Memory Access Dependencies in Shared-Memory Multiprocessors

MICHEL DUBOIS, MEMBER, IEEE, AND CHRISTOPH SCHEURICH, MEMBER, IEEE

Abstract—**A multiprocessor system designed to support multithreading must adhere to a simple logical model of concurrency. Besides executing each process correctly, the multiprocessor must preserve the dependencies among processes. Dependencies among concurrent processes are specified by explicit statements such as critical sections or by the sharing of writable data. Parallelizing compilers and programmers using concurrent languages must conform to the model of concurrency of the machine, to generate correct code.**

The presence of high-performance mechanisms in shared-memory multiprocessors, such as private caches, extensive pipelining of memory accesses and combining networks may render a logical concurrency model complex to implement or inefficient. In this paper, the problem of implementing a given logical concurrency model in a multiprocessor is addressed. Two concurrency models are considered, and simple rules are introduced to verify that a multiprocessor architecture adheres to the models. The rules are applied to several examples of multiprocessor architectures.

Index Terms—**Cache-based systems, combining networks, memory coherence, ordering of events, parallel algorithms, sequential consistency, shared-memory multiprocessors.**

I. Introduction

SHARED memory MIMD systems are becoming very popular because of their versatility and the fact that they are a natural evolution from traditional architectures based on the proven von Neumann single-processor concept. Off-the-shelf, low-cost microprocessor components can be connected in a tightly-coupled configuration [1], and attain computing speed comparable to that of single-processor mainframes. Similarly, in high-end machines, the limit to technology improvements leads to the use of tightly-coupled configurations to meet the ever-increasing demand for computing power. This trend is clear from observing the design of recent machines such as the IBM3090 [2] and the Cray-XMP [3]. At the same time, many advanced designs for shared memory multiprocessor supercomputers are being researched both in academia and in industry: examples are the Cedar at the University of Illinois [4], the NYU Ultracomputer [5], and the IBM RP3 multiprocessor [6].

The emergence of very fast pipelined RISC processors in VLSI, MOS, ECL, or even Gallium Arsenide technologies, poses a challenge to the designers of multiprocessor systems. The major problem is to maintain good processor utilization even though the multiprocessor environment adds penalties to fetch instructions, access operands, and synchronize processes. Hardware solutions to the memory access problem include caching and prefetching. In a multiprocessor, the flexibility of such mechanisms is limited because the logical concurrency model must remain simple and well-defined.

In Sections II and III, we briefly overview the concurrency models supported by shared-memory multiprocessors as well as the possible hardware mechanisms to speed up multiprocessor execution. In Sections IV and V, respectively, the importance of processor behavior and memory behavior is discussed. Section VI combines processor and memory behavior models to define two useful concurrency models. In Section VII the enforcement of concurrency models in different architectures is discussed. A simple performance analysis is presented in Section VIII and conclusions are drawn in Section IX.

Manuscript received May 1, 1987; revised May 13, 1988 and August 16, 1989. Recommended by B. W. Wah. This work was supported by the National Science Foundation under Grants DMC-8505328 and CCR-8709997.

The authors are with the Department of Electrical Engineering—Systems, University of Southern California, Los Angeles, CA 90089.

IEEE Log Number 9034810.

II. Concurrency Models

Concurrently executing threads of a single program need to synchronize and to communicate data among themselves. Processes must synchronize in order to enforce mutual exclusion during the execution of operations on shared writable data (e.g., *critical section*) or to coordinate execution (e.g., barrier synchronization) [7]. The concurrency model of a multiprocessor defines the permissible ways to synchronize and communicate in the multiprocessor. The global ordering of shared storage accesses by the hardware directly affects the concurrency model of the architecture.

A. Time-Shared Uniprocessors and Sequential Consistency

Multiprocessing can be implemented in time-shared uniprocessors. In a time-shared uniprocessor, only one process runs at a time but "slices" of each process execution are executed in turn. In this case, communication and synchronization protocols among processes can be implemented with simple Loads and Stores of shared data. A trivial synchronization algorithm using only Loads and Stores is shown in Fig. 6(a) of Section IV for two processes. An extension of this algorithm to enforce mutual

```
      begin integer j;
L1:   choosing[1]:=1;
      number[i]:=1+max(number[1],...,number[N]);
      choosing[i]:=0;
      for j:=1 to N do
          begin
L2:       if choosing[j] ≠ 0 then goto L2;
L3:       if number[j] ≠ 0 and
          ((number[j],j) < (number[i],i)) then goto L3;
          end;
      <critical section>
      number[i]:=0;
      end;
```

Fig. 1. N-process mutual exclusion (bakery) algorithm [8], shown for process i. The relation $(a, b) < (c, d)$ is true if $a < c$, or if $a = c$ and $b < d$.

exclusion of N processes, using simple Loads and Stores was given by Lamport [8]; it is shown in Fig. 1. Lamport's "bakery" algorithm relies on processes "taking a number" and waiting for their turn to enter the critical section. If two processes have the same number, then the process with the lower process ID has priority. Similar algorithms had been derived previously by Dijkstra [9], Knuth [10], and others.

Fig. 2 shows a barrier synchronization algorithm for N processes operating concurrently on the *same* iteration of a loop. Before a process is allowed to start processing the next iteration, all processes must have finished the present iteration. One master process N controls the barrier, and $N - 1$ slave processes cooperate with the master process.

Fig. 3 shows a simple producer-consumer algorithm using a limited buffer of size b [11]. The producer may not overwrite data in the buffer or attempt to write to a full buffer. The consumer reads a data item only once, and may not attempt to read from an empty buffer.

Many other communication and synchronization algorithms relying on simple Loads and Stores on shared data could be designed by individual programmers. The concurrency model in which these algorithms are permissible is called *sequential consistency*, and is easily satisfied in a time-shared uniprocessor. In order for a multiprocessor to adhere to the model of sequential consistency, storage accesses must be *strongly ordered*, as we will show in Section VI.

B. Hardware Synchronization and Communication Primitives

In many multiprocessors, it is not permitted to write algorithms accessing shared writable data without explicit synchronization using built-in synchronization and communication primitives. The reason is that the machine does not enforce sequential consistency; as a result, valid programs designed for a sequentially consistent multiprocessor do not run correctly.

For example, with a simple *Test_and_set* primitive, the critical sections of Figs. 1 and 6(a) can be implemented by the code, shown in Fig. 4 [12]. This code is simpler than the code using Loads and Stores only. The program for the communication channel of Fig. 3 must be rewritten, in a system requiring explicit synchronization with

```
      MASTER PROCESS N:
      for i= 1 to k do
          begin
          do_iteration(N,i);
          for j = 1 to N-1 do
L1:         if flag[j] = 0 then goto L1;
          for j = 1 to N-1 do flag[j] := 0;
          end;
      SLAVE PROCESS p:
      for i = 1 to k do
          begin
          do_iteration(p,i);
          flag[p] := 1;
L2:       if flag[p] = 1 then goto L2;
          end;
```

Fig. 2. N-process barrier synchronization. All processes concurrently work on iteration i and may not proceed to iteration $i + 1$ until all processes are finished with iteration i. Initially all flags are set at 0. Process N synchronizes processes 1 through N-1.

```
      PRODUCER:
L1:   if s-r mod k = b then goto L1;
      <put message into buffer>
      s:= s+1 mod k;
      goto L1;
      CONSUMER:
L2:   if s-r mod k = 0;
      <take message from buffer>
      r:=r+1 mod k;
      goto L2;
```

Fig. 3. A producer consumer algorithm from Lamport [11].

```
L1:   If test&set(x) = 1 then goto L1;
      <critical section>
      reset(x);
```

Fig. 4. Mutual exclusion algorithm based on a Test&set primitive.

built-in primitives; a possible implementation is shown in Fig. 5. Other implementations are possible, and more sophisticated primitives than Test_and_set may exist in any given multiprocessor. However, the need for indivisible read-modify-write operations—such as Test_and_set—poses problems to the hardware. Either memory locations must be locked, and access to them is denied to all processors except to the one executing the read-modify-write operation, or the memory system itself must *execute* the entire read-modify-write operation indivisibly. When multiple copies of data exist in the system, such as in cache-based systems, the execution of read-modify-write instructions can become difficult and inefficient. On the other hand, when indivisible instructions are not used to implement synchronization algorithms, then the propagation of ordinary Load and Store accesses in the memory system is restricted. Depending on the underlying architecture of the multiprocessor, this restriction can be difficult to enforce. The complexities and efficiencies of the programs written with and without special synchronization primitives are difficult to compare and probably depend on the specific problem. Ideally, multiprocessor systems should support a set of flexible synchronization primitives and be sequentially consistent as well.

```
      PRODUCER:
L1:   If (test&set(x) = 1) then goto L1;
      If (s-r mod k = b) then {reset(x); goto L1;}
      <put message into buffer>
      s:= s+1 mod k;
      reset(x);
      goto L1;
      CONSUMER:
L2:   If (test&set(x) = 1) then goto L2;
      If (s-r mod k = 0) then {reset (x); goto L2;};
      <take message from buffer>
      r:=r+1 mod k;
      reset(x)
      goto L2;
```

Fig. 5. Producer-consumer algorithm using the Test&set primitive.

III. Buffering, Caching, and Combining

In high-end processors, the pipelining of instruction execution is common place and is aided by extensive prefetching and buffering of memory accesses.

A. Memory Access Buffering Techniques

In the IBM 3033 processor, operand prefetching is implemented through a set of operand registers [13]. Up to six operand accesses may be in progress at the same time. A similar procedure is implemented for the Stores. Stores to memory occur later in the pipeline, when the execution of an instruction is completed. Usually, Stores can be completed in one pipeline cycle by simply writing data and addresses to an operand Store buffer. Buffering Stores is particularly efficient because the processor does not need to wait for the return of information from memory, contrary to the Load of an operand. As a result, Loads are more critical for performance and are often given higher priority over Store requests.

In a multiprocessor system, memory accesses can be buffered in the processor, in the interconnection network, and in the shared memory. Several paths may be possible between a processor and a memory module. If invalidations or Stores have to modify an entry in a private memory or cache of another processor, then these updates may be buffered in the destination processor node. A good example is the BIAS filter in the IBM3033 which stores and selectively filters invalidation signals coming from other caches [14]. In general, systems with dual cache directories may be analyzed as systems with single invalidation buffers, since a lookup of the shadow directory introduces a delay of at least one cycle between the bus and the cache; a modification of the shadow directory does not immediately affect the cache content as "seen by" the processor [15]. Buffering instruction fetches is safe for a pipelined machine in a multiprocessor if instructions are not modifiable. Self-modifying code poses special problems which will not be addressed in this paper.

Buffer management refers to the order in which multiple buffered requests are treated. In most cases, the requests are serviced in a strict FIFO order (First-In-First-Out). In some cases requests may be allowed to pass each other in the buffer. This is referred to as *jockeying*. Jockeying is often permitted among memory requests for different memory words, but it is not permitted between requests destined to the same memory word. Jockeying with this restriction is called *restricted jockeying* in the rest of the paper. Whether or not any jockeying is permitted depends on the assumed concurrency model and the location of the buffer.

B. Caching Techniques

Memory caching is an effective method to reduce both the average memory access latency and contention. In multiprocessors, caching of shared writable data causes the well-known multiprocessor cache coherence problem. This problem has been addressed by many researchers [16], [17]. In [18], multiprocessor cache coherence is defined in the context of sequentially consistent multiprocessors.

To support a processor with extensive operand buffering, accesses to the cache are usually pipelined and the cache may be lockup-free [19]. A lockup-free cache does not block (or lock up) the processor on a miss. Rather, it records the status of the memory request causing the miss and keeps accepting and servicing requests from the processor.

C. Combining Interconnections

Combining interconnection networks are capable of *combining* Loads and Stores to the same memory location within the interconnection [5]. Loads can be completed before they propagate all the way to the memory. As a result, both Load latencies and memory contention are reduced. Combining networks may cause logical problems, since concurrent Load operations may return different values.

IV. Processor Behavior

A uniprocessor generally executes instructions one at a time, in the order specified by the program. If the processor is pipelined, however, then several consecutive instructions may be executed concurrently or even out of their intended order. This is allowable in uniprocessors, provided hardware mechanisms (interlocks) exist to check data and control dependencies between instructions to be executed concurrently [20]. This checking is local to the processor and can be done efficiently.

If processors are part of a multiprocessor which executes a parallel program, then such local dependency checking is still necessary but, depending on the assumed concurrency model, may not be sufficient to preserve the expected outcome of a concurrent execution. Since data are shared and interrupts can be sent between processors, processes running on different processors may affect the outcome of each other. Enforcing data dependencies among processors which are physically distant is not as easy as enforcing them in a single processor.

If the programmer or compiler "view" the multiprocessor as a fast multiprocessing uniprocessor, then the system has to be sequentially consistent to behave correctly. Lamport [21] defines sequential consistency as follows.

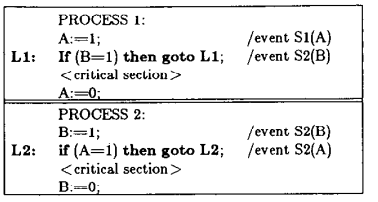
Definition 4.1—Sequential Consistency: A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

In order to apply this definition of sequential consistency, one has to define the meaning of *operation*. An operation may be as complex as a set of successive statements in a program or as simple as the latching of an address in a register. Usually a programmer assumes that, at the very least, simple load and store operations are indivisible. However, in a high level language statement such as $C \leftarrow A + B$, where A, B, and C are variables stored in memory, accesses from different processors may be interleaved between the Loads of A and B and the Store of C. In the mind of the programmer, there is an implicit ordering of events, e.g., A is first fetched then B is fetched, then C is stored.

The only way that two concurrent processes can affect each other's execution is through the sharing of data and the sending of interprocessor interrupt signals. Take for example the multitasked program of Fig. 6(a). In this program, two processes synchronize to enter a critical section through global variables A and B. (The fact that this algorithm allows deadlock is orthogonal to this discussion.) We define an *ordered interleaving* of memory accesses as an interleaving such that the references from each process appear in program order. The ordered interleavings of executions of events on global variables A and B in program 6(a) are displayed in Fig. 6(b). All six ordered interleavings of the first two statements of each program are possible. Note that, in Fig. 6(a), if process 1 is allowed to prefetch B before setting A to 1, and if process 2 is allowed to prefetch A before setting B to 1, a nonordered interleaving may result with both processors entering the critical section at the same time. Note that such prefetching does not violate any intraprocess dependencies—it does, however, violate an interprocess dependency.

For other types of synchronization and communication protocols the effects of nonsequentially consistent multiprocessors can be more subtle. Take for example the barrier protocol of Fig. 2. Apparently slave processes cannot overrun the synchronization barrier since an intraprocess dependency *does* exist. This is due to the fact that processes block *themselves* by setting flag[p] equal to 1 after an iteration and then spin on the flag. However, if the loop is viewed as an unravelled sequence of *do_iteration*(p, i) procedures, separated by barrier synchronizations, then it becomes apparent that the synchronization points do not prevent the process from "entering" the next iteration if, for example, the processor prefetches operands. This is the case since no intraprocess dependency exists between the loop which tests flag[p] and the next iteration of *do_iteration*(p, i). Consequently, the barrier synchronization of Fig. 2 might not function correctly in a nonsequentially consistent multiprocessor.

A programmer, having used the synchronization protocols given in Figs. 1 and 2, assumes the concurrency model of sequential consistency, and therefore a sequentially consistent multiprocessor must only allow ordered interleavings of events. Likewise, a programmer using the communication protocol of Fig. 3, will expect sequentially consistent behavior of the multiprocessor—otherwise the protocol will not function correctly. Since interprocess dependencies are caused by accesses to shared writable data (we ignore interrupts), the coherence enforced on such data by the memory system is also relevant. Censier and Feautrier define a coherent memory scheme as follows [22].



(a)

Ordered Interleavings	Results:
$S1(A) \rightarrow L1(B) \rightarrow S2(B) \rightarrow L2(B)$	Processor 1 enters CS.
$S2(B) \rightarrow L2(A) \rightarrow S1(A) \rightarrow L1(B)$	Processor 2 enters CS.
$S1(A) \rightarrow S2(B) \rightarrow L1(A) \rightarrow L2(B)$	Deadlock.
$S1(A) \rightarrow S2(B) \rightarrow L2(B) \rightarrow L1(A)$	Deadlock.
$S2(B) \rightarrow S1(A) \rightarrow L1(A) \rightarrow L2(B)$	Deadlock.
$S2(B) \rightarrow S1(A) \rightarrow L2(B) \rightarrow L1(A)$	Deadlock.

(b)

Fig. 6. (a) Synchronization protocol using two shared variables A and B. Initially $A = B = 0$. (b) All possible ordered interleavings of Stores and Loads for (a).

Definition 4.2—Memory System Coherence: 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.

In an environment where Stores can be buffered in a Store buffer associated with each processor and where multiple copies of data can exist, the notion of *latest Store* is vague. It is not clear whether it refers to the execution of the Store by a processor, or to the update of memory. In the next section this issue is discussed.

V. Ordering of Events in Multiprocessor Systems

When examined individually, processors and memories of multiprocessor systems can easily be proven to conform to the expected model of behavior (usually sequential consistency). But, when several processors and memories are active concurrently, the additional factor of sequencing of events among processors must be taken into consideration. In multiprocessor systems we are primarily concerned with *events* which can affect the overall system. If we disregard interrupts, then shared memory accesses are such events which can affect the entire system. Stores change the state of a memory word which, when subsequently read by another processor, can affect that processor's future activity. Loads enable processors to read shared memory words and thus enable them to be

affected by other processors. Store and Load events, however, cannot always be viewed to occur at specific instances, but rather sometimes occur relative to each other, independent of absolute real time [39].

Definition 5.1—Memory Request Initiating, Issuing, and Performing: A memory access request is *initiated* when a processor has sent the request and the completion of the request is out of its control. An initiated request is *issued* when it has left the processor environment[1] and is in transit in the memory system. A Load is considered *performed* at a point in time when the issuing of a Store to the same address cannot affect the value returned by the Load. A Store on X by processor i is considered *performed* at a point in time when an issued Load to the same address returns the value defined by a Store in the sequence $\{Si(X)\}+$.

In this definition, we denote Load and Store accesses by processor i on variable X as $Li(X)$ and $Si(X)$, respectively. In a system where memory Store operations become "observable" at the same time for all processors, the Store events on one variable can be ordered based on the physical time at which they are performed. The notation $\{Si(X)\}+$ is used to denote the sequence of Stores on X following the Store $Si(X)$ and including $Si(X)$ in the ordering based on the times when the Stores are performed. Similarly, the notation $\{Si(X)\}-$ denotes the sequence of Stores on X preceding $Si(X)$ but not including $Si(X)$.

For example, in the system depicted in Fig. 9, operand prefetching is implemented through a prefetch buffer at the processor. The processor *initiates* the operand prefetch by placing the operand's address in that buffer. Then the buffer controller *issues* the operand fetch to the shared interconnection and memory. The request transits in the interconnection and it is *performed* when it is latched in a buffer associated with the memory, provided this buffer is FIFO (restricted jockeying is allowed in the memory buffer). The situation is similar for a Store request. The processor *initiates* the Store by placing the request in a Store buffer at the processor. Later on, the Store buffer controller *issues* the Store request to the interconnection. The request is then in transit in the interconnection. It will be *performed* as soon as it is latched in the FIFO buffer associated with the destination memory.

Definition 5.1 is relevant whether or not multiple copies of the same data exist in the system. When multiple copies of data do exist, such as in cache-based systems (but not only limited to cache-based systems), we define the notion of *performed with respect to a processor.*

Definition 5.2—Performing with Respect to a Processor: A Store by processor i is considered *performed with respect to processor* k, at a point in time when an issued Load to the same address by processor k returns the value defined by a Store in the sequence $\{Si(X)/k\}+$. Similarly, a Load by processor i is considered performed with respect to processor k, at a point in time when an issued Store to the same address by processor k cannot affect the value returned by the Load.

The definitions of $\{Si(X)/k\}+$ and of $\{Si(X)/k\}-$ are similar to the definitions of $\{Si(X)\}+$ and $\{Si(X)\}-$. However, the Stores on X are ordered according to the time when they are performed with respect to processor k.

From the above definition it should now be clear why we consider some events not to occur at instances in time, but rather relative to each other. In some architectures, for example, at time t a Store may have been performed with respect to processor x but not yet with respect to processor y. It is certainly futile to attempt to associate a particular point in time with the occurrence of the Store event. Note that the notion of *performed* refers to the mutability of a subsequent event (i.e., after a Store is performed w.r.t. processor k, a Load by processor k *cannot* reflect anymore the value that was valid previous to the Store). It may be difficult for a processor to be able to pinpoint the exact moment when an access is performed with respect to another processor. In practice, the processor can detect that an access *has been* performed when it receives an acknowledgment signal, or if the maximum time to perform the access is known.

In systems where accesses are performed at different times with respect to different processors, we say that an access is *performed* at the point in time when it is *performed with respect to all processors.*

Definition 5.3—Performing an Access Globally: A Store is *globally performed* when it is performed with respect to all processors. A Load is *globally performed* if it is performed with respect to all processors and if the Store which is the source of the returned value is globally performed.

Once a Store is globally performed, no issued Load by any processor can return an old value which was valid before the Store took place. There is a subtle difference between a Load *performed with respect to all processors* and a *globally performed* Load. Once a Load is performed with respect to all processors, the value it returns is fixed and cannot be altered, independent of any action of any processor. When a Load is globally performed it is additionally true that any other issued Load by any processor cannot return a word which is "older" (i.e., a word in $\{Si(X)\}-$) than the word returned by the performed Load.

At this time we can distinguish between two types of memory systems. Memory systems in which accesses are atomically performed for all and any copies of the data are referred to as systems with *atomic accessibility.* In the second type of system, memory is not atomically accessible. In systems with atomic accessibility, the hardware causes the memory system to appear as unique memory cells associated with each address, to which accesses are

[1]The processor environment includes the CPU and local buffers.

serialized. In systems with nonatomic accessibility the hardware does not enforce such a memory model and updates of different copies of data may be observably staggered over time. Loads can always be considered as atomic, since a Load access by a processor interacts with the rest of the system only at the time when the returned value is bound; even if the Load has to propagate through the system, it remains "private" to the issuing processor, except at the point in time when the value is bound and this binding is always atomic. A Store, on the other hand, can affect the rest of the system at different points in time while it propagates. Hence, atomic accessibility refers to the manner in which Stores are propagated throughout the memory system.

VI. Strong and Weak Ordering of Events

In a multiprocessor with atomically accessible memory an appropriate refinement of condition 4.2 becomes easy. In such a system, memory coherence is upheld if a Load to a memory location always returns the value of a Store, to the same memory location, performed most recently before the Load is performed. Note, that this definition of memory coherence does not imply sequential consistency. Processors could still issue memory accesses out of program order: in this case, memory coherence is maintained, but sequential consistency is violated. However, sequential consistency *does* imply memory coherence in multiprocessors with atomic memory accessibility.

In systems without atomic memory accessibility the concept of memory coherence becomes much more difficult to interpret. This is due to the fact that the time at which a Store is performed with respect to distinct processors varies for each processor. An event having occurred "recently" with respect to one processor may occur in the future with respect to another processor. If such a multiprocessor should be sequentially consistent then it is less ambiguous and simpler to rely on the rules of sequential consistency as the conditions of correctness rather than to prove memory coherence.

A. Strong Ordering of Memory Accesses

Strong ordering of memory access is the hardware quality of a multiprocessor which guarantees sequential consistency without further software aid. Note that a nonstrongly ordered architecture can be made to appear sequentially consistent with proper compiler support. In a system with access atomic memories, it is easy to see that accesses are strongly ordered by simply performing them in program order. That is, to guarantee strong ordering of accesses in such a system, accesses must be initiated and issued so that they will be performed in order. This is true because before the access has been performed, it can be considered as "private" to the processor that issued it. Immediately after they are performed, a Store affects all other processors and a Load ceases to be affected by all other processors at the same time. The only problem is therefore to ensure that successive accesses from the same processor affect or are affected by all other processors in program order. It can generally be assumed that it is simple to control the sequencing of accesses up to the issuing level. Once an access is issued it must not be possible for subsequent access by the same processor to pass it.

Condition 6.1—Strong Ordering of Memory Accesses in Systems with Access Atomic Memory: Strong ordering of memory accesses is preserved in multiprocessors with atomic memory accessibility if individual processors perform all accesses in program order.

It is easy to see that condition 6.1 enforces sequential consistency and thus memory coherence [23], but the question remains whether it is necessary. In Fig. 7, three program fragments are given. They are sequences of Loads and Stores on different variables. For each fragment a specific condition for sequential consistency is given. It can be seen that if any two accesses belonging to the same process are performed out of order, then this condition may be violated.

A distinct problem is the possibility that a processor bypasses the memory system for successive accesses to the *same* variable, a technique called short-circuiting in [20]. A Load on a variable directly following a Store on the same variable can bypass the memory because the resulting interleaving is ordered. However, the Store must be performed before any other subsequent access to a different shared variable can be performed. In essence, the atomic memory accessibility of condition 6.1 is violated if short-circuiting is allowed. Therefore, a multiprocessor which allows short-circuiting does not constitute a system with access atomic memory, per se. Consequently, the conditions defined in the next section should be applied.

Condition 6.1 is not sufficient to satisfy strong ordering in a system where the memory is not access atomic. Take for example the program fragment of Fig. 8(a) further illustrated in Fig. 8(b). In this program, if $L2(A)$ reads the value produced by $S1(A)$, and if $L3(B)$ returns the value produced by $S2(B)$, then $L3(A)$ must also return the value produced by statement $S1(A)$. However, if, for some reason, event $S1(A)$ takes much more time to propagate to processor $P2$ than it does to processor $P3$, then there may be enough time (depending on conflicts and distances) for $P2$ to perform event $L2(A)$, then $S2(B)$, and for $P3$ to perform $L3(A)$ on a value of A previous to $S1(A)$. This is not an ordered interleaving of accesses.

The necessary condition for strong ordering of memory accesses can be derived from the paper by Lamport [23].

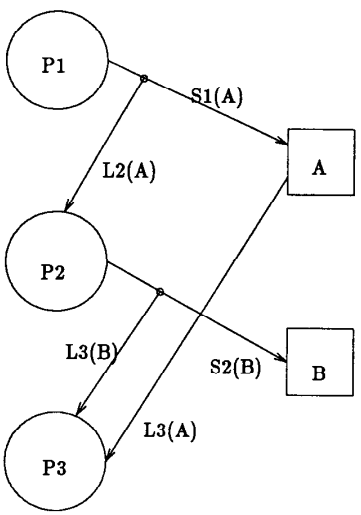
Condition 6.2—Strong Ordering of Memory Accesses in Systems with Nonaccess Atomic Memory (I): Strong ordering of memory accesses is preserved in systems with nonaccess atomic memory if accesses by any one processor are performed in program order with respect to each processor, and if the order in which all Stores are performed is the same with respect to all processors.

FRAGMENT 1:	
PE1:	PE2:
S1(A)	S2(B)
L1(B)	L2(A)
If L1(B) returns a value in {S2(B)}- then L2(A) must return a value in {S1(A)}+.	
FRAGMENT 2:	
PE1:	PE2:
L1(A)	L2(B)
S1(B)	S2(A)
If L1(A) returns a value in {S2(A)}+ then L2(B) must return a value in {S1(B)}-.	
FRAGMENT 3:	
PE1:	PE2:
S1(A)	L2(B)
S1(B)	L2(A)
If L2(B) returns a value in {S1(B)}+ then L2(A) must return a value in {S1(A)}+.	

Fig. 7. Behavior of three program fragments in a sequentially consistent system.

Processor 1	Processor 2	Processor 3
.	.	.
.	.	.
S1(A)	L2(A)	L3(B)
.	S2(B)	L3(A)
.	.	.
.	.	.

(a)



(b)

Fig. 8. (a) Three processes sharing variables A and B. (b) Possible outcome of concurrent execution of programs in 8(a).

In most cases, the particular sequencing of events, as is required to fulfill condition 6.2, cannot be enforced without implying more stringent control over events. Specifically, in generic systems where all timings are random, the following condition becomes more applicable.

Condition 6.3—Strong Ordering of Memory Accesses in Systems with Nonaccess Atomic Memory (II): Condition 6.2 is satisfied in *any* system if an access may not be performed with respect to any processor until the previous access by the same processor has been globally performed.

Note that this condition implies condition 6.2. It is important to realize that, if the memory is access atomic, condition 6.2 becomes equivalent to condition 6.1. This stems from the fact that, in such a system, accesses are performed with respect to all processors at the same time. Another observation is that all the program fragments given so far will execute correctly in a system which complies to condition 6.2. The problem with the program fragment of Fig. 8(a) and its execution sequence depicted in Fig. 8(a) is that $S1(A)$ was performed with respect to $P2$ when $P2$ issued $S2(B)$ but was not performed with respect to $P3$ when $L3(B)$ was performed. Therefore condition 6.2 is violated. To satisfy condition 6.3 and therefore condition 6.2, it suffices to enforce that $L2(A)$ is globally performed before $S2(B)$ can be performed with respect to $P3$.

B. Weak Ordering of Memory Accesses

To maintain strong ordering, potential dependencies on *every* data access to shared memory have to be assumed. However, most of these data are not *synchronizing variables*, i.e., shared variables used to control the concurrency between several processes [such as variables A and B in Fig. 6(a)]. In multitasked programs such variables are used to synchronize processes and to maintain the integrity of shared modifiable data structures or variables.

Strong ordering of memory accesses forbids most jockeying in queues and often disallows the pipelining of accesses, because accesses have to be (globally) performed in order. Especially, in systems with complex packet-switched networks where the network delay may be high, a new access may not be issued until an acknowledgment has been received that the previous access has been globally performed. In most cases, though, performing accesses out of their order will not cause logical problems. The policy in which shared memory accesses can be issued optimally is called *weak ordering of memory accesses* and is introduced and defined below.

In a system with weak ordering of accesses, two types of shared variables are distinguished: first the shared operands appearing in algorithms whose values do not control the concurrent execution; and second synchronizing variables which protect the access to shared writable operands or implement synchronization among different processes. If a shared variable is modified by one process and is accessed by other processes, then the access to the variable must be protected, and it is the responsibility of the programmer to ensure mutual exclusion for each access to the variable by using high-level language constructs such as critical sections [7]. Critical sections are in turn implemented by basic synchronization primitives such as locks. It is assumed that, at run time, the system can distinguish between accesses to synchronizing variables and to other shared variables. Synchronizing variables can be distinguished by the type of instruction (Test_and_set,

Compare_and_swap, Fetch_and_execute, Reset or special Load and Store instructions, for example).

Definition 6.1 Weak Ordering of Memory Accesses: In a multiprocessor system, memory accesses are *weakly ordered* if

1) accesses to global synchronizing variables are strongly ordered,

2) no access to a synchronizing variable is issued by a processor before all its previous global data accesses have been globally performed, and

3) no access to global data is issued by a processor before its previous access to a synchronizing variable has been globally performed.

The dependency conditions on shared variables in such a system are not checked continuously but only at explicit synchronization points. Between two consecutive operations on hardware-recognized synchronization variables, no assumption can be made by the programmer of a process about the order in which Stores are observed by other processes. The order of successive Stores by a processor, to the *same* address, must however be respected. Buffering and restricted jockeying are allowed in all buffers, except for operations on hardware-recognized synchronizing variables.

In order that the program of Fig. 6(a) executes correctly in a system with a weak ordering of memory accesses, variables *A* and *B* must have been declared as synchronizing variables. Special Load and Store instructions may therefore be generated by the compiler for such variables.

VII. Access Ordering in Different Architectures

The rules to implement strong ordering of memory accesses have been defined in conditions 6.1 and 6.2 for access atomic and nonaccess atomic memories, respectively. We now examine how these rules may be enforced in different multiprocessor system architectures.

We consider five different types of systems in the following discussion. These structures are representative of the system structures for multiprocessors with global data. In the following, the adjectives *shared* or *private* refer to the physical location of the memory. The adjectives *global* or *local* refer to the accessibility of the data. Global data are accessible by all the processors while local data are only accessible by the local processor. Similarly, a local buffer can only be accessed by one processor.

A. Multiprocessor Systems with Shared Global Memory

In the system of Fig. 9, the shared memory is accessed by all processors The memory is access atomic since any access is performed at the same time with respect to all processors as soon as it is buffered at the memory buffer, provided the buffer is FIFO. An example of this type of system is the C.mmp [24]. This system has been analyzed by Lamport [21]. It is assumed that the interconnection network is passive. If the delay through the network is constant or bounded for all requests (a rare case in practice because of conflicts), or if there is only one path from

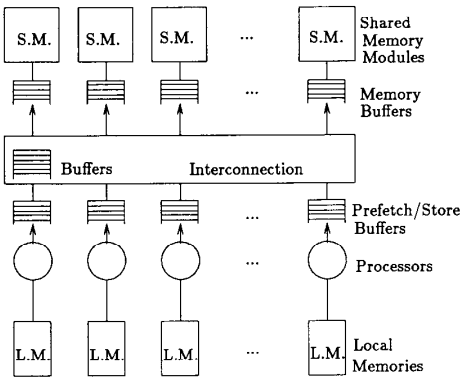


Fig. 9. System with shared global memory.

processors to memories (e.g., in single bus systems) then successive requests can be issued in program order without waiting for acknowledgments from the memory. In general, however, because of conflicts and random delays, the only way that a processor can ensure that its global data requests are performed in program order is to issue the requests one at a time and to wait for an acknowledgment after each request. The following are the rules for enforcing strong ordering of events in the system shown in Fig. 9.

1) Global memory accesses are performed at the FIFO buffer of the destination memory.

2) Individual processors initiate global data accesses in program order. These accesses (both for Loads and Stores) are buffered in the same local buffer associated with the processor. This combined buffer is managed by a strict FIFO policy. Short-circuiting of memory accesses [20] (i.e., bypassing the memory) in a processor is restricted as explained in Section VI-A.

3) The controller of the combined Prefetch/Store buffer issues and performs the memory accesses one-by-one, in the FIFO order of the buffer.

B. Multiprocessor System with Distributed Global Memory

A multiprocessor with distributed global memory which consists of an interconnection of private memories is considered here. Several existing multiprocessors fall into this category, such as the Cm*, the BBN Butterfly, and the IBM RP3 [24], [25], [6]. The private memories contain global, as well as local data and are accessed randomly (no multiple copies of data exist) (Fig. 10). The systems depicted in Figs. 10(a) and (b) differ in that memories in the second system are accessed via two separate buffers, while the system of Fig. 10(a) uses only one buffer per memory. In the system of Fig. 10(b), the Prefetch/Store buffer queues accesses from one processor only (the processor which *owns* that particular memory module), while the Remote Access buffer queues accesses coming from all other processors. In the system of Fig. 10(a), all accesses to a particular memory module are queued in the same Memory buffer.

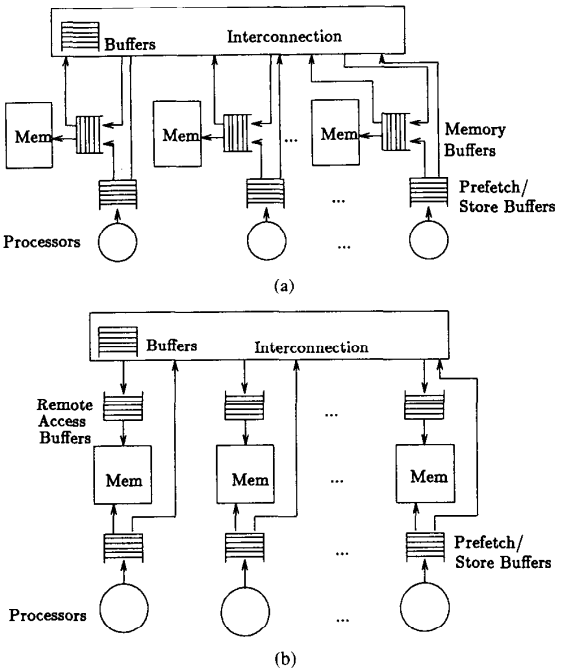


Fig. 10. System with distributed global memory. Remote accesses are buffered in Memory buffers. (b) System with distributed global memory. Remote accesses are buffered in Remote Access buffers.

It should be evident that the system depicted in Fig. 10(a) is logically equivalent to the system shown in Fig. 9. Access delays due to remote and local accesses are different in system 10(a), but such delays were never taken into consideration when the conditions for strong ordering were laid down in the previous subsection. Hence, the conditions for strong ordering of events in system 10(a) are the same as those for the system shown in Fig. 9. Any access is performed when it is buffered in the Memory buffer of the destination memory. All accesses are *initiated* by being placed in the Prefetch/Store buffer of the processor. A private access is *performed* as soon as it is *issued* by the Prefetch/Store buffer to the local Memory buffer. A remote access is *issued* by the Prefetch/Store buffer by introducing it to the interconnection. The access is *performed* when it is latched in the destination Memory buffer. It should be evident that a Prefetch/Store buffer should wait until the most recent remote access has been performed before it can issue another access. A well-known example of the system shown in Fig. 10(a) is the Cm* [24].

In the system depicted in Fig. 10(b), remote accesses are buffered in a distinct remote access buffer. It should be obvious that the memory of this system in its general form is not access atomic. Once an access is buffered in one of the two queues it is only performed with respect to the processor(s) which use(s) the same queue (provided that queues are accessed in FIFO order). The policy used in sequencing accesses queued in different buffers will affect the logical properties of the system.

If accesses by the two buffers are interleaved randomly, then no access is globally performed until it is executed at the memory. The Prefetch/Store buffer, however, preserves the order of the accesses initiated by the local processor. Remote accesses initiated by the local processor are only performed when they are executed in the remote memory. This strategy conforms to condition 6.2. This scheme may be inefficient, because in the case of a remote access the Prefetch/Store buffer is blocked until the remote access has been dequeued at the appropriate Remote Access buffer. If Remote Access buffers often contain many pending accesses, this blocking time can be relatively long.

One way of avoiding the blocking of the Prefetch/Store buffer is to assign an absolute priority to Remote Access buffers. This means that all accesses waiting in Remote Access buffers are serviced before accesses waiting in Prefetch/Store buffers. When a remote access is buffered at the Remote Access buffer it can be considered globally performed and the local access buffer which issued it may proceed without blocking. A remote access is globally performed when it is buffered at the Remote Access buffer of memory module X because

1) it is performed with respect to all processors which access memory module X via the Remote Access buffer (this is due to the FIFO nature of this buffer) and

2) it is performed with respect to the processor which accesses memory module X via the Prefetch/Store buffer, because of the absolute priority of the Remote Access buffer over the Prefetch/Store buffer.

The time to perform an access in system 10(b) may be quite long, if the buffers are not prioritized. Weak ordering may be more efficient, as supported by the simple model of the next section. In the case of weak ordering, the Prefetch/Store buffer issues local references by starting the memory cycle and issues remote references by simply latching them in the first stage of the interconnection. However, whenever a processor executes an instruction on a declared synchronizing variable (this is detected by the fact that the data have been tagged by the compiler, or that special instructions are used), it must ensure that all its previous data accesses have been globally performed, and stop issuing Prefetches and Stores of operands until the access to the synchronizing variable is also globally performed. In practice, the processor waits for an acknowledgment signal or for a time-out period equal to the worst-case delay to globally perform the access.

C. Multiprocessor Systems with Private Caches

Figs. 11(a) and (b) show multiprocessor architectures with shared global memory and private caches. One example of this type of system is the Sequent Symmetry multiprocessor [26]. The shared memory contains code and data, and the caches are accessed associatively. Caches may be write-through or write-back caches [27]. If no global writable data can be loaded into caches then no coherence problem exists between the caches. This technique relies on software to avoid the coherence prob-

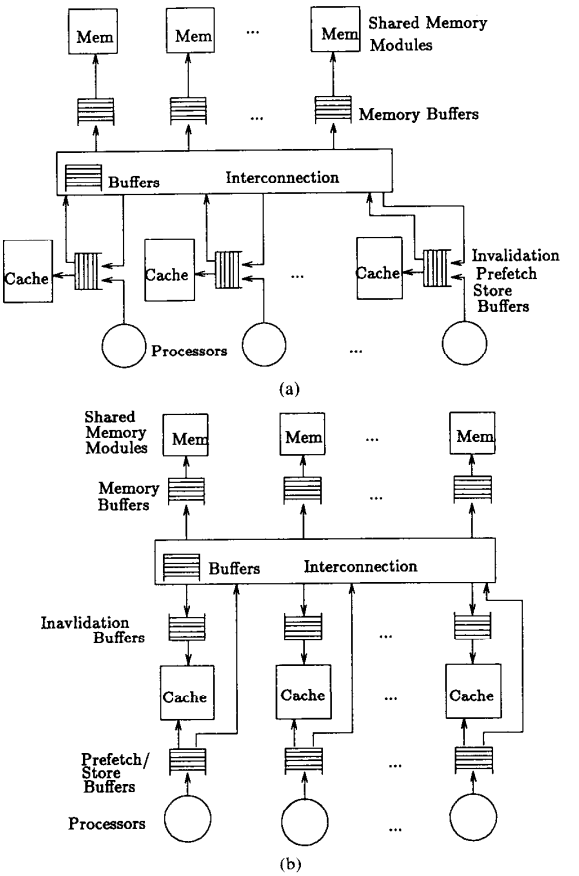


Fig. 11. (a) Cache-based system. Invalidations are buffered in the Prefetch/Store buffers. (b) Cache-based system. Invalidations are buffered in independent Invalidation buffers.

lem. At any time, the caches contain local data or nonmodifiable global data. The distinction between the types of data is made at compile-time, possibly with some indication from the programmer. Accesses to global writable data in the shared memory can be buffered at the processor in a common Prefetch/Store buffer. With respect to buffering, the problems with cache-based systems in which the cache never contains shared writable data are very similar to the problems analyzed in Section VII-A.

Of particular interest is the case of multiprocessors with private caches that can contain global data and with hardware-enforced coherence [22], [16]. Coherence among multiple copies of these data can be maintained through hardware invalidation signals. Also, in some coherence algorithms, a processor may broadcast a Load to all caches in the case of a miss, in order to read the data directly from another cache. In the following, "P-data" refers to data that are private to the cache (one single copy exists) and "S-data" refers to data that are shared among several caches (several copies may exist in different caches) [16].

Two buffer configurations are shown in Fig. 11. In Fig. 11(a), there is a unique buffer per processor. The buffer contains data Prefetch and Store requests for the local processor plus the accesses made by remote processors (Loads or Invalidations). The local processor initiates its Stores and Loads in the private Prefetch/Store buffer. No jockeying is allowed in this buffer. The private buffer controller issues requests to the cache one at a time. A Store request on S-data in the cache may require invalidating the data in other caches. A Store issued [Fig. 11(a)] by processor i is performed with respect to processor k when the cache is updated (Store hit on P-data) or at a point in time when it is placed in the memory buffer and when an invalidation (if it is necessary) has been placed in the local buffer of processor k (Store to S-data). Similarly, on a Load miss, a cache may read a block from a different cache. A Load request on a miss in processor i is performed with respect to processor k when the Load request is buffered in the local buffer of processor k.

The Loads causing misses and the Stores on S-data causing Invalidations can be broadcast on a bus to all caches and to memory so that they are performed with respect to all processors at the same time. This means Stores can be performed atomically; this solution can be extended to systems with a few buses.

In the system of Fig. 11(b), the buffer for Invalidations and for Loads issued by remote processors is distinct from the buffer for Loads and Stores from the local processor. An invalidation of a block in a remote cache of processor k is performed with respect to processor k when it is executed in the remote cache. Alternatively, the Invalidation buffer could have priority over the local processor, as described in Section VII-B.

In a weakly ordered system, the processors can issue shared memory requests without waiting for previous requests to be performed. This results in a system with very high efficiency. In this case, the only troublesome accesses are accesses to synchronizing variables. The buffer controller must still record the status of all cache accesses that it has issued but not performed, so that it can perform them every time an access to a synchronizing variable is detected. The implementation of such a buffer may be very complex. The details of an implementation for a given cache coherence mechanism would be interesting in order to understand the practical aspects of the concept of weak ordering in cache-based systems, but it is beyond the scope of this paper. How a cache-based multiprocessor can be lockup-free and maintain weak ordering of accesses in discussed in [28].

Consider again the system of Fig. 11(a). But this time, let us assume that Invalidations are not broadcast on a bus but rather are propagated from cache to cache by the interconnection. It is interesting to note, that sequential consistency is preserved in this system, provided the memory copy is locked during the propogation of the Invalidations and provided that the invalidated caches cannot obtain a "new" copy of the invalidated block until all other caches have been invalidated as well. The details on maintaining sequential consistency in such a system are discussed in [18].

D. Multiprocessor Systems with Combining Networks

In Section VII-A, we have assumed that the network does not combine accesses, i.e., no access can be performed with respect to *any* processor while it is in transit in the interconnection network. Combining networks such as the one proposed for the NYU ultracomputer [5] are not passive. In this section we analyze the logical properties of such networks for sequences of Loads and Stores in light of condition 6.3.

1) Combining of Ordinary Load and Store Accesses: When memory accesses "collide" in a switch box of a combining network the following combinings occur, depending on the type of access.

a) Case of Multiple Loads to the Same Variable x: Only one Load is propagated to memory. The other Loads are buffered in the switch node until the return of the value from the memory; then the value is returned to all processors having issued the Load.

b) Case of Multiple Loads and Stores on the Same Variable x: Only one value defined by one of the Stores is propagated to memory. Acknowledgments or data values defined by the Store are returned to all processors having issued the Loads or Stores.

c) Case of Multiple Stores on the Same Variable x: Only one value defined by one of the Stores is propagated to memory. Acknowledgments are returned to all processors having issued the Stores.

Memory accesses can terminate in three ways. Either an access terminates at the memory, or it terminates at a switch with combining, or it is eliminated at a switch. In case a), all but one Load terminate at the switch where the combining takes place. These Loads wait for a value to be returned by the not yet terminated Load which propagates towards memory. This Load either combines again with other accesses at another switch or terminates at the memory. A Load which terminates at the memory is globally performed when the Store which defined the value seen by the Load has been globally performed (Definition 5.3). As soon as a Load terminating at the memory is globally performed, all Loads which are waiting for a value to be returned by the Load are also globally performed, provided the blocked Loads do not combine with any Store. On the other hand, the blocked Loads could combine with Stores: if Loads waiting at a switch for a value to be returned by the Load propagating towards memory have the possibility to combine with a Store to the same memory location, then they may do this under the provisions described for combining Loads and Stores [case b)].

In case b) the Load(s) combining with a Store terminate at the switch. Such Loads are not globally performed until the Store which they combine with is globally performed. The Loads may return the bound value to their respective processors, but assuming condition 6.3, the processors may not issue another global memory access until the Store propagating towards memory has been globally performed. A Store terminating at memory is globally performed once all copies of the to be written data have been updated. A Store which has previously combined with Loads may also subsequently combine with more Loads or other Stores [case c)].

In case c) all but one Store are eliminated at the switch. A single Store propagates towards memory. The Stores combining may have previously combined with other Stores or Loads. A Store which is eliminated by combining with other Stores is globally performed once the remaining Store, propagating towards memory, has been globally performed. This is due to the strict interpretation of condition 5.3 which states that an access is not performed if a subsequent Load can return an "old" value. Even though an eliminated Store may have no impact on the system, if sequential consistency is to be preserved, then a "new" value must become visible due to the eliminated Store. A Store may be eliminated even if it has previously combined with Loads.

Note, that the above rules refer to correctness under sequential consistency. Implementing these rules may be difficult. For example, if Loads combine with a Store and wait for some type of acknowledgment for the Store to be globally performed, the implementation of the acknowledgment mechanism will be complicated if the Store can be eliminated in transit to memory.

2) Combining of Atomic Read_Modify_Write Accesses: The Fetch_and_add instruction returns the value of a memory location and increments it atomically by a selectable value. It is desirable to allow multiple Fetch_and_Adds to combine in an interconnection, even if ordinary Loads and Stores cannot combine. This is due to the fact that the destination of Fetch_and_add accesses often forms a *hot spot* (i.e., a memory location for which contention is high). Multiple Fetch_and_adds can combine in a switch by buffering all but one Fetch_and_add access at the switch and forwarding a single modified Fetch_and_add which returns the unmodified memory location value and increments it by the sum of the increments values of the combined Fetch_and_adds. Multiple modified Fetch_and_add operations may also combine in later stages of the network. Each buffered Fetch_and_add instruction returns a differently offset value of the value returned by the Fetch_and_add which terminated at the memory. Fetch_and_add instructions are globally performed once an ultimate modified Fetch_and_add terminates at the memory.

Other indivisible Read_modify_write operations can combine at switch nodes under similar rules. Likewise, Read_modify_write operations may also combine with ordinary Load or Store accesses to the same memory location.

E. Multiprocessor System with Replicated Memories

Condition 6.2 is difficult to apply in some practical situations. This is why we have introduced condition 6.3. However, this condition is overly restrictive. An example for which condition 6.2 is verified while condition 6.3 is not upheld is given by Collier [29].

The system described by Collier is a ring of processors where each processor maintains a copy of all global data. A processor always performs Loads using its own copy of the global data only. Stores, however, must be propagated to all other processors' memories as well. The updating of *all* copies of the data, including the writing processor's own copy, always occurs in the same order, i.e., the copy of processor $P0$ is first updated, then the copy of processor $P1$ and so on to processor Pn. These updates, may be propagated around a ring. A processor issuing a Store to X is blocked until the copy of processor Pn has been updated. However, a Load is always executed in the local memory without waiting.

This system does indeed satisfy condition 6.2. However, because of the deterministic way in which Stores are propagated, the system does not have to satisfy condition 6.3 in order to be strongly ordered. In particular, a processor does not need to wait until a Load is globally performed before issuing and performing the next Load.

VIII. Simple Performance Analysis

In this section we present a simple method comparing the upper bounds of the MIPS rates achievable by three designs: a design with no buffers at the processors, a design with strong ordering of accesses as specified by condition 6.3, and a design with weak ordering of events. Each processor alternates between compute and wait phases. A processor "computes" while it accesses local data and instructions. On a reference to global data, it has to wait for the completion of the access. This waiting time depends on the type of the access.

Let t_p be the average duration of the compute phase between two successive data accesses to global memory in one of the P processors. If p_s is the probability that an instruction contains an access to global data and $I_{s.p.}$ is the MIPS rate of a processor when all accesses are local (single processor configuration), then $t_p = 1/(I_{s.p.}p_s)$. The memory system is characterized by t_{issue} and t_{perform}, the minimum times to issue and to perform a request respectively. All these parameters depend on the machine design.

We neglect memory access conflicts as well as dependencies in the CPU. We also assume that all buffers are infinite. The following results show the relative effects of the three design approaches. Let t_{inref} be the average interreference time between two consecutive accesses to global variables by the same processor, i.e., it is the total duration of the compute and the wait phases between two accesses to global memory; the following inequalities define upper bounds of the MIPS rate of the three systems.

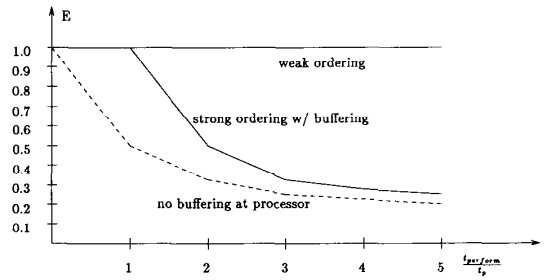


Fig. 12. Comparison of systems without buffering (dashed line) and with buffering and strong ordering (solid line). Comparison of buffering with strong and weak ordering ($t_{\text{issue}} < t_p$).

$$t_{\text{inref}} \geq t_p + t_{\text{perform}} \text{ (no buffering at the processor)},$$

$$t_{\text{inref}} \geq \text{MAX}\,[t_p, t_{\text{perform}}] \text{ (buffering with strong ordering), or}$$

$$t_{\text{inref}} \geq \text{MAX}\,[t_p, t_{\text{issue}}] \text{ (buffering with weak ordering).}$$

The first inequality results from the fact that, in a nonbuffered system, the processor is blocked during the time it performs globally a shared memory access. In the second or third cases, the processor and the Prefetch/Store buffer controller form a pipeline with average segment times of t_p and t_{perform} (strong ordering), or t_p and t_{issue}, (weak ordering.) The throughput of this pipeline is determined by the bottleneck segment.

The efficiency of the multiprocessor system denoted E is the ratio of the MIPS rate of a processor in the tightly-coupled ($I_{t.c.}$) and in the single processor ($I_{s.p.}$) configurations.

$$E = (I_{t.c.}/I_{s.p.}) = (t_p/t_{\text{inref}}).$$

From the above formulas, we can see that the effectiveness of various designs depends on the relative values of t_p and t_{issue} or of t_p and t_{perform}. Note that the value of t_p depends both on the MIPS rate of the processor as a single processor, and on the probability that an instruction references data in the global memory. In Fig. 12 the upper bounds on multiprocessor efficiencies in the cases of no buffering at the processor and of buffering with strong ordering are compared as a function of t_{perform}/t_p. Buffering is more attractive for highly-pipelined machines, and for cases where the access rate to global memory is high. For the third curve in Fig. 12, we have assumed that $t_{\text{issue}} < t_p$ and the lower bounds on multiprocessor efficiency of buffered systems with weak or strong ordering is shown as t_{perform}/t_p increases. This would be the case if the number of processors increases and the interconnection network is packet-switched so that the delay through the network increases, but the time to issue remains the same. It appears from the lower bound model, that weak ordering in buffered systems is only effective for systems where $t_p < t_{\text{perform}}$.

IX. Conclusion

We have presented in this paper a framework to analyze the coherence properties of shared memory multiproces-

sor systems. The concepts and results presented in this paper are extensions of Lamport's results [23], [21], [30]. We have introduced states in which a global memory request may be. We have demonstrated that these states are fundamental by using them to analyze the logical properties of multiprocessor systems.

To alleviate the performance problems with strong ordering, we have introduced the concept of weak ordering of memory accesses. Weak ordering results in the highest possible processor efficiency. A weakly ordered system is not sequentially consistent. The programmer must declare explicitly what we have called synchronizing variables, i.e., variables used to synchronize processors, and to protect the integrity of shared writable data through mutual exclusion.

Contrary to the "common wisdom" a processor in a multiprocessor system does not have to block on every shared memory accesses for the system to be strongly ordered. Multiple global memory accesses may be in progress at any time, provided that conditions 6.1, 6.2, or 6.3 are respected. These conditions permit a considerable flexibility in the buffering of accesses at different levels of the memory system. More surprising is probably the fact that memory accesses atomicity is not a necessary condition for sequential consistency in asynchronous multiprocessors.

A simple model has been presented to give insight into the effectiveness of various design approaches for different ranges of system parameters. The fundamental approach taken in this paper has allowed us to identify the basic restrictions of buffering imposed by the two ordering policies in the case of some very complex systems, such as cache-based systems. We believe that more work is warranted in this direction. More precisely, the implementation problems caused by the buffering mechanisms and the cost effectiveness of each scheme should be identified. However, even if the mechanisms are complex they may be necessary if we want to take advantage of the very fast VLSI uniprocessors (that are being developed) in multiprocessor configurations. Indeed, while it is possible with current technology to build very fast pipelined uniprocessors, the delays incurred in bus protocols and shared memory transfers are very difficult to overcome in the context of multiprocessors.

REFERENCES

[1] *Special Session on Commercial Cache-Based Multiprocessors*, in *Proc. 12th Int. Symp. Computer Architecture*, June 1985, pp. 208-240.

[2] G. Tucker, "The IBM 3090 system: An overview," *IBM Syst. J.*, vol. 25, no. 1, pp. 4-19, Jan. 1986.

[3] S. C. Chen, "Large-scale and high-speed multiprocessor system for scientific applications—CRAY-X-MP-2 series," in *Proc. NATO Advanced Research Workshop High Speed Computation*, Nuclear Research Center, Julich, West Germany, June 1983.

[4] D. Gajski *et al.*, "CEDAR: A large scale multiprocessor," *Comput. Architecture News*, ACM Sigarch, Mar. 1983.

[5] A. Gottlieb *et al.*, "The NYU ultracomputer—Designing an MIMD shared memory parallel computer," *IEEE Trans. Comput.*, vol. C-32, no. 2, pp. 175-189, Feb. 1983.

[6] G. F. Pfister *et al.*, "The IBM Research parallel processor prototype (RP3): Introduction and architecture," in *Proc. 1985 Int. Conf. Parallel Processing*, Aug. 1985, pp. 764-771.

[7] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *Comput. Surveys*, vol. 15, no. 1, pp. 3-43, Mar. 1983.

[8] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, no. 8, pp. 453-454, Aug. 1974.

[9] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, pp. 569-570, Sept. 1965.

[10] D. E. Knuth, "Additional comments on a problem in concurrent programming control," *Commun. ACM*, vol. 9, no. 5, May 1966.

[11] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Software Eng.*, vol. SE-3, no. 2, pp. 125-143, Mar. 1977.

[12] M. Dubois, C. Scheurich, and F. Briggs, "Synchronization, coherence, and event ordering in multiprocessors," *Computer*, vol. 21, no. 2, pp. 9-21, Feb. 1988.

[13] W. D. Connors, "The IBM 3033: An inside look," *Datamation*, pp. 198-218, May 1979.

[14] B. M. Bean, "Bias filter memory for filtering out unnecessary interrogations of cache directories in a multiprocessor system," U.S. Patent 4 142 234, Feb. 27, 1979.

[15] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *Proc. 10th Int. Symp. Computer Architecture*, June 1983, pp. 124-131.

[16] M. Dubois and F. A. Briggs, "Effects of cache coherency in multiprocessors," *IEEE Trans. Comput.*, vol. C-31, no. 11, pp. 1083-1099, Nov. 1982.

[17] J. Archibald and J.-L. Baer, "An evaluation of cache coherence solutions in shared-bus multiprocessors," Dep. Comput. Sci., Univ. Washington, Seattle, Tech. Rep. 85-10-05, Oct. 1985.

[18] C. Scheurich and M. Dubois, "Correct memory operation of cache-based multiprocessors," in *Proc. 14th Annu. int. Symp. Computer Architecture*, June 1987, pp. 234-243.

[19] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. 8th Annu. Symp. Computer Architecture*, June 1981, pp. 81-85.

[20] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.

[21] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690-691, Sept. 1979.

[22] L. M. Censier and P. Feautrier, "A solution to coherence problems in multicache systems," *IEEE Trans. Comput.*, vol. C-27, no. 12, pp. 1112-1118, Dec. 1978.

[23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[24] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

[25] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar, "Performance measurements on the 128-node butterfly parallel processor," in *Proc. 1985 Int. Conf. Parallel Processing*, Aug. 1985, pp. 531-540.

[26] "Symmetry technical summary," Sequent Computer Systems Inc., Beaverton, OR, 1987.

[27] A. J. Smith, "Cache memories," *Comput. Surveys*, vol. 14, no. 3, pp. 473-530, Sept. 1982.

[28] C. Scheurich and M. Dubois, "Concurrent miss resolution in multiprocessor caches," in *Proc. 1988 Int. Conf. Parallel Processing*, Aug. 1988, pp. 118-125.

[29] W. W. Collier, "Architectures for systems of parallel processes," IBM Corp. Tech. Rep. TR00.3253, Jan. 27, 1984.

[30] L. Lamport, "The mutual exclusion problem: Part I-A theory of interprocess communication," *J. ACM*, vol. 33, no. 2, pp. 312-326, Apr. 1986.

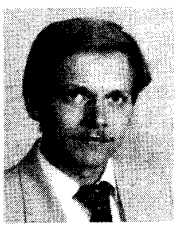
[31] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors," in *Proc. 11th Int. Symp. Computer Architecture*, June 1984, pp. 340-347.

[32] W. C. Brantley, K. P. Mc Auliffe, and J. Weiss, "RP3 processor-memory element," in *Proc. 1985 Int. Conf. Parallel Processing*, Aug. 1985, pp. 782-789.

[33] F. A. Briggs and M. Dubois, "Effectiveness of private caches in multiprocessors with parallel-pipelined memories," *IEEE Trans. Comput.*, vol. C-32, no. 1, pp. 48-59, Jan. 1983.

[34] F. A. Briggs, "Effects of buffered memory requests in multiprocessor systems," in *Proc. ACM/Sigmetrics Conf. Simulation, Measurements, and Modeling of Computer Systems*, May 1979, pp. 73-81.

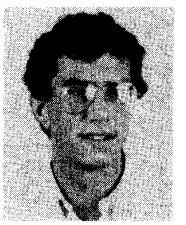
[35] C.-Y. Chin and K. Hwang, "Packet-switching networks for multiprocessor and data-flow computers," in *Proc. 11th Symp. Computer Architecture*, June 1984, pp. 99-109.
[36] P. Bitar and A. Despain, "Multiprocessor cache synchronization: Issues, innovations, evolution," in *Proc. 13th Ann. Int. Symp. Computer Architecture*, June 1986, pp. 424-433.
[37] L. Philipson, B. Nilsson, and B. Breidegard, "Communication structure for a multiprocessor computer with distributed global memory," in *Proc. 10th Ann. Int. Symp. Computer Architecture*, June 1983, pp. 334-340.
[38] E. F. Gehringer, A. K. Jones, and Z. Z. Segall, "The Cm* testbed," *Computer*, pp. 40-53, Oct. 1982.
[39] C. Scheurich, "Access ordering and coherence in shared memory multiprocessors," Ph.D. dissertation, Dep. EE—Systems, Univ. Southern California, Los Angeles, Tech. Rep. CENG 89-19, May 1989.



Michel Dubois (S'79-M'81) received the Engineering degree from the Faculte Polytechnique de Mons in Belgium, the M.S. degree from the University of Minnesota, and the Ph.D. degree from Purdue University, West Lafayette, IN, all in electrical engineering.

He has been an Assistant Professor in the Department of Electrical Engineering of the University of Southern California, Los Angeles, 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.

Dr. Dubois is a member of the Association for Computing Machinery and the IEEE Computer Society.



Christoph Scheurich (S'85-M'88) received the B.S.E.E. degree from the University of the Pacific, Stockton, CA, in 1981, and the M.S. and Ph.D. degrees in computer engineering from the University of Southern California, Los Angeles, in 1985 and May 1989, respectively.

His interests lie in computer architecture, parallel processing, and the design and implementation of multiprocessor memory systems.

Dr. Scheurich is a member of the Association for Computing Machinery and the IEEE Computer Society.