DELAYED CONSISTENCY PROTOCOLS

Michel Dubois

USC Technical Report No. CENG 90-21

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-0781, USA
(213)743-8080
dubois@priam.usc.edu

July 25, 1990

[0]This research is supported by NSF Research Grant No. CCR-8709997.

DELAYED CONSISTENCY PROTOCOLS

Abstract

A new class of cache consistency protocols is introduced. The basic idea is that, in some cases, coherence need only be enforced at synchronization points, and therefore coherence enforcement can be delayed. A delayed consistency protocol can be designed from any on-the-fly (non-delayed) protocol. In this report, we will develop a write-invalidate delayed protocol. The main feature of the protocol is that the effect of out-going Stores and in-coming Invalidations are delayed until the next synchronization point in each processor. Moreover, block updates at memory are done partially.

The main gain is in the reduction of the number of invalidations due to block sharing and a corresponding increase in the hit ratio, especially when the number of processors increases. A second potential gain is that coherence actions such as invalidations are totally asynchronous with the processor, and can be propagated at times selected to optimize performance.

We present some results from a performance comparison of the delayed protocol with the corresponding on-the-fly consistency protocol through execution-driven simulations of two parallel algorithms; they show that there are significant gains to be obtained on the hit rate and on the number of invalidations, in some cases, by delaying consistency.

The protocols are not only applicable to cache-based systems but would be most useful for implementing shared memory in distributed systems.

1. Introduction

The design of shared memory systems that can scale up for large number of processors is a current topic of active research. There are two approaches. In one approach, private caches are associated with each processor and coherence is maintained among caches in hardware with possible compiler assistance. The second approach consists in emulating shared memory on a distributed system. The techniques presented in this report are applicable to both cases, but here we concentrate on cache coherence.

It has been argued that cache coherence and synchronization are related [DSB88]. The global ordering of shared memory accesses defines the concurrency model of a multiprocessor architecture. The strongest model is sequential consistency defined by Lamport [Lam79]. Sequential consistency constrains the timing of shared memory accesses and coherence activity. For example, sequential consistency is enforced if all shared memory accesses are globally performed in program order; a shared memory access in a processor cannot be issued before the previous shared memory access has been globally performed. When a Store is performed globally, all copies of blocks with the same data have been updated/invalidated [DuSc90]. Clearly, sequential consistency does not permit the delaying of coherence actions. We say that coherence must be maintained on the fly.

When sequential consistency is not enforced, memory accesses are weakly ordered [DSB86]. In a weakly ordered system, the programmer cannot make any assumption on the order in which accesses are propagated and observed by other processors, except at the execution of hardware-recognized synchronization primitives. Rather, synchronization points are "fences": before a process can "jump over" the fence, all its previous accesses must be globally performed; moreover no accesses following the fence in program order can be issued before the successful execution of the synchronization primitive. Clearly, in this model, coherence actions do not have to be taken on the fly; in principle, they can be delayed at most until the next synchronization point. Still, in all existing multiprocessors and in all proposals, coherence is enforced as soon as possible, on the fly. The major reason is that on-the-fly, write-invalidate protocols rely on the acquisition of unique or exclusive block copies for Stores; they cannot cope with the multiple, exclusive copies, which may result from delays in sending invalidations.

In delayed protocols, some coherence actions are deliberately delayed, and multiple, inconsistent block copies may exist in different processors at any one time. Therefore the hardware complexity is increased, but the pay off is a greater concurrency in accessing shared cache blocks. In-coming invalidations are buffered in the processor; similarly, when a cache must send invalidations to acquire a unique (exclusive) copy of a block on a Store, the propagation of these invalidations can occur asynchronously with processor execution: the processor is not blocked and the time to propagate invalidations may be selected optimally by the hardware. More importantly, delayed consistency increases concurrency by reducing the coherence activity on shared blocks due to false sharing.

2. False Sharing

False sharing is the sharing of cache blocks without actual sharing of data. In parallel applications, shared data structures are partitioned statically or dynamically and different processes work on different partitions of the structures. In general, partition boundaries do not coincide with cache block boundaries. As a result, cache blocks are shared while no data are actually shared. A more rigorous definition of false sharing can be found in [TLH90].

To demonstrate occurrences of false sharing we will show two simple examples. The first example is an algorithm with static partitioning of the data, the S.O.R. iterative algorithm. In this algorithm, an array (grid) of iterate components is updated iteratively by a linear combination of the iterate and its four neighbors in the 2-D grid. In the example of Figure 1.a, the grid has been partitioned among four processors. There are private iterate components, and shared iterate components, as indicated in the Figure. In a shared memory, organized as a linear address space, the array will be stored row-wise or column-wise. Assume that it is stored row-wise (i.e., first row 1, then row 2, and so on), and assume that the row size is not a multiple of the block size. Then false sharing occurs for blocks of type 2. False sharing (and true sharing) occurs also for blocks of type 1. Clearly, in this static case, the compiler could easily deal with the problem by allocating an integer number of blocks per row. However, for blocks of type 1, it will be difficult to achieve this in general, without wasting a lot of cache space or complicating drastically the addressing to contiguous array components. In real-life PDE algorithms, grid partitions are more complex than in the simple case of Figure 1.a, and the compiler will not always be able to reduce false sharing significantly. Some simple compiling techniques, which in some cases can reduce the effect of false sharing, are introduced in [TLH90]. It is obvious however that for a given problem size the false sharing problem becomes worse as the granularity of parallelism decreases, ie for larger number of processors.

The second example, in Figure 1.b, is an algorithm with dynamic partitioning of the shared data structure, the dynamic quicksort algorithm. In this algorithm, a processor acquires exclusive access to a subfile and splits it in two. False sharing occurs at the boundaries between consecutive subfiles. The boundary between two subfiles cannot be predicted at compile time.

False sharing results in non-optimum protocols. In the case of a write-invalidate protocol, such as the Illinois protocol [PaPa84], more invalidations are sent than strictly needed by the parallel application and its data-sharing requirements. Invalidations create traffic, and delays in the processor issuing them; moreover they increase the miss rate, because an invalidated block must be reloaded if it is accessed again. If the protocol enforces coherence on the fly, then a block can literally "ping-pong" several times between two processors, even if they reference different data elements in the block. For a given number of processors, the effect of the block size is very similar to the effect of the block size in uniprocessor systems, but for different reasons. As the block size increases, the miss rate curve first goes down because of spatial locality, then it bottoms out and goes up. This behavior is observed even in caches of infinite sizes. The miss rate curve bottoms out because of the increase in false sharing, which quickly offsets the gains due to spatial locality. Because of these effects, some researchers have advocated a block size of one and extensive prefetching for shared data [LYL87].

These effects are clear from the curves of Figure 2 and 3, which show the effect of false sharing on the total number of misses and invalidations sent for executions of the S.O.R. algorithm (Figure 2) and the quicksort (Figure 3). The results in these Figures were obtained through execution-driven simulations, following a technique introduced in [DBPB86]. In these simulations, all caches have infinite sizes and each simulated processor executes in turn until it accesses a shared data; at that point, the simulator simulates a different processor. This is done in a round-robin fashion. In Figure 2.a (resp. 2.b) we have plotted the total number of shared-data misses (resp. invalidations) for the S.O.R.

algorithm with four processors, a grid size of 128x128 and 100 iterations. Two curves are shown: in one curve it is assumed that all processors are working at the same speed and start each iteration at the same time (best case); in this case the effect of false sharing as the block size increases is very small. In the second curve (worst case), processor 2 is slightly slower so that it reaches a given block of type 2 at the same time as processor 1 (and similarly for processors 3 and 4); this could happen because of the order in which the processors reach and execute the barrier synchronization; the effect of false sharing is maximum here. This Figure demonstrates that

1) False sharing can have a large impact on the shared data miss rate for the S.O.R. algorithm, and that

2) Trace-driven simulation results can be very misleading to evaluate the effect of false sharing on the miss rate, because it is very dependent on the exact timing of accesses by each processor of the target multiprocessor [Bit90].

The plots for the quicksort are shown in Figure 3; the number of processors is varied from 1 to 32, the file to sort is made of 32K random integers drawn from a uniform distribution; each point is the average of the number of misses and invalidations for 10 files. For 32 processors, the miss rate curve bottoms out for block sizes of 8x4=32 bytes.

The situation is similar in write-broadcast protocols, such as the Firefly protocol [TSS88]. In these protocols sharing is detected dynamically and multiple copies of the same block can be modified at the same time by different processors, provided modifications are broadcast to all processors with a copy. Clearly, the update traffic should be limited to data elements that are actually shared; however, because of false sharing, a lot of redundant updates corresponding to different data elements in the same block will be propagated. Delayed consistency can be applied to both write-invalidate and write-broadcast protocols. In this report, we will develop a write-invalidate delayed protocol.

3. Weak Ordering

The class of concurrent programs for which delayed consistency is applicable has been defined in [AdHi90]. It is based on a weakly-ordered concurrency model called DRF (Data-Race Free). Programs assuming sequential consistency will not run correctly in general on machines with delayed consistency. In this model, processes must synchronize in such a way that no set of processes can ever access a shared variable at the same time unless all these accesses are Loads. Clearly, critical sections and semi-critical sections are allowed in such programs.

In the following, we call a *datom* the smallest unit of addressing in the machine. This is usually a byte (8 bits), but could also be a 32-bit word. Under DRF, a datom goes through different phases. There are Read/Write phases by a single process, and/or Read phases by multiple processes. To enforce DRF in asynchronous multiprocessors, these phases must be "framed" by explicit, hardware-recognized synchronizations.

When a process needs to enter a Read/Write phase (critical section) for a datom, it must first acquire a lock; then, at the end of the phase, it must release the lock. Critical sections are present in both algorithms. In the S.O.R. algorithm, each process executes a barrier synchronization between consecutive sweeps of the grid, and in each sweep, different iterates are updated exclusively by different processes. In the quicksort example, when a process fetches a new subfile descriptor, some form of hardware synchronization is assumed, so that updates to a given subfile are done exclusively by one process; similarly, when a process has completed the split of a subfile, it must execute some form of hardware-recognized synchronization.

Some datoms in the S.O.R. algorithm (the ones at the boundaries) can be read by multiple processors (two or three) in every other sweep of the algorithm. Again, each sweep is framed by barrier synchronizations.

4. Summary of Motivations

We summarize here the basic motivations behind write-invalidate delayed protocols.

• Read-Write sharing of blocks without actual sharing of datoms results in non-optimum protocols. More invalidations, and more misses than strictly needed reduce the efficiency of on-the-fly protocols.

• If we give up sequential consistency, then ownership is only required for explicit synchronization variables. Implementing ownership for all datoms is useless and costly. In particular, under DRF, the software ensures that one datom cannot be modified by one process if other processes can access the datom.

• In between two explicit synchronization points, datoms can be modified freely, because it is known that no other processor will attempt to read/write the datom. If a block is accessed by multiple processors at the same time and one access is a write, then we know that these accesses cannot be for the same datom in the block.

• In the delayed protocol, invalidations can be propagated asynchronously with processor execution, at times selected optimally.

• At hardware-recognized synchronization points however, updates of datoms must propagate, because the processor may be terminating its Read/Write phase for a given datom, allowing other processors to read it.

5. Maximum Delay Protocol

We describe in this Section a write-invalidate delayed protocol. In an on-the-fly protocol, updates on non-owned blocks result in the sending of invalidations; later, these invalidations are received and executed by individual processors. Each of these two phases can be delayed, ie the sending of invalidations, and the execution of those invalidations. The protocol derived in this Section assumes maximum delays for both sending and receiving invalidations. To maximize the delay, we also assume that at the hardware level, distinction can be made between acquiring a lock (*lock* operation) and releasing a lock (*unlock* operation).

The delayed protocol works in conjunction with two buffers per processor: one Send-Invalidation Buffer (SIB) to buffer out-going invalidations and one Receive-Invalidation Buffer (RIB) to buffer in-coming invalidations. The size of each of these buffers is assumed to be infinite (ie, they must have the same number of entries as the cache size in blockframes.) Moreover, a given entry in each buffer can be removed by accessing the buffer with the address (this feature is needed for replacements).

There is one dirty bit per datom in each blockframe. If a dirty bit is reset, it becomes set at the time the datom is modified in the cache. When a memory block is updated from a cache, only the datoms in the block with a set dirty bit are modified in the memory.

When a processor modifies a clean block, it does not need to acquire a unique copy, and furthermore, the modification does not have to be *visible* to other processors before the execution of the next *unlock* (at the latest). The reason is that the programmer intended it this way. Similarly, when a cache receives an invalidation for a block, the local copy can remain valid and accessible by the local processor until the execution of the next *lock* instruction; we say that the copy is *stale*. Note that a stale copy is only valid for the local processor, which can read and modify the block copy; the rest of the system considers it as invalid. In particular, in a directory based system such as the one described in [CeFe78], the presence bit for a stale copy is reset. Therefore, when we talk about the state of a block copy we have to distinguish between the *processor* point of view (state stored in the cache directory) and the *system* point of view (state stored in memory directories).

5.1 System Point of View

From the system point of view, a block copy can be in three states in any one cache: I (stale, invalid, or not in cache), O (Owner), or K (Keeper). If a cache is the *Owner* of the block, then the cache must deliver the copy on a miss by another processor; otherwise, if the cache has a valid copy but is not the owner, then the cache is a *Keeper* of the block. From the system perspective, if a cache is an *Owner* of a block, then the copy must be unique and the system must enforce this view. Later, we will see that from the processors' point of view, an Owned copy is not necessarily unique.

At the system level, the protocol is a conventional write-invalidate protocol. It will be effective for private data, and by by-passing all buffers, it will also work for accesses to synchronization variables. Addresses of synchronization variables must be distinguishable from addresses of other variables in each processor node (but not at the system level).

The following commands can be issued by one processor node to the memory system (remember that the words *invalid* or *valid* must be interpreted in the context of the system, not the individual caches):

• **ReqO** (Request Ownership): This command can be issued if the cache has no valid copy (case of a write miss) or the copy is in state K. If there are copies in other caches, they must be invalidated. If there is an Owned copy and it has been modified, then the modified copy with the dirty bits must be received by the cache issuing the command; the new copy inherits the dirty bits and is obtained by merging the local copy (if any) with the remote copy.

• **ReqC** (Request Copy): This command can be issued if the copy is I (case of a read miss). If no valid copy exists in the system, then the cache issuing the command receives the copy from memory and becomes Owner. Otherwise, if there are valid copies, then the cache receives a K-copy from the memory; in case there was an O-copy, this copy must first update memory if it is modified and becomes a K-copy.

• **Inv** (Invalidate copies): This command is issued whenever a non-owned copy has been modified, and the modifications must be propagated (replacement or removal from SIB). All copies must be invalidated, and if there is an O-copy, this copy must update memory in case it was modified.

• **UpdM** (Update memory): This command is issued on a replacement or on a removal from SIB in conjunction with the Inv command. The update is partial, based on the settings of the dirty bits.

The state transition diagram is shown in Figure 6. In this Figure, i is the local processor, and j is a different processor.

5.2 Processor Point of View

The possible states of a block copy in a cache are

1) **I X C:** Invalid, Clean,

2) **I X M:** Invalid, Modified,

3) **V K C:** Valid, Keeper, Clean,

4) **V K M:** Valid, Keeper, Modified,

5) **S K C:** Stale, Keeper, Clean,

6) **S K M:** Stale, Keeper, Modified,

7) **V O C:** Valid, Owner, Clean, and

8) **V O M:** Valid, Owner, Modified.

Therefore, from the processor point of view, there are three validity levels for a block copy: Valid, Invalid and Stale. *Stale* means that an invalidation is pending in the RIB. When a copy of a block is stale, part of the block is valid and part is invalid; however, the processor can still access the copy (Read and Write) for as long as it accesses the valid part (in practice until the next *lock*). Note that from the system point of view a stale copy is considered invalid.

A block is *Clean* if no dirty bits are set in the block; it is *Modified* otherwise. When the cache is a *Keeper*, the dirty bits are cleared when the block is loaded in cache and after memory is updated (on an *unlock*). When the cache is an *Owner*, dirty bits may be set when the block is first loaded in cache (if the copy is inherited from an Owned and Modified copy); in this case the block is *Modified* at the time it is loaded in cache.

In the following protocol, both the system and the processor states agree that a block copy is *Owned*. However, the system and processor states disagree in the case where the cache is the keeper of a stale copy (the processor "believes" that it is a *Keeper*, while the system "believes" that the copy is invalid).

From the previous discussion in Section 5.1, it appears that a cache may receive two commands from the memory system. They are:

• **Invalidate:** The block copy becomes *Stale* and if it was *Owned* and *Modified* then it must be forwarded to the memory system (with the dirty bits).

• **Release Ownership:** This command can be received only if the block is *Owned* by the cache. The cache must become a *Keeper* and if the block copy is *Modified* then it must be forwarded to the memory system (with the dirty bits).

Let's now examine the state transitions due to accesses made by the local processor.

• **Read hit.** No action.

• **Read Miss.** The processor sends a *ReqC* command to the memory system. The returned copy will be either in state K or O.

• **Write hit.** The dirty bit of each modified datom is set to 1. If the block is *Modified*, then no further action is needed. If the block is *Clean*, then it becomes *Modified*, and if the cache is a *Keeper*, then an entry is inserted in the SIB.

• **Write Miss.** The cache issues a *ReqO* command. The returned copy is *Owned*; it is *Clean* or *Modified* depending on the source of the copy.

• **Replacement.** If the block is *Modified*, then the cache issues commands *Inv&UpdM* to the memory system. If there is an entry in the SIB or in the RIB, it must be removed. Note that replacements can be done through a Write-back buffer, just like in on-the-fly protocols.

• **Acquiring a lock** (*lock*). The RIB must be emptied, right AFTER the lock has been acquired. For each entry in the RIB the cache block is invalidated (ie, all *Stale* copies become *Invalid* from the processor point of view.)

• **Releasing a lock** (*unlock*). The SIB must be emptied right BEFORE releasing the lock. For each Stale entry in the SIB, an *Inv&UpdM* command is sent to the memory system. For each Valid entry in the SIB, a *ReqO* command is sent to the memory system.

When a block is *Stale* and *Modified*, a *lock* results in simple invalidation; the actual update of memory takes place later when an *unlock* is executed. This is why we have distinguished between *Invalid&Modified* and *Invalid&Clean* in the list of states. The reason behind this distinction is to facilitate the design of the RIB.

6. Implementation

There are major implementation problems with the protocol described in Section 5. First, infinite buffers are needed to accommodate the worst case situation. The major problem, is that, at replacement, it is very difficult to pull out of the SIB or the RIB the entry corresponding to a given blockframe, unless we are willing to implement fully associative buffers. Another problem is that at synchronization points, there will be a large number of invalidations sent to other processors; in the case of a barrier synchronization, the interconnection network may be swamped by invalidations; a large amount of time may be needed to empty the RIB and SIB. Therefore, it would seem more reasonable to spread these invalidations over a period of time. Finally, to take care of asynchronous algorithms, i.e. algorithms that potentially never synchronize, some form of periodical flushing of the buffers will have to take place anyway, in the absence of *lock* or *unlock* instructions. In the following, we present a practical implementation of the ideal protocol of Section 5. Since the protocol from the system point of view is unchanged, we only concentrate on the processor point of view. The two buffers RIB and SIB must be easily and efficiently implemented.

6.1 Implementation of the RIB

An infinite size RIB is easy to implement, in the above protocol. It can be done by associating with each blockframe a Stale bit (S-bit). When an invalidation is propagated to the cache, the S bit is set. When the S-bit is set, the blockframe is still accessible by the local processor, but for the rest of the system, the block has been invalidated in the cache.

Therefore, to implement the protocol described in Section 5, we propose to associate 4 bits with each blockframe: the S-bit (Stale bit), the I-bit (Invalid bit), the O-bit (Ownership bit), and the M-bit (Modified bit). The encoding of the different states of Section 5 should be obvious.

When the processor acquires a lock, the S-bits and I-bits are ORed into the I-bit, and the S-bit is cleared for all blockframes in the cache. Whenever a new block is loaded in a blockframe, the S-bit must be cleared. This simple mechanism works because in the protocol, no specific action has to be taken when a *Stale* block is invalidated (at *lock* time) besides setting the Invalid bit. To insert an invalidation in the RIB, the S-bit is set; to remove it, the S-bit is cleared.

Note that stale bits do not need to be set immediately, since they are only needed at a synchronization point. A small invalidation buffer can still be present, so that invalidations on non-owned copies are propagated to the cache at the most opportune time (eg, when the processor does not access the cache).

6.2 Implementation of the SIB

The SIB must be accessible in two ways. Most of the time it functions as a FIFO buffer: entries are entered at one end, and when the buffer is full, entries are removed at the other end. But, if there is a replacement in the cache, the entry must be removed from the buffer, and therefore, the buffer must be accessible fully associatively. The circuit to implement this function is feasible and is very similar to the circuit implementing LRU, shown in Figure 7, for two bits of address (this Figure is taken from [HwBr84, p. 117]).

To be infinite, the SIB would have to contain as many entries as the number of blockframes in the cache. This may be difficult for very large caches. Also, while an infinite SIB results in maximum delays, and therefore minimizes false sharing, it will create very non-uniform traffic in the interconnection.

Therefore, the protocol may be more efficient if the size of the SIB is less than the size of the cache. When the SIB is full, an entry must be removed from the head of the FIFO buffer. To extend the protocol, we must indicate the actions to take when an entry is removed from the SIB. Assume first that the SIB has at least size one. At the time when an entry is removed from the SIB the same actions must be taken as on an *unlock* (which is the only time when an entry is removed from the SIB in the case of an infinite size buffer). Therefore, the modification of the protocol is trivial in this case. The only modification to Section 5.2 is that "Releasing a lock" should be replaced with "Removing an entry from SIB", with the understanding that the SIB must be emptied on an *unlock*.

The protocol is specified in the state transition table corresponding to a Mealy Finite State Machine shown in Figure 8.

To verify the correctness of a protocol like the one in Figure 8 is in general very complex: one has to prove that a Load of a datom always returns the value defined by the latest Store on the same datom, given that the system is weakly ordered and the concurrency model is DRF. In this model, modifications of datoms by a processor do not need to be *visible* to the rest of the system until the next *unlock*; as long as a datom modification is local and has not propagated (it is in the SIB), the system ignores it, as if it had not happened yet. Similarly, when an invalidation is received for a block by a processor, the copy becomes stale: the local processor can still access it up until the next *lock* operation, while, for the rest of the system, the copy has been invalidated. From the perspective of the protocol among processors, a stale copy is an invalidated copy and a modification of a block has not happened for as long as it is buffered in the SIB. The ownership bit has been introduced to deal efficiently with private blocks, and the protocol has been optimized to avoid extra overhead in the case of private blocks. The protocol also makes sure that either the memory or a cache have the latest copy of an overall block (the notion of "latest" here takes into account the rules for visibility of updates as described above).

Extensive testing of the protocol has been done by generating random sequences of locks, unlocks, reads and writes so that, in the sequence, the succession of locks/unlocks may not deadlock, and furthermore the reads and writes to shared writable data obey the weakly ordered model. These sequences were applied to the simulators of the delayed protocol and of the corresponding on-the-fly protocol obtained by removing all buffers. In both cases, the state of the shared variables in memory had to be the same after applying each sequence to both simulators. This simple technique allowed us to detect several subtle errors in the protocol.

6.3 No SIB

At the limit, the SIB could be of size zero; in this case, invalidations are sent immediately when they are produced, as in the on-the-fly protocol. Note that false sharing is still reduced in this case, because invalidations are still delayed in the RIB of the destination processors. In this case, no action is taken on *unlocks*, but a write can only be executed on a unique (owned) copy. Figure 9 displays the new protocol.

7. Effect on False Sharing

The effectiveness of the protocol of Section 5 is demonstrated in Figure 10, for random sequences of reads and writes of a given block, as well as locks and unlocks by 3 processors. The changes of states are indicated on the Figure. Only four misses occur in the maximum delay protocol. The reader can verify that the on-the-fly protocol of Section 5.1 would cause misses on practically every access.

We have run the same examples as in Figure 2 and 3, and the improvement due to the protocol of Figure 6 is shown in Figures 4 and 5. The on-the-fly protocol used for Figure 2 and 3 is the protocol of Section 5.1 (System point of view) with no delays in the processors. In the miss and invalidation curves for the delayed protocol, the increases due to false sharing have practically disappeared. For a given problem size, the delayed protocol reduces the number of false sharing misses as the number of processor increases. For these examples, it appears that delayed protocols are more scalable than on-the-fly consistency protocols. At this point, we have no result for the protocol with an SIB of finite size.

8. Hardware Complexity

The cost of implementing delayed consistency is higher than for on-the-fly consistency. It includes:

- One additional dirty bit per datom and one stale bit per block,
- An efficient circuit to OR the Stale bits and the Invalid bits of all blockframes,
- An SIB buffer accessible associatively as well as FIFO, and
- A circuit on the processor board to distinguish between accesses to synchronization variables and to other variables.

There is no added complexity for the compiler, except for the separation of synchronization variables from other shared data in memory.

9. Conclusion

In this report, we have introduced the class of delayed consistency protocols in which the coherence overhead due to false sharing is reduced with respect to on-the-fly consistency protocols. Another advantage of delayed consistency is that the sending of invalidations and the receiving of invalidation can be done asynchronously with the local processor execution; the time to propagate those invalidations can be selected by the hardware to reduce conflicts and multiple invalidations pending in the SIB may be sent at the same time. This feature reduces the coherence penalty seen by the processor and is critical to good processor efficiency. From our preliminary evaluations (which only include the effect on false sharing), it appears that delayed protocols are more scalable than traditional protocols; this increased scalability is obtained with no assistance from the programmer and the compiler, and therefore it is particularly useful for general-purpose multiprocessors.

We have described a protocol for systems with infinite size invalidation buffers (both for send and receive). The hardware for such systems seems quite costly. However, the Receive Invalidation Buffer can be implemented at very low cost by adding a stale bit in each blockframe and by a special circuit to OR the invalid and stale bits after a successful *lock* operation is executed. The protocol has been extended to accommodate a Send-Invalidation Buffer of finite size. One interesting case is the case where the SIB is removed. The hardware complexity is then very low, and some advantages of the protocol with maximum delay are retained. This simplified implementation may be the way to go. However, at this time we have no evaluations to demonstrate the performance of this approach.

Synchronization variables must be stored in different regions of shared memory than other shared data. Accesses to the region of memory reserved for synchronization variables must by-pass all buffers, so that an on-the-fly protocol is enforced on these variables.

Much more work remains to be done. First of all the protocol may not be optimum and it may be possible to refine it once simulation results are available. Second, we need to investigate the effect of delays on write-broadcast protocols. Third, we need to study physical implementations of the protocol in bus-based systems, in directory-based systems (implemented by linked lists or by tables), and in systems with multi-level caches. But foremost, a more thorough evaluation of the delayed protocol, including finite caches, finite SIB, and interconnection traffic is needed to demonstrate the superiority of this new class of protocols.

10. Acknowledgements

I wish to acknowledge the work of Jin-Chin Wang, with whom I was able to discuss some of the problems associated with delayed consistency protocols, and who ran many simulations to debug the correctness and performance of the protocols; he detected several subtle errors in the protocol and without him, this work would have been impossible. Andy Glew gave me the idea of the Stale-bit to implement the RIB. Finally, several discussions with Philip Bitar were very useful. Philip coined the word "datom".

11. References

[DSB88] M. Dubois, C. Scheurich and F.A. Briggs,"Synchronization, Coherence and Ordering of Events in Multiprocessors," *IEEE Computer,* Vol. 21, No 2, pp.9-21, Feb. 1988.

[Lam79] L. Lamport,"How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers,* Vol. C-28, No.9, pp.690-691, Sept. 1979.

[DSB86] M. Dubois, C. Scheurich, and F.A. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. of the 13th Int. Symp. on Computer Architecture,* pp. 434-442, 1986.

[DuSc90] M. Dubois and C.Scheurich, "Memory Access Dependencies in Shared Memory Multiprocessors," *IEEE Transactions on Software Eng.,* 16(6), pp. 660-674, June 1990.

[TLH90] J. Torrellas, M.S. Lam, and J.L. Hennessy, "Measurement, Analysis, and Improvement of the Cache Behavior of Shared Data in Cache Coherent Multiprocessors," Technical Report No. CSL-TR-90-412, Stanford University, Feb. 1990.

[LYL87] R.L. Lee, P.C. Yew, and D.H. Lawrie, "Multiprocessor Cache Design Considerations," *Proc. of the 14th Int. Symp. on Computer Architecture,* pp. 253-262, 1987.

[PaPa84] M. Papamarcos and J. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proc. of the 11th Int. Symp. on Computer Architecture,* pp. 348-354, 1984.

[DBPB86] M. Dubois, F.A. Briggs, I. Patil, and M. Balakrishnan,"Trace-driven Simulations of Parallel and Distributed Algorithms in Multiprocessors," *Proc. Int. Conf. on Parallel Processing,* pp.909-916, Aug. 1976.

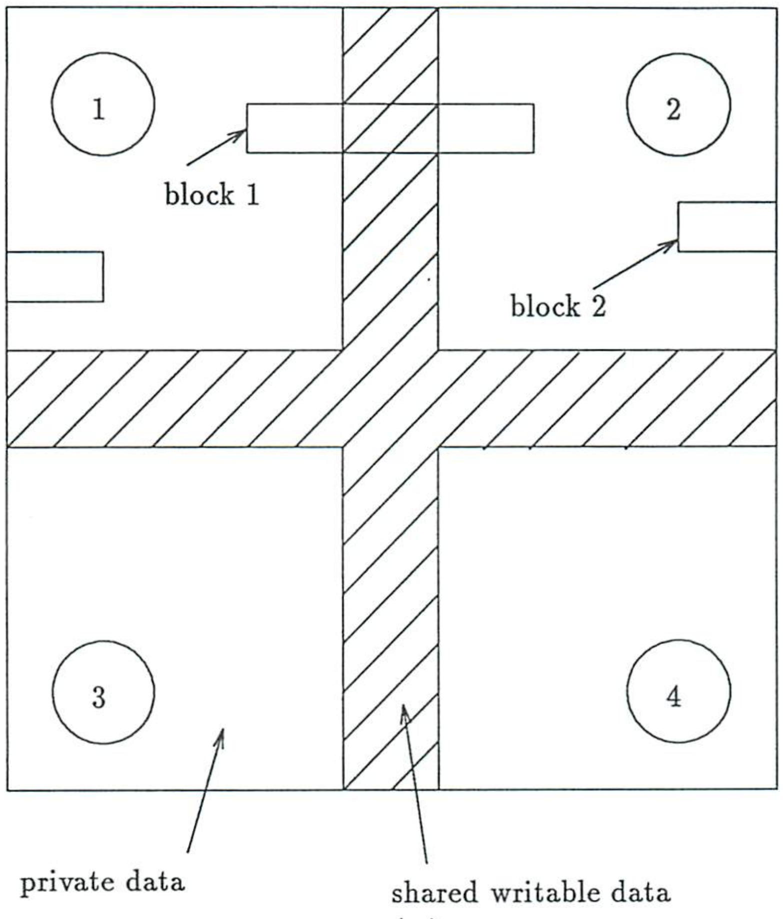
[Bit90] P. Bitar, "A Critique of Trace-Driven Simulation for Shared-Memory Multiprocessors," in *Cache and Interconnect Architectures in Multiprocessors,* M. Dubois and S. Thakkar, Kluwer Academic Publisher, July 1990.

[TSS88] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite, "Firefly: A Multiprocessor Workstation," *IEEE Trans. on Computers,* Vol. 37, No. 8, pp. 909-920, Aug.1988.

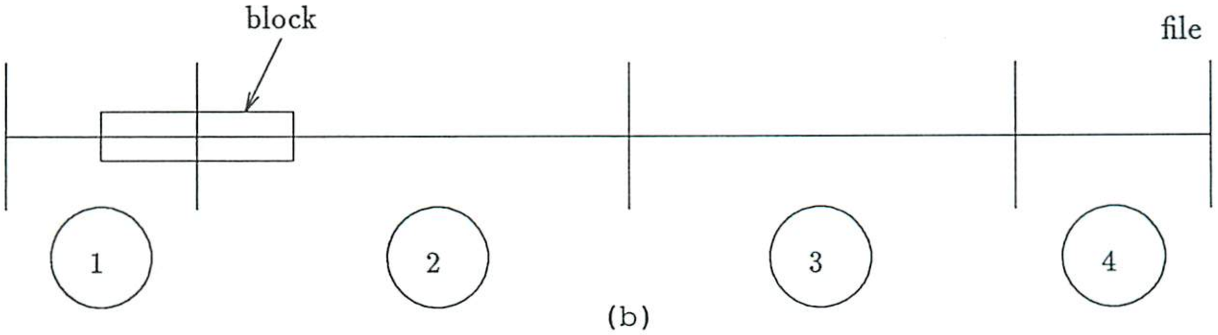
[AdHi90] S.V. Adve and M. D. Hill, "Weak Ordering-A New Definition," *Proc. of the 17th Int. Symp. on Computer Architecture,* pp.2-14, 1990.

[CeFe78] L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Computers,* Vol. C-27, No. 12, pp. 1112-1118, Dec. 1978.

[HwBr84] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing,* Mc Graw-Hill, 1984

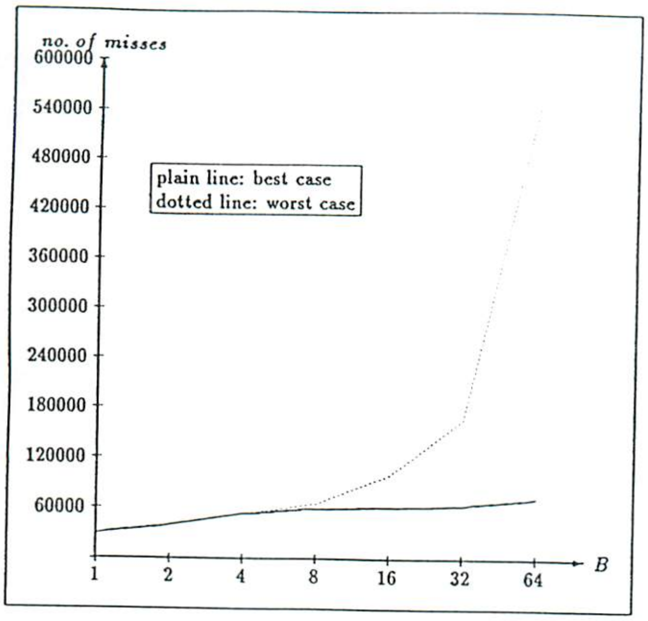


(a)

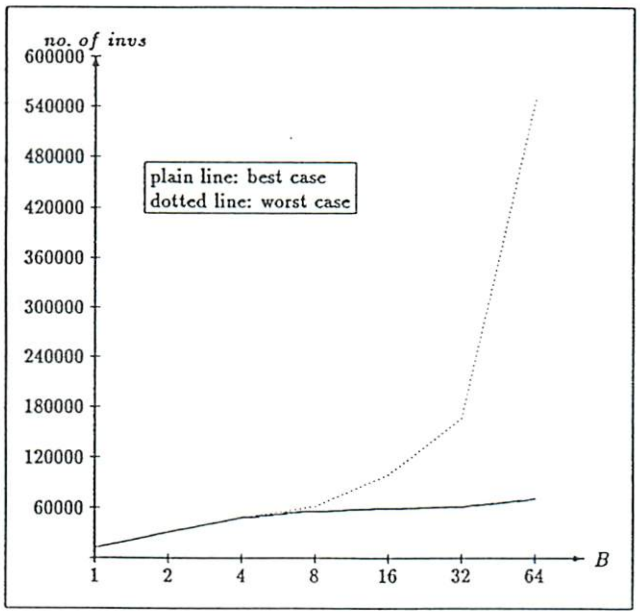


(b)

Figure 1. Illustration of false sharing in the S.O.R. (a) and Quicksort (b) algorithms.

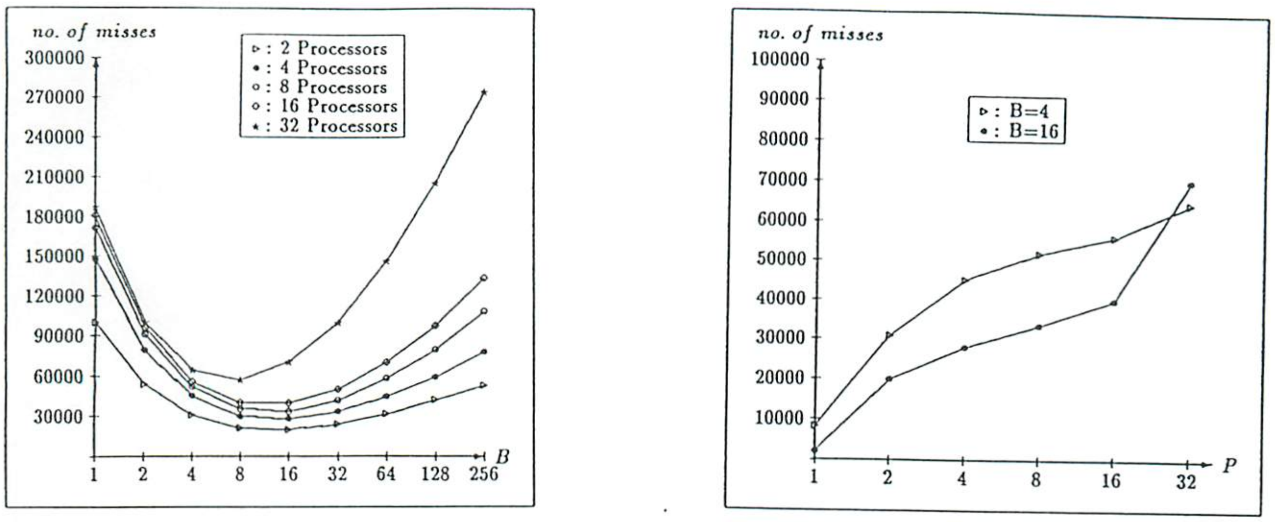


(a)

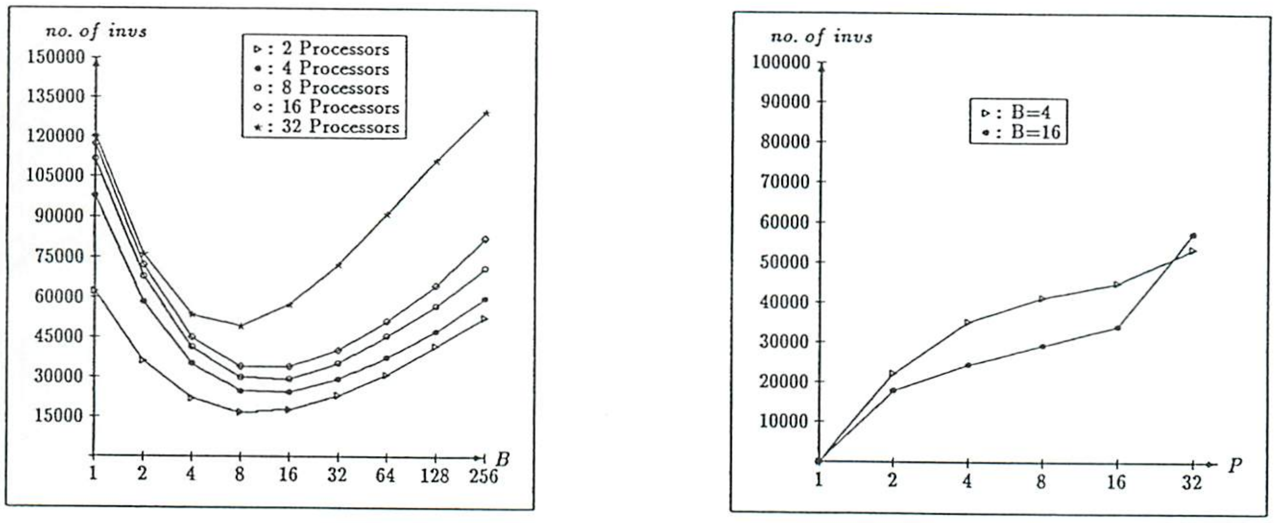


(b)

Figure 2. Number of misses (a) and invalidations (b) for the S.O.R. algorithm with on-the-fly protocol.

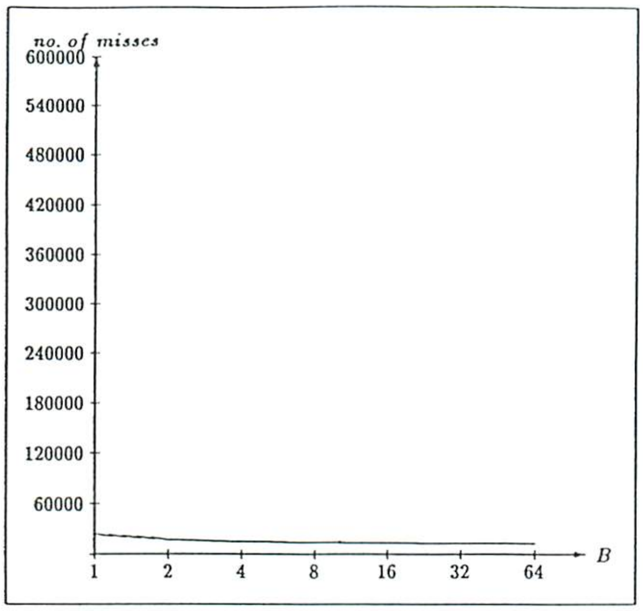


(a)

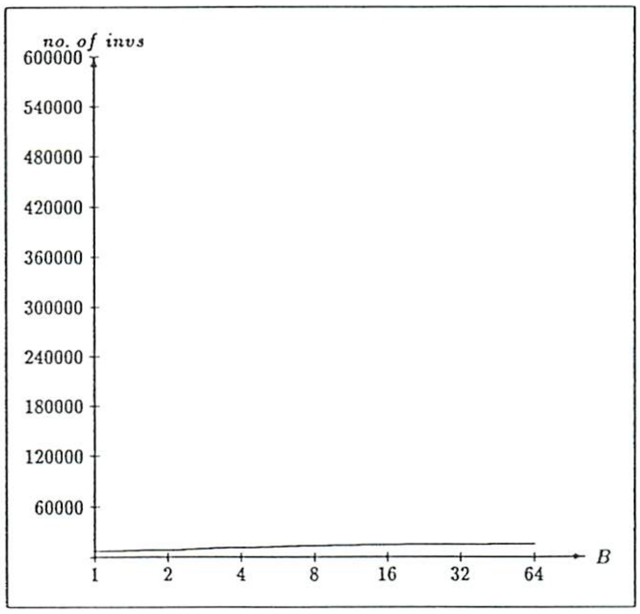


(b)

Figure 3. Number of misses (a) and invalidations (b) for the Quicksort algorithm with on-the-fly protocol.

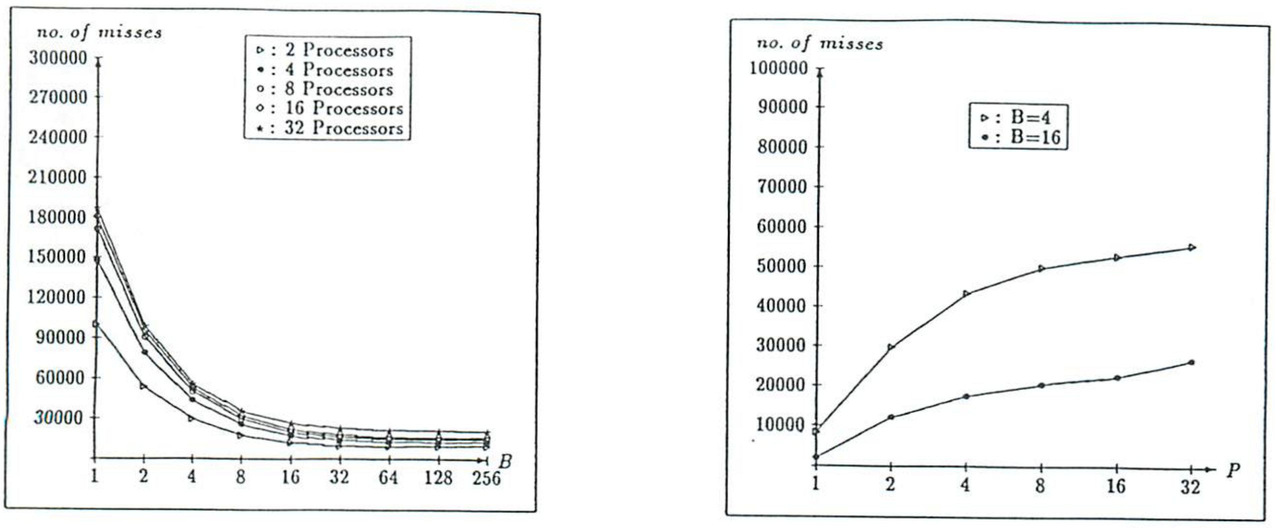


(a)

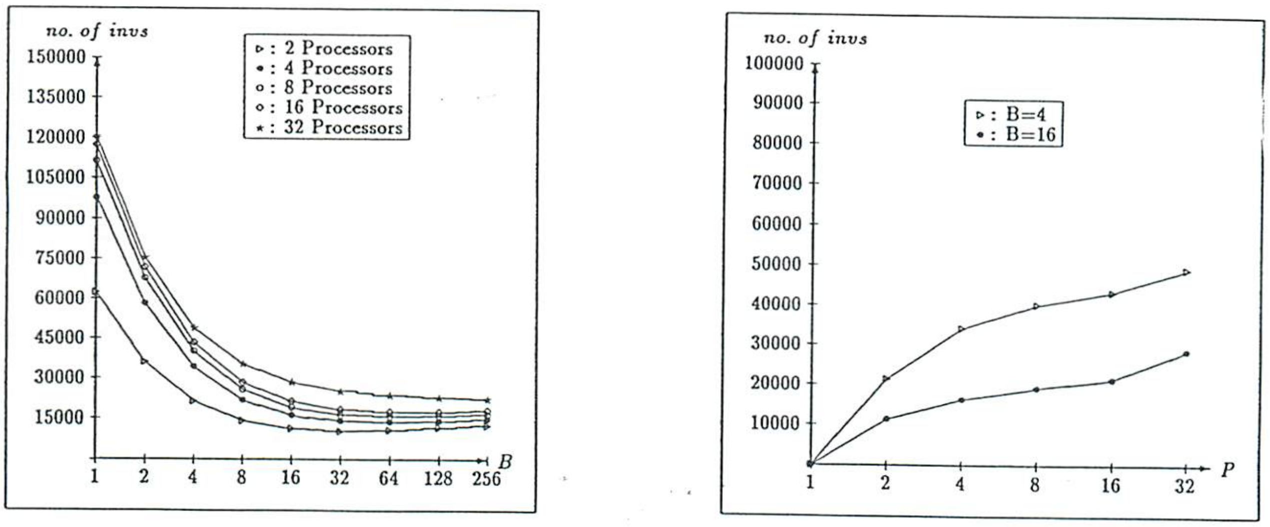


(b)

Figure 4. Number of misses (a) and invalidations (b) for the S.O.R. algorithm with delayed protocol.



(a)



(b)

Figure 5. Number of misses (a) and invalidations (b) for the Quicksort algorithm with delayed protocol.

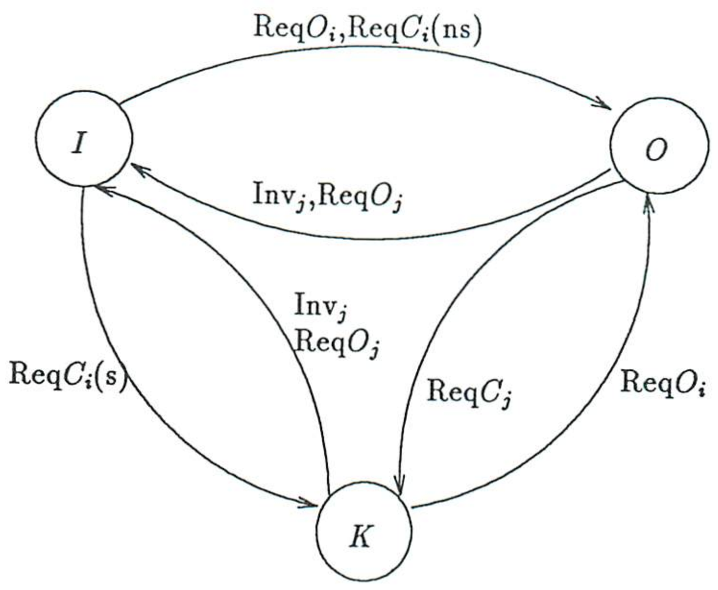
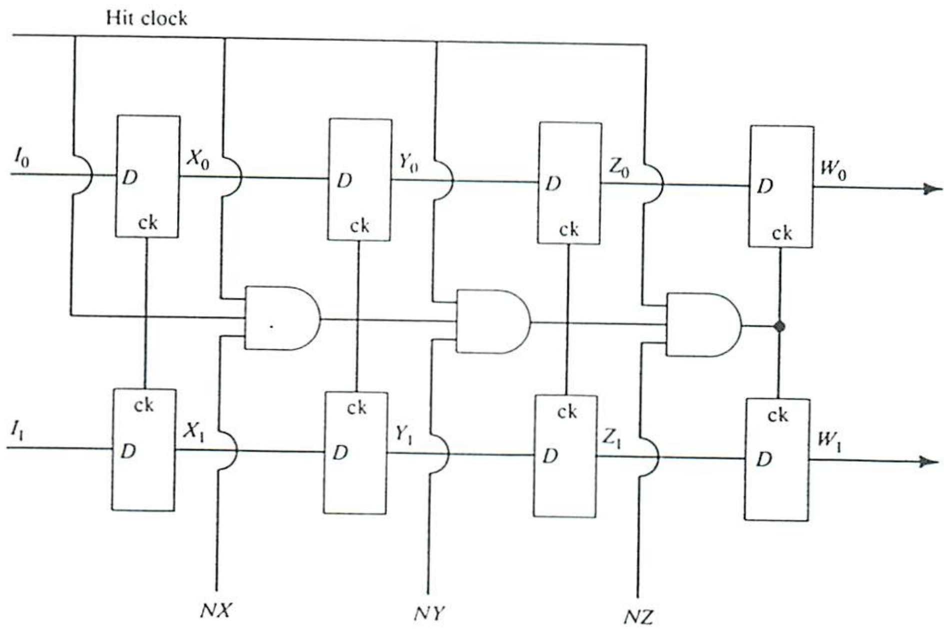


Figure 6: State diagram for the delayed consistency protocol (system viewpoint)



$NX = (X_1 \oplus I_1) \vee (X_0 \oplus I_0)$
$NY = (Y_1 \oplus I_1) \vee (Y_0 \oplus I_0)$
$NZ = (Z_1 \oplus I_1) \vee (Z_0 \oplus I_0)$

Figure 7: Circuit for LRU replacement (from [HwBr84])

	Read	Write	Lock	Rem SIB	Replace	INV$_j$	Rel O_j
$\overline{SI}OM$					Inv&UpdM	UpdM goto $S\overline{IOM}$	UpdM goto $\overline{SIOM}$
$\overline{SI}O\overline{M}$		goto $\overline{SI}OM$				goto $S\overline{IOM}$	goto $\overline{SIOM}$
$\overline{SIO}M$				Req$O$ goto $\overline{SIOM}$	Inv&UpdM	goto $S\overline{IO}M$	
$S\overline{IO}M$			goto $\overline{SIO}M$	Inv&UpdM goto $S\overline{IOM}$	Inv&UpdM		
$\overline{SIOM}$		InsSIB goto $\overline{SIO}M$				goto $S\overline{IOM}$	
$S\overline{IOM}$		InsSIB goto $S\overline{IO}M$	goto $\overline{SIOM}$				
$X\overline{IO}M$	Inv&UpdM ReqC goto $\overline{SIOM}$(ns) goto $\overline{SIO}M$(s)	UpdM Req$O$ goto $\overline{SIO}M$		Inv&UpdM	Inv&UpdM		
$X\overline{IOM}$ (NIC)	ReqC goto $\overline{SIOM}$(ns) goto $\overline{SIO}M$(s)	Req$O$ goto $\overline{SIO}M$					

Figure 8: State table for the delayed consistency protocol
(processor viewpoint)

S: Stale bit; I: Invalid bit; O: Ownership bit; M: Modified bit; X: 0 or 1
Rem SIB: Remove from SIB (must be done before an unlock instruction
Ins SIB: Insert in SIB
ns: not shared (no copies in other caches
s: shared (other copy exists)
Other commands are described in the text.

	Read	Write	Lock	Replace	INV$_j$	Rel O_j
$\overline{SI}OM$				UpdM goto $\overline{SIOM}$	UpdM goto $S\overline{IOM}$	UpdM goto $\overline{SIOM}$
$\overline{SI}O\overline{M}$		goto $\overline{SI}OM$			goto $S\overline{IOM}$	goto $\overline{SIOM}$
$\overline{SIO}M$		Req$O$ goto $\overline{SIO}M$			goto $S\overline{IO}M$	
$S\overline{IOM}$		Req$O$ goto $S\overline{IO}M$	goto $\overline{SIOM}$			
$XIXX$ (NIC)	ReqC goto $\overline{SIOM}$(ns) goto $\overline{SIO}M$(s)	Req$O$ goto $\overline{SIO}M$				

Figure 9: Simplified protocol for the case where there is no SIB

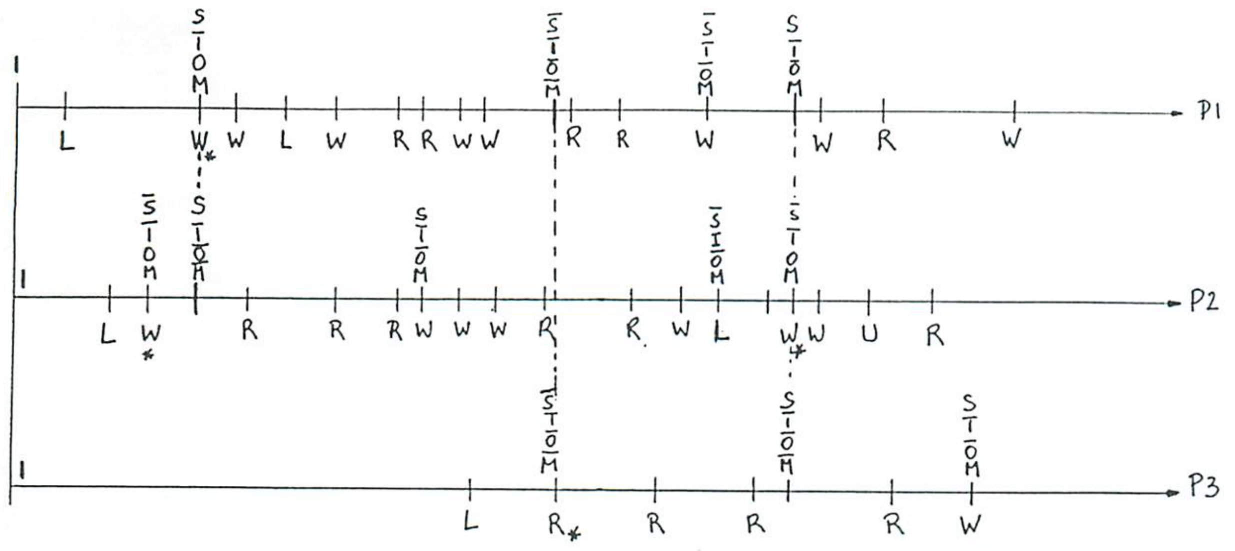


Figure 10. Illustration of the effectiveness of the delayed protocol for false sharing
R: Read; W: Write; L: Lock; U: Unlock
Misses are marked with an asterisk