To appear in the *Proceedings of the 1990 International Conference on Parallel Processing*

Implementing Sequential Consistency In Cache-Based Systems†

Sarita V. Adve
Mark D. Hill

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

ABSTRACT

A model for shared-memory systems commonly (and often implicitly) assumed by programmers is that of *sequential consistency*. For implementing sequential consistency in a cache-based system, it is widely believed that (1) implementing *strong ordering* is *sufficient* and (2) restricting a processor to one shared-memory reference at a time is *practically necessary*.

In this paper we show that both beliefs are false. First, we prove that (1) is false with a counter-example. Second, we argue that (2) is false by giving sufficient conditions and an implementation that allow a processor to have simultaneous incomplete shared-memory references. While we do not demonstrate that this implementation is superior, we do believe it is practical and worthy of consideration.

Keywords: shared-memory multiprocessors, sequential consistency, strong ordering, cache coherence.

1. Introduction

A model of memory for shared-memory MIMD multiprocessor systems commonly (and often implicitly) assumed by programmers is that of *sequential consistency*, formally defined by Lamport [Lam79] as follows:

> [Hardware 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.

Application of the definition requires a specific interpretation of the terms *operations* and *result*. We assume that *operations* refer to memory operations or accesses (e.g., reads and writes) and *result* refers to the union of the values returned by all the read operations in the execution and the final state of memory. With these assumptions, the above definition translates into the following two conditions: (1) all memory accesses appear to execute atomically in some total order, and (2) all memory accesses of each processor appear to execute in an order specified by its program.

Unlike uniprocessor systems, simple interlock logic within a processor is not sufficient to ensure sequential consistency in multiprocessors. Particularly, as potential for parallelism increases, the conditions for ensuring sequential consistency become quite complex, and impose several restrictions on the hardware [DSB86].

This paper is concerned with the hardware implementation of sequential consistency, specifically for cache-based shared memory multiprocessors. We first show that strong ordering, proposed by Dubois, Scheurich and Briggs [DSB86], is not equivalent to sequential consistency (Section 2). Next we introduce the second, common approach for implementing sequential consistency that requires processors to perform memory references ‘‘one-at-a-time’’ [BMW85, RuS84, ScD87], and illustrate with sufficient conditions and an implementation proposal that this one-at-a-time approach is not necessary in practice (Section 3).

2. Strong Ordering

Strong ordering was defined by Dubois, Scheurich and Briggs in [DSB86] as follows:

> In a multiprocessor system, storage accesses are strongly ordered if 1) accesses to global data by any one processor are initiated, issued and performed in program order, and if 2) at the time when a STORE

† The material presented here is based on research supported in part by the National Science Foundation's Presidential Young Investigator and Computer and Computation Research Programs under grants MIPS-8957278 and CCR-8902536, A. T. & T. Bell Laboratories, Digital Equipment Corporation, Texas Instruments, Cray Research and the graduate school at the University of Wisconsin--Madison.

> on global data by processor I is observed by processor K, all accesses to global data performed with respect to I before the issuing of the STORE must be performed with respect to K.[1]

It was claimed earlier that a system that is strongly ordered is also sequentially consistent [DSB86, ScD87]. However, Figure 1 shows an execution that is not precluded by strong ordering, but violates sequential consistency. Thus, implementing strong ordering is not sufficient for implementing sequential consistency (confirmed by Dubois and Scheurich [DuS89]).

Initially X = Y = 0 in processor caches

P_1	P_2	P_3
X = 1		
	x_2 = X	
	y_2 = Y	
		Y = 1
		x_3 = X

Result - $x_2 = 1, y_2 = 0, x_3 = 0$

Figure 1. Strong ordering ≠ sequential consistency. Consider a cache-based system with a general interconnect. X and Y are shared variables initially present in processor caches with value 0. x_2 and y_2 are local variables, possibly registers, belonging to P_2, and x_3 is local to P_3. Let all the accesses be performed in program order. It is possible for the write on X to be propagated to P_2 before P_3, and before the write on Y is propagated to P_2. Thus reads issued by P_2 can return a 1 for X and a 0 for Y, making it appear that X was written before Y. P_3 can still return a 0 for X, making it appear that Y was written before X. Therefore, there does not exist any total ordering of memory accesses for this execution and hence the system is not sequentially consistent. However, none of the sufficient conditions for strong ordering (or concurrent consistency) are violated.

Admittedly, it appears that the accesses in Figure 1 would never arise in ‘‘real’’ programs. Apparently for this reason, Scheurich introduces *concurrent consistency* for systems that are sequentially consistent ‘‘except for programs which explicitly test for sequential consistency or take access timings into consideration’’ [Sch89]. Without a formal characterization of these exceptions, however, it is difficult for hardware designers to determine whether an implementation is correct and what restrictions, if any, should be given to software. For this reason, we advocate that hardware designers build sequentially consistent memory systems directly, rather than using strong ordering or concurrent consistency as intermediate models.

1. Refer to [DSB86] for a precise definition of the terms *issued*, *initiated*, and *performed*.

3. Relaxing the One-at-a-time Approach

Most implementations of sequential consistency in shared memory systems require a processor to stall before issuing a memory access until all effects of the previous access are observed by all the components of the system. For cache-based systems where processors are connected to memory through a common bus, most of the cache-coherence protocols proposed in the literature satisfy this condition [ArB86, RuS84]. The RP3 is a cache-based system that consists of a general interconnection network but does not support the caching of shared variables in hardware [BMW85]. Sequential consistency is maintained by stalling on every request to memory until an acknowledgement is obtained, again satisfying the one-at-a-time condition.

For cache-based systems with general interconnects that allow shared variables to be cached, a cache-coherence protocol is not sufficient. Scheurich and Dubois first proposed a sufficient condition for ensuring sequential consistency in such systems [ScD87, Sch89]. The condition is satisfied if all processors issue their accesses in program order, and no access is issued by a processor until its previous access is *globally performed*. A write is said to be globally performed when its modification has been propagated to all processors so that future reads cannot return old values that existed before the write. A read is globally performed when the value to be returned is bound, and the write that wrote this value is globally performed.

Sequential consistency can be maintained, however, without requiring a processor to globally perform each reference before beginning the next. Consider a system with an invalidation-based cache coherence protocol. A write miss to a line in read-only state is globally performed only when all the read-only copies have been invalidated. Scheurich [Sch89] observes that sequential consistency will be maintained even if a processor begins its next reference as soon as the invalidations are buffered at all other processors, provided other processors process buffered invalidations before handling a cache miss. Such references are not globally performed, because processors with unprocessed invalidates can still read old values. However, a processor reading an old value has not communicated with other processors via a cache miss since the invalidation was posted. Intuitively, therefore, sequential consistency is preserved, because these reads could have been done before the invalidation.

Exploiting this intuition more formally, we next give a new set of sufficient conditions for implementing sequential consistency that do not require a processor to stall for a write miss to be globally performed (Section 3.1). We then describe an implementation of these conditions on a general cache-based system (Section 3.2) and qualitatively compare the new implementation to one based on the one-at-a-time approach (Section 3.3).

3.1. Sufficient Conditions for Sequential Consistency

A processor can be allowed to proceed after issuing a write even while the corresponding invalidations (or updates) are on their way to the other processors in the system if some additional precautions are taken. Particularly, while its previous write is pending, the effect of the subsequent accesses of a processor should not be made visible to any other processor in the system. Hence to other processors, it is as if these accesses were actually performed after the pending write was globally performed. Based on this notion, we give below a set of conditions that is sufficient for implementing sequential consistency. (Accesses below need only refer to memory operations on shared, writable locations. If accesses to read-only and private locations can be distinguished, they can be handled more aggressively.)

1. Accesses are issued in program order.

2. All processors observe writes to a given location in the same order.

3. An access cannot be issued by a processor if

   (a) a read previously issued by it is not globally performed or

   (b) a write previously issued by it has not modified some copy of the line it accessed.

4. Let processor P_i issue an access A for a line on which a write W issued by processor P_j is globally performed while some writes of P_j before W are not globally performed. Then,

   (a) if A is a write operation, it can be globally performed only after all writes issued by P_j before W are globally performed,

   (b) if A is a read operation, it can return the value written by W only after all writes issued by P_j before W are globally performed.

5. Let processor P_i issue a write W for a line on which a read R issued by processor P_j is globally performed while some writes of P_j before R are not globally performed. Then, W can be globally performed only after all writes issued by P_j before R are globally performed.

6. A read issued by a processor while some of its previous writes are not globally performed should return the last value written on *any* copy of the accessed line, where *last* is defined by the order ensured by condition 2.

Condition 1 above is required so that accesses occur in program order. Condition 2 is the requirement for cache coherence and ensures correct ordering between writes to the *same* location. Condition 3 states when a processor has to stall because of its own incomplete accesses. Thus, a processor cannot proceed after issuing a read until a return value is bound to the read and all other processors have observed this value. (This restriction is similar to that in [ScD87, Sch89] and we do not have any optimizations for it.) A processor cannot proceed after issuing a write until it has the permission to write the line requested. A violation of this condition could lead to deadlock or necessitate rollback.

Conditions 4 and 5 state when a processor can be stalled because of incomplete accesses of other processors. Condition 4 ensures that if a line l is written by processor P_j while it has previous writes pending, then a read on l by another processor can only return an older value of the line until it observes the pending write of P_j. Thus it appears to all processors as if the write to l actually occurred after the pending write. Condition 5 ensures that if a line l is read by a processor P_j while it has previous writes pending, no subsequent write to l can be globally performed until the pending writes have been observed by all processors. This ensures that after P_j does its read, no other processor can read a later value of the line until the pending write of P_j is observed by it. This ensures that the subsequent read of P_j also appears to occur after the pending write.

Condition 6 ensures that while previous writes are pending, a processor always procures the latest copy of any line. Along with condition 3b, this avoids the possibility of deadlock or the need to roll back.

Based on the above reasoning, a proof of correctness of the conditions is given in [AdH89]. The proof is based on assigning unique hypothetical timestamps to each access in an execution. The timestamps are assigned so that an ordering of accesses in increasing order of their timestamps is consistent with the result of the execution and program order. This implies sequential consistency.

3.2. Implementation

This section discusses an implementation of the new conditions on a cache-coherent system with a general interconnect. A straightforward directory-based, write-back, invalidation cache-coherence protocol, similar to those discussed in [ASH88], is assumed. Our protocol allows, however, a line requested by a write to be forwarded to the requesting processor in parallel with the sending of its corresponding invalidations. On receipt of an invalidation, a cache is required to return an acknowledgement message to the directory (or memory). When the directory (or memory) receives all the acknowledgements pertaining to a particular write, it is required to send its acknowledgement to the processor cache that issued the write. The implementation described below allows only one write miss of any given processor to be outstanding. In [AdH89], we describe how more outstanding misses might be accommodated.

Condition 1 of the algorithm is ensured directly by requiring a processor to issue accesses in program order. Condition 2 is satisfied due to the cache-coherence protocol. For condition 3a, it is sufficient to stall a processor on a read until the requested line is procured. Satisfying the remaining conditions discussed below will not allow the line to be procured until the read is globally performed. For condition 3b, a processor is made to wait on a write until it obtains a *read-write* copy of the line in its cache and updates it.

For conditions 4 and 5, a one-bit counter is associated with every processor and a *reserve* bit is associated with every cache line. The counter is set when a line originally in *read-only* state is returned in response to a write request. It is reset on the receipt of the acknowledgement from memory. Thus, a counter that is set implies that a write issued by the corresponding processor has its acknowledgement outstanding, and hence is not globally performed. If a read or a write to a cache line is globally performed while the counter is set, then the reserve bit of the line is set. In addition, when a write returns a line which was originally read-only, its reserve bit is also set. All reserve bits are reset as soon as the counter is reset.

A reserved line written with the counter set is the only valid copy of the line in the system, and hence a request to this line is routed to the processor holding it reserved. Similarly, a write to a reserved line read with the counter set will result in an invalidation request reaching the processor with the reserved line. These read, write and invalidation requests are not serviced as long as the line is reserved which is until the processor counter is reset, i.e., until all previous writes of the processor are globally performed[2]. Now conditions 4 and 5 can be met if it is ensured that a line with its reserve bit set, is never flushed out of a processor cache. A processor that requires such a flush is made to stall until the pending write is globally performed. To ensure condition 6, while the counter is set, a read access is considered a hit only if the line accessed is held in read-write state in the processor cache. All other read accesses are sent to the directory and the processor stalls until they are serviced.

2. This might be accomplished by maintaining a queue of stalled requests to be serviced when the counter reads zero, or a negative acknowledgement may be sent to the processor that issued the request, asking it to try again.

3.3. A qualitative analysis

Figure 2 illustrates how the implementation described in the previous section can perform better than one based on the one-at-a-time approach for a general cache-based system. The one-at-a-time approach requires a processor to stall on a write miss, until the line requested is procured by its cache, *and* until all other copies of the line in the system are invalidated (indicated by an acknowledgement). In the new implementation, a processor is allowed to proceed on a write miss as soon as the requested line is procured by its cache. Subsequent accesses to lines held in read-write state in the processor cache are immediately serviced from the cache. A read to a line that is held in read-only state is also serviced, albeit from memory. Thus, the new implementation can perform better than one based on the one-at-a-time approach by allowing more memory accesses to be overlapped.

For applications involving a lot of sharing, however, it is possible for accesses to be blocked often due to outstanding writes of other processors. In the case a blocked access belongs to the critical path of the computation, performance could be adversely affected.

In any case, the one-at-a-time approach is not necessary in practice, and the new implementation cannot be rejected outright since its additional cost is small -- only a one bit counter per processor, a reserve bit per cache line and a mechanism for a processor to stall requests from other processors directed to it.

4. Conclusions

A memory model for shared-memory multiprocessor systems most commonly assumed by programmers is that of sequential consistency. Strong ordering has been believed to be a sufficient condition for implementing sequential consistency. We have shown with a counter-example that this is not true. As a necessary condition for a practical implementation of sequential consistency, it has been widely believed that a processor should not be allowed to issue an access until all effects of its previous access are observed. We have shown that this belief is also false by giving a set of sufficient con-

One-at-a-time	New Implementation
Write X; assume miss.	Write X; assume miss.
STALL.	STALL.
Block for X returned.	Block for X returned.
STALL.	Do additional reads and writes to blocks cached in read-write state.
STALL.	For read miss or read hit to block in read-only state, stall until block can be re-acquired from memory.
Acknowledgement for X returns.	Acknowledgement for X returns.
Perform accesses already done by new implementation.	Continue.
Continue.	

Figure 2. Qualitative analysis of new implementation.

ditions and an implementation that allow a processor to proceed after a write miss even if the write is not propagated to other processors.

Although our new conditions are less strict than the one-at-a-time approach, they still impose several restrictions on hardware. For higher performance, software support can be solicited [AdH90, DSB86, ShS88].

5. References

[AdH89] S. V. Adve and M. D. Hill, Weak Ordering - A New Definition And Some Implications, Computer Sciences Technical Report #902, University of Wisconsin, Madison, December 1989.

[AdH90] S. V. Adve and M. D. Hill, Weak Ordering - A New Definition, *To appear in the 17th Annual International Symposium on Computer Architecture*, May 1990.

[ASH88] A. Agarwal, R. Simoni, M. Horowitz and J. Hennessy, An Evaluation of Directory Schemes for Cache Coherence, *Proc. 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, June 1988, 280-289.

[ArB86] J. Archibald and J. Baer, Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, *ACM Transactions on Computer Systems 4*, 4 (November 1986), 273-298.

[BMW85] W. C. Brantley, K. P. McAuliffe and J. Weiss, RP3 Process-Memory Element, *International Conference on Parallel Processing*, August 1985, 772-781.

[DSB86] M. Dubois, C. Scheurich and F. A. Briggs, Memory Access Buffering in Multiprocessors, *Proc. Thirteenth Annual International Symposium on Computer Architecture 14*, 2 (June 1986), 434-442.

[DuS89] M. Dubois and C. Scheurich, Private Communication, November 1989.

[Lam79] L. Lamport, How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers C-28*, 9 (September 1979), 690-691.

[RuS84] L. Rudolph and Z. Segall, Dynamic Decentralized Cache Schemes for MIMD Parallel Processors, *Proc. Eleventh International Symposium on Computer Architecture*, June 1984, 340-347.

[ScD87] C. Scheurich and M. Dubois, Correct Memory Operation of Cache-Based Multiprocessors, *Proc. Fourteenth Annual International Symposium on Computer Architecture*, Pittsburgh, PA, June 1987, 234-243.

[Sch89] C. E. Scheurich, Access Ordering and Coherence in Shared Memory Multiprocessors, Ph.D. Thesis, Department of Computer Engineering, Technical Report CENG 89-19, University of Southern California, May 1989.

[ShS88] D. Shasha and M. Snir, Efficient and Correct Execution of Parallel Programs that Share Memory, *ACM Trans. on Programming Languages and Systems 10*, 2 (April 1988), 282-312.