, both approaches reduce the overall hit rate of the caches and hence the speedup obtained by using caches.

Another common drawback of tagging shared writable data rather than maintaining coherence on the fly is the inefficiency caused by process migration. Caches must be flushed before each migration or process migration must be disallowed at the cost of limiting scheduling flexibility.

Scheme 3—flushing caches only when synchronization variables are accessed—has performance problems. In practice the whole cache has to be flushed, or else the data accessed in a critical section must be tagged in the cache. I/O must also be preceded by cache flushing. Note that the programmer must be aware that coherence is restored only at synchronization points. Scheme 3 appears to be attractive only for small caches.

Scheme 4 solves the problems caused by I/O accesses and process migration. However, a global table that must be accessed by all cache controllers can become a bottleneck, even when XIs are filtered by hardware. But the main problem of this coherence scheme is the distance between the processors and the global table. As processors become faster, the access latency of the table becomes a limiting factor of system performance; in particular, when cache access times are very fast, the time penalty for a miss (*miss penalty*) must be minimized.

By distributing the table among the caches, the last scheme partly solves the problems of table access contention and latency. However, the complexity of the bus interface unit is increased because it has to "watch" the bus. Furthermore, since the scheme relies on a broadcast bus, the number of processors that can be interconnected is limited by the bus bandwidth.

Ping-pong effect. In systems with caches employing scheme 4 or 5, the execution of synchronization primitives, such as atomic read-modify-write memory cycles, can create additional access penalties. If two or more processors are spinning on a lock, RMW cycles that cause the lock variable to bounce repeatedly from one cache to another are generated. This can be aggravated by clustering different locks into a given block of memory. However, if RMW operations are implemented carefully, spin-locks can be efficient.

Let us illustrate the ping-pong problem by an example and discuss techniques for reducing system performance degradations. In this example we will assume the use of the Test&Set(lock) instruction; however, the problem can occur with other primitives. The traditional segment of code executed to acquire access to a critical section via a spin-lock is the following:

```
while (TEST&SET(lock) = 1) do nothing;
  /* spin-lock with RMW cycles */
     < execute critical section >
RESET(lock);
  /* exit critical section */
```

Assume that each processor has a private write-back cache and that three or more processors attempt to access the critical section concurrently. If processor P_0 succeeds in acquiring the lock, the other processors (P_1 and P_2) will spin-lock and cause the modified lock variable to be invalidated in the other processors' caches for each access to the lock. As a result of the invalidation of the modified lock variable, the block is transferred to the requesting cache—a significant penalty. The modification is a result of the writing in the last part of the RMW memory cycle.

One technique for avoiding the ping-pong effect is to use the following segment of code in place of the while statement in the previous code segment:

```
repeat
   while (LOAD(lock) = 1) do nothing;
      /* spin without modification */
until (TEST&SET(lock) = 0);
```

In this segment of code the lock is first loaded to test its status. If available, a Test&Set is used to attempt acquisition. However, while a processor is attempting to acquire the lock, it "spins" locally in its cache, repeating the execution of a tight loop made of a Load followed by a Test. This spinning causes no invalidation traffic on the bus. On a subsequent release of the lock, the processors contend for the lock, and only one of them will succeed. The ping-pong problem is solved; spinlocks can therefore be implemented efficiently in cache-based systems.

Ping-ponging also occurs for shared writable variables. A typical example is the index N in the Doall loop described earlier, in the section on hardware-level synchronization mechanisms. Unless the implementation of Fetch&Add is carefully designed, accesses to the index N create a "hot spot,"[9] which in a cache-based system results in intense ping-ponging between the caches. The careful implementation of synchronization primitives and the creation of hot spots in cache-based systems are research topics that deserve more attention.

Strong and weak ordering of events

The mapping of an algorithm as conceived and understood by a human programmer into a list of machine instructions that correctly implement that algorithm is a complex process. Once the translation has been accomplished, however, it is relatively easy in the case of a uniprocessor to understand what modifications of the machine code can be made without altering the outcome of the execution. A compiler, for example, can resequence instructions to boost performance, or the processor itself can execute instructions out of order if it is pipelined. This is allowable in uniprocessors, provided that hardware mechanisms (interlocks) exist to check data and control dependencies between instructions to be executed concurrently or out of program order.

Local dependency checking is necessary, but it may not preserve the intended outcome of a concurrent execution.

If a processor is a part of a multiprocessor that executes a concurrent program, then such local dependency checking is still necessary but may not be sufficient to preserve the intended outcome of a concurrent execution. Maintaining correctness and predictability of the execution of concurrent programs is more complex for three reasons:

(1) The order in which instructions belonging to different instruction streams are executed is not fixed in a concurrent program. If no synchronization among instruction streams exists, then a very large number of different instruction interleavings is possible.

(2) If for performance reasons the order of execution of instructions belonging to the same instruction stream is different from the order implied by the program, then an even larger number of instruction interleavings is possible.

(3) If accesses are not atomic (for example, if multiple copies of the same data exist), as is the case in a cache-based system, and if not all copies are updated at the same time, then different processors can individually observe different interleavings during the same execution. In this case the total number of possible execution instantiations of a program becomes still larger.

To illustrate the possible types of interleavings, we examine the following three program segments to be executed concurrently by three processors (initially $A = B = C = 0$, and we assume that a Print statement reads both variables indivisibly during the same cycle):

P1	P2	P3
a: $A \leftarrow 1$	c: $B \leftarrow 1$	e: $C \leftarrow 1$
b: Print BC	d: Print AC	f: Print AB

If the outputs of the processors are concatenated in the order P1, P2, and P3, then the output forms a six-tuple. There are 64 possible output combinations. For example, if processors execute instructions in program order, then the execution interleaving a,b,c,d,e,f is possible and would yield the output 001011. Likewise, the interleaving a,c,e,b,d,f is possible and would yield the output 111111. If processors are allowed to execute instructions out of program order, assuming that no data dependencies exist among reordered instructions, then the interleaving b,d,f,e,a,c is possible and would yield the output 000000. Note that this outcome is not possible if processors execute instructions in program order only.

Of the 720 (6!) possible execution interleavings, 90 preserve the individual program order. We have already pointed out that of the 90 program-order interleavings not all six-tuple combinations can result (i.e., 000000 is not possible). The question remains whether out of the 720 non-program-order interleavings all six-tuple combinations can result. So far we have assumed that the memory system of the example multiprocessor is access atomic; this means that memory updates affect all processors at the same time. In a cache-based system such as depicted in Figure 1, this may not be the case; such a system can be nonatomic if an invalidation does not reach all caches at the same time.

In an atomic system it is easy to show that, indeed, not all six-tuple combinations are possible, even if processors need not adhere to program order. For example, the outcome 011001 implies the following: Processor P1 observes that C has been updated and B has not been updated yet. This implies that P3 must have executed statement e before P2 executed statement c. Processor P2 observes that A has been updated before C has been updated. This implies that P1 must have executed statement a before P3 executed statement e. Processor P3 observes that B has been updated but A has not been updated. This implies that P2 must have executed statement c before P1 executed statement a. Hence, e occurred before c, a occurred before e, and c occurred before a. Since this ordering is plainly impossible, we can conclude that in an atomic system, the outcome 011001 cannot occur.

The above conclusion does not hold true

in a nonatomic multiprocessor. Let us assume that the actual execution interleaving of instructions is *a,c,e,b,d,f*. Let us further assume the following sequence of events: When P1 executes *b*, P1's own copy of *B* has not been updated, but P1's own copy of *C* has been updated. Hence, P1 prints the tuple 01. When P2 executes *d*, P2's own copy of *A* has been updated, but P2's own copy of *C* has not been updated. Hence, P2 prints the tuple 10. When P3 executes *f*, P3's own copy of *A* has not been updated, but P3's own copy of *B* has been updated. Hence, P3 prints the tuple 01. The resulting six-tuple is indeed 011001. Note that all instructions were *executed* in program order, but other processors did not *observe* them in program order.

We might ask ourselves whether a multiprocessor functions incorrectly if it is capable of generating any or all of the above-mentioned six-tuple outputs. This question does not have a definitive answer; rather the answer depends on the expectations of the programmer. A programmer who expects a system to behave in a sequentially consistent manner will perceive the system to behave incorrectly if it allows its processors to execute accesses out of program order. The programmer will likely find that synchronization protocols using shared variables will not function. The difficulty of concurrent programming and parallel architectures stems from the effort to disallow all interleavings that will result in incorrect outcomes while not being overly restrictive.

Processor 1	Processor 2	Processor 3
.	.	.
		.
$S_1(A)$	$L_2(A)$	$L_3(B)$
.	$S_2(B)$	$L_3(A)$
.	.	.
.	.	.

Figure 10. Concurrent program for three processors accessing shared variables.

Systems with atomic accesses. We have shown in an earlier work[14] that a necessary and sufficient condition for a system with atomic memory accesses to be sequentially consistent (the strongest condition for logical behavior) is that memory accesses must be performed in the order intended by the programmer—i.e., in program order. (A Load is considered performed at a point in time when the issuing of a Store from any processor to the same address cannot affect the value returned by the Load. Similarly, a Store on X by processor i is considered performed at a point in time when an issued Load from any processor to the same address cannot return a value of X preceding the Store.) In a system where such a condition holds, we say that storage accesses are *strongly ordered*.

In a system without caches a memory access is performed when it reaches the memory system or at any point in time when the order of all preceding accesses to memory has become fixed. For example, if accesses are queued in a FIFO (first in, first out) buffer at the memory, then an access is performed once it is latched in the buffer. When a private cache is added to each processor, Stores can also be atomic in the case of a bus system because of the simultaneous broadcast capability of the buses; in such systems the invalidations generated by a Store and the Load requests broadcast by a processor are latched simultaneously by all the controllers (including possibly the memory controllers). As soon as each controller has taken the proper action on the invalidation, the access can be considered performed.

Buffering of access requests and invalidations also become possible if the rules governing sequential consistency are carefully observed. With extensive buffering at all levels, and provided that the interconnection and the memory system have sufficient bandwidth, the efficiency of all processors may be very high, even if the memory access latency is large compared to the processor cycle time. Two articles present more detailed discussions of the buffering of accesses and invalidations in cache-based multiprocessors.[7,15]

In a weakly ordered system the condition of strong ordering is relaxed to include accesses to synchronization variables only. Synchronization variables must be hardware-recognizable to enforce the specific conditions of strong ordering on them. Moreover, before a lock access can proceed, all previous accesses to nonsynchronization data must be allowed to "settle." This means that all shared memory accesses made before the lock operation was encountered must be completed before the lock operation can proceed. In systems that synchronize very infrequently, the relaxation of strong ordering to weak ordering of data accesses can result in greater efficiency. For example, if the interconnection network is buffered and packet-switched, the interface between the processor and the network can send global memory requests only one at a time to the memory if strong ordering is to be enforced. The reason for this is that in such a network the access time is variable and unpredictable because of conflicts; in many cases waiting for an acknowledgment from the memory controller is the only way to ensure that global accesses are performed in program order. In the case of weak ordering the interface can send the next global access directly after the current global access has been latched in the first stage of the interconnection network, resulting in better processor efficiency. However, the frequency of lock operations will be higher in a program designed for a weakly ordered system.

Systems with nonatomic accesses. In a multiprocessor system with nonatomic accesses, it has been shown that the previous condition for strong ordering of storage accesses (and sequential consistency) is not sufficient.[14]

Example 1. In a system with a recombining network[2] the network can provide for access short-circuiting, which combines Loads and Stores to the same address within the network, before the Store reaches its destination memory module. For the parallel program in Figure 10—$S_i(X)$ and $L_i(X)$ represent global accesses "Store into location X by processor i" and "Load from location X by processor i," respectively—such short-circuiting can result in the following sequence of events:

(1) Processor 1 issues a command to store a value at memory location A.

(2) Processor 2 reads the value written by processor 1 "on the fly" before *A* is updated.

(3) Because of the successful read of *A* in step 2, processor 2 issues a command to write a value at memory location *B*.

(4) Processor 3 reads the value written by processor 2; it reflects the updated *B*.

(5) Processor 3 reads memory location *A* and an old value for *A* is returned because the write to *A* by processor 1 has not propagated to *A* yet.

Each processor performs instructions in the order specified by the programmer, but sequential consistency is violated. Processor 2 implies that step 1 has been completed by processor 1 when it initiates step 3. In step 4 processor 3 recognizes that implication by successfully reading *B*. But when processor 3 then reads *A*, it does not find the implied new value but rather the old value. Consequently, processor 3 observes an effect of step 1 before it is capable of observing step 1 itself.

Example 2. In a cache-based system where memory accesses and invalidations are propagated one by one through a packet-switched (but not recombining) network, the same problem as in the previous example may occur. Initially, all processors have an RO copy of *A* in their cache.

(1) Processor 1 issues a command to store a value at memory location *A*. Invalidations are sent to each processor with a copy of *A* in its cache. (For simplicity we assume that the size of a cache block is one word.)

(2) Processor 2 reads the value of *A* as updated by processor 1, because the invalidation has reached its cache; processor 1 writes the data back to main memory and forwards a copy to processor 2.

(3) Because of the successful read of *A* in step 2, processor 2 issues a command to write a value at memory location *B*, sending invalidations for copies of *B*.

(4) Processor 3 reads the value written by processor 2; it reflects the updated *B*, because the associated invalidations have propagated to processor 3.

(5) Processor 3 reads memory location *A* and an old value for *A* is returned because the invalidation for *A* caused by processor 1 has not yet propagated to the third processor's cache.

Again each processor executes all instructions in program order. Furthermore, a processor does not proceed to issue memory accesses before all previous invalidations broadcast by the processor have been acknowledged. Yet the same problem occurs as in the previous example; sequential consistency is violated. This is the case because invalidations are essentially memory accesses. Because invalidations are not atomic, the system is not strongly ordered.

User interface

The discussion in this article shows that the issues of synchronization, communication, coherence, and ordering of events in multiprocessors are intricately related and that design decisions must be based on the environment for which the machine is destined. Coherence depends on synchronization in some coherence protocols because the user has to be aware that synchronization points are the only points in time at which coherence is restored. Strict ordering of events may be enforced all the time (strong ordering) or at synchronization points only (weak ordering).

At the user level most features of the physical (hardware) architecture are not visible. The instruction set of each processor and the virtual memory are the most important system features visible to the programs. Depending on the features of the physical architecture that are visible to the programmer, the task of programming the machine may be more difficult, and it may be more difficult to share the machine among different users.

Nontransparent coherence or ordering schemes. A sophisticated compiler may succeed in efficiently detecting and tagging the shared writable data to avoid the coherence problem. Such a compiler may also be able to make efficient use of synchronization primitives provided at different levels. The compiler may be aware of access ordering on a specific machine and generate code accordingly. It is not clear that compiler technology will improve to a point where efficient code can be generated for these different options.

If a program is written in a high-level concurrent language, the facility to specify shared writable data may not exist in the language, in which case we must still rely on the compiler for detecting the minimum set of data to tag as noncachable. It should be emphasized that perfectly legal programs in concurrent languages that allow the sharing of data, generally will not execute correctly in a system where events are weakly ordered.

User access to synchronization primitives. Programmers of concurrent applications may have in their repertoires different hardware- or software-controlled synchronization primitives. For performance reasons it may be advisable to let basic hardware-level synchronization instructions be directly accessible to users, who know their applications and can tailor the synchronization algorithm to their own needs. The basic drawback of such a policy is the increased possibility of deadlocks, resulting from programming errors or processor failure. Spin-locks and suspend-locks consume processor cycles and bus cycles. Therefore, such locks should never be held for a long time. Ideally, a processor should not be interruptible during the time that it owns a lock; for example, one or several processors may spin forever on a lock if the process that "owns" the lock has to be aborted because of an exception. In a virtual-machine environment the user process does not have any control over the interruptibility of the processor, and thus a process can be preempted while it is owning a lock. This will result in unnecessary, resource-consuming spinning from all other processes attempting to obtain the lock.

A solution to this problem is the task-force scheduling strategy,[1] in which all active processes of a multitask are always scheduled and preempted together. Another solution is the implementation of some kind of time-out on spinning. The drastic solution to all these problems is to involve the operating system in every synchronization or communication, so that it can include these mechanisms in its scheduling policy to maximize performance.

Making a multiprocessor function correctly can be a simple or an extremely difficult task. Basic synchronization mechanisms can be primitive or complex, wasteful of processor cycles or highly efficient. In any case the underlying hardware must support the basic assumptions of the logical model expected by the user. In a strongly ordered system such an assumption usually is that the system behaves in a sequentially consistent manner.

Increased transparency comes at the cost of efficiency and increased hardware complexity. But traditional and significant advantages such as the ability to protect users against themselves and other users, ease of programming, portability of programs, and efficient management of shared resources by multiple users are strong arguments for the designers of general-purpose computers to accept the hardware complexity and the negative effect on performance. The designers of general-purpose machines will probably prefer coherence enforcement on the fly in hardware, strong ordering of memory accesses, and restricted access to synchronization primitives by the user.

On the other hand, for machines with limited access by sophisticated users, such as supercomputers and experimental multiprocessor systems, the performance of each individual task may be of prime importance, and the increased cost of transparency may not be justified.

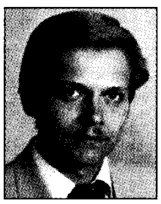
The challenge of the future lies in the ability to control interprocess communication and synchronization in systems without rigid structures. Efficient multiprocessing will be provided by systems in which synchronization, coherence, and logical ordering of events are carefully analyzed and blended together harmoniously in the context of efficient hardware implementations. It is necessary, however, to provide the programmer with a simple logical model of concurrency behavior. When multiprocessors do not conform to the concept of a single logical model, but rather must be viewed as a dynamic pool of processing, storage, and connection resources, the control in software over communication and synchronization becomes a truly formidable task. The concepts of strong and weak ordering as defined in this article correspond to two widely accepted models of multiprocessor behavior, and we believe that future designs will conform to one of the two models.□

Acknowledgment

Through many technical discussions, William Collier of IBM Poughkeepsie helped shape the content of this article.

References

1. A.K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems—A Status Report," *Computing Surveys*, June 1980, pp. 121-165.
2. A. Gottlieb et al., "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Computers*, Feb. 1983, pp. 175-189.
3. D. Gajski and J.-K. Peir, "Essential Issues in Multiprocessor Systems," *Computer*, June 1985, pp. 9-27.
4. A.J. Smith, "Cache Memories," *Computing Surveys*, Sept. 1982, pp.473-530.
5. L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, Dec. 1978, pp.1112-1118.
6. W.W. Collier, "Architectures for Systems of Parallel Processes," IBM Technical Report TR 00.3253, Poughkeepsie, N.Y., Jan. 1984.
7. M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering In Multiprocessors," *Proc. 13th Int'l Symp. Computer Architecture*, June 1986, pp. 434-442.
8. L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Trans. Computers*, Sept. 1979, pp. 690-691.
9. G.F. Pfister et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 Parallel Processing Conf.*, pp. 764-771.
10. D.A. Padua, D.J. Kuck, and D.H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. Computers*, Sept. 1980, pp. 763-776.
11. J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Int'l Symp. Computer Architecture*, June 1983, Stockholm, Sweden, pp. 124-131.
12. J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Computer Systems*, Nov. 1986, pp. 273-298.
13. G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Mar. 1983, pp. 3-43.
14. M. Dubois and C. Scheurich, "Dependency and Hazard Resolution in Multiprocessors," Univ. of Southern Calif. Technical Report CRI 86-20.
15. C. Scheurich and M. Dubois, "Correct Memory Operation of Cache-Based Multiprocessors," *Proc. 14th Int'l Symp. Computer Architecture*, June 1987, pp. 234-243.

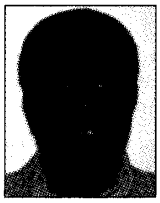


Michel Dubois has been an assistant professor in the Department of Electrical Engineering of the University of Southern California since 1984. Before that, he was a research engineer at the Central Research Laboratory of Thomson-CSF in Orsay, France. His main interests are computer architecture and parallel processing with an emphasis on high-performance multiprocessor systems. He has published more than 30 technical papers on multiprocessor architectures, performance, and algorithms, and he served on the program committee of the 1987 Architecture Symposium.

Dubois holds a PhD from Purdue University, an MS from the University of Minnesota, and an engineering degree from the Faculté Polytechnique de Mons in Belgium, all in electrical engineering. He is a member of the ACM and the Computer Society of the IEEE.

Christoph Scheurich is a doctoral student and research assistant in the Department of Electrical Engineering-Systems at USC. He received the BSEE in 1981 from the University of the Pacific, Stockton, California, and the MS degree in computer engineering in 1985 from USC. His current interests lie in computer architecture, specifically the design and implementation of multiprocessor memory systems.

Scheurich is a student member of the ACM and the Computer Society of the IEEE.



Fayé A. Briggs is in the Advanced Development Group at Sun Microsystems. He was an associate professor of electrical and computer engineering at Rice University, and prior to that he was on the faculty of Purdue University. He has also served as a consultant to IBM, TI, and Sun. His current research interests are multiprocessor and vector architectures, their compilers, operating systems, and performance. He has published more than 35 technical papers in these areas and is the coauthor of *Computer Architecture and Parallel Processing* (McGraw-Hill).

Briggs has a PhD from the University of Illinois and an MS from Stanford University, both in electrical engineering.

Readers may write to the authors c/o Michel Dubois, Dept. of Electrical Engineering, University of Southern California, University Park, Los Angeles, CA 90089-0781.