# Delayed Consistency and Its Effects On The Miss Rate Of Parallel Programs

Michel Dubois, Jin Chin Wang, Luiz A. Barroso,
Kangwoo Lee, and Yung-Syau Chen

CEng Technical Report 91-14

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA. 90089-2562 (213) 740-4475

April 8, 1991

# DELAYED CONSISTENCY AND ITS EFFECTS ON THE MISS RATE OF PARALLEL PROGRAMS

Michel Dubois, Jin Chin Wang*, Luiz A. Barroso, Kangwoo Lee, and Yung-Syau Chen

Department of Electrical Engineering Systems
University of Southern California
Los Angeles, CA 90089-0781
(213) 740-4475
dubois@priam.usc.edu

* Tandem Computers Incorporated
M. P. Division
Austin, TX 78728
(512) 244-8397

April, 1991

## Abstract

In cache based multiprocessors a protocol must maintain coherence among replicated copies of shared writable data. In delayed consistency protocols, the effect of out-going and in-coming invalidations or updates are delayed. There are two potential advantages in delaying coherence: delayed coherence can reduce processor blocking time as well as the effects of false sharing.

In this paper, we introduce several implementations of delayed consistency for cache-based systems, in the framework of a weakly-ordered consistency model. A performance comparison of the delayed protocols with the corresponding On-the-Fly (non-delayed) consistency protocol is made, through execution-driven simulations of four parallel algorithms. The results show that in some cases significant reductions in the data miss rate of parallel programs can be obtained with just a small increase in the cost/complexity of the cache system.

## 1.0 Introduction

The design of shared memory multiprocessors that can scale up for large number of processors is a current topic of active research. It has been argued that technological constraints will ultimately put a limit on the processing rate of uniprocessors. Therefore, future systems will need to incorporate some form of parallelism. The shared-memory model appears at this point in time to be the choice parallel architecture for general-purpose computing.

These systems must efficiently support parallel multithreaded applications, as well as single thread processes, for three reasons. First, users need to run programs on a single processor. Second, some applications have very limited parallelism. Third, serial bottlenecks in the code set an upper limit on achievable speedups for large number of processors [Ahmd67]. It is therefore critical that single-thread processes run at peak efficiency.

There are two major problems in shared memory multiprocessors: shared memory bandwidth and shared-memory access latency. Both problems can be addressed by private caches associated with each processor. Most processor accesses are satisfied by the cache, at processor speed. However, coherence must be maintained among caches. Every time a miss occurs in a cache or every time a processor needs to modify a cache block present in several caches, the cache controller must access the global memory through an interconnection. During each such access, the processor and the cache are usually blocked. The time during which the processor is blocked will be referred to as a *penalty*. Penalties can be very high, especially for fast uniprocessors in large-scale multiprocessor configurations. Therefore, there is a need to reduce these penalties.

Another problem in shared memory multiprocessors is the problem of false sharing. False sharing is the sharing of cache blocks without actual sharing of data. It occurs because cache blocks contain more than one data item. 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. 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. 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 are propagated.

To reduce the effects of penalties and of false sharing we propose to delay consistency, under a weakly ordered model [AdHi90]. In general, in a cache-based system, a Store may cause an invalidation (write-invalidate protocol) or updates to other caches (write-broadcast protocol). To keep this discussion general, we will refer to both as *coherence updates*. In present systems, when a coherence update occurs, the cache controller is blocked while the update signals propagate and are acknowledged. If those update signals are buffered so that the cache is not blocked during the propagation of the signals, we say that coherence is *send delayed*. Note that this buffering is different from the usual Store buffer needed in systems with write-through caches (see Figure 1). Coherence can be also *receive delayed*. Namely, if a coherence update signal reaches a cache, the

effect of the update can be temporarily buffered.

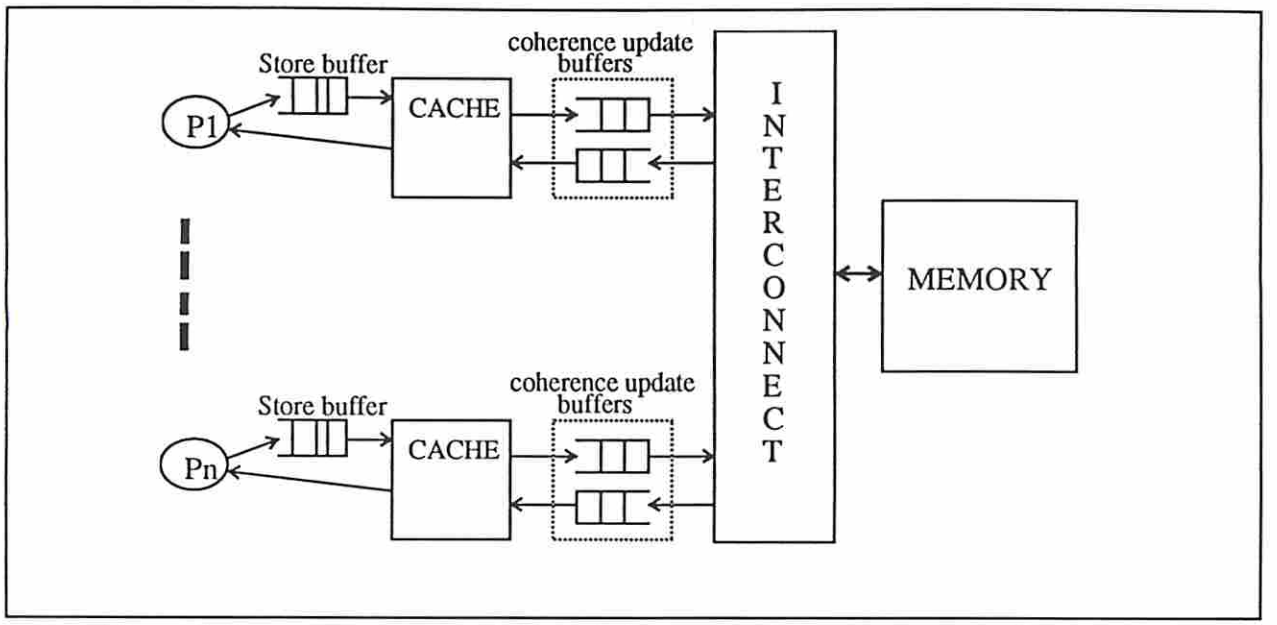


Figure 1: System structure with Store buffers and coherence update buffers

By delaying the sending of updates, update propagation can be overlapped with cache activity. In the protocol that we will describe, a processor or the cache never blocks on a Write hit, even if it hits on a non-unique, non-exclusive copy. Moreover, in a proposed variant of the protocol, the processor cache never blocks on a Write miss as well. Therefore, latency of Stores should be reduced. Also, the delaying of update propagation allows multiple processors to have dirty copies, and increases the concurrency of accesses to shared modifiable blocks.

The major contributions of this paper are the specifications of two delayed protocols derived from Censier and Feautrier's directory scheme [CeFe78], the hardware implementation details of both protocols, as well as simulation results showing the reduction in false sharing misses. In the following, we present some background material and discuss previous related work. In Section 3.0 we describe the basic protocols and the variants that we will consider, as well as implementation issues. Performance results derived using execution-driven simulations are shown in Section 4.0, followed by some concluding remarks.

## 2.0 Related Work and Background

A lot of work has been done on multiprocessor cache-coherence protocols. A good survey can be found in [Sten90]. In this section we will concentrate in the discussion of false sharing and latency tolerance.

### 2.1 False Sharing

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. This gives rise to false sharing transitions [TLH90], which create coherence or miss activity which would not happen if each cache block contained a single data item.

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 to solve Poisson's equation on a square domain [Youn71]. 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 2.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 such as block 2. False sharing (and true sharing) occurs also for blocks such as block 1.

The second example, in Figure 2.b, is an algorithm with dynamic partitioning of the shared data structure, the dynamic quicksort algorithm [Sedg80]. In this algorithm, a processor acquires exclusive access to a subfile, estimates a "pivot" element, and splits the subfile in two around the pivot. False sharing occurs at the boundaries between adjacent subfiles. The boundary between two subfiles cannot be predicted at compile time.

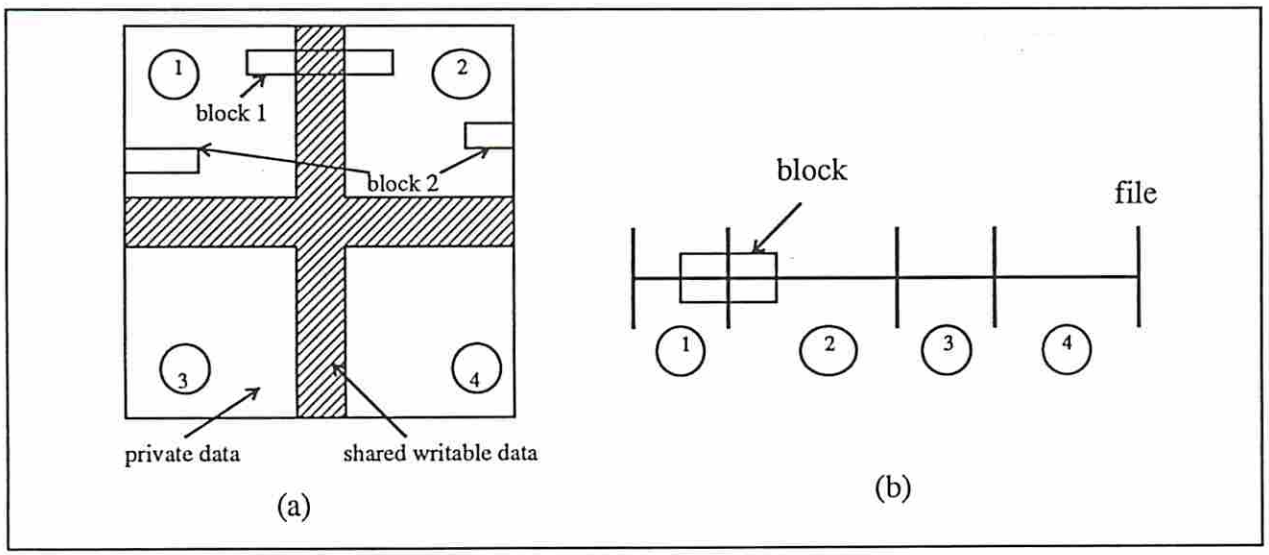


Figure 2: Illustration of false sharing in the S.O.R. (a) and Quicksort (b) algorithms

Because of false sharing a block may "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 decreases because of spatial locality, then the trend is reversed and the miss rate increases for larger block sizes. This behavior is observed even in caches of infinite sizes.This is due to the increase in false sharing, which quickly offsets the gains due to spatial locality.

These effects are clear from the curves of Figure 3, which show the effect of false sharing on the total number of misses for executions of the S.O.R. algorithm (Figure 3.a) and the quicksort (Figure 3.b). 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 or executes a synchronization primitive; at that point, the simulator simulates a different processor. This is done in a round-robin fashion. In Figure 3.a we have plotted the total number of shared-data misses for the S.O.R. algorithm with four processors, a grid size of 128x128 and 100 iterations (the data structure size corresponding to this grid is actually 130x130 because of boundary conditions). 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 - plain line); in this case the effect of false sharing as the block size increases is small. In the second curve (worst case - dotted line), processor 2 is slightly slower so that it reaches a block such as block 2 (i.e., a block at the end of a row) 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; here the effect of false sharing is maximum.

The plots for the quicksort are shown in Figure 3.b; the number of processors is varied from 2 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 for 10 independent files. For 32 processors, the miss rate curve bottoms out for block sizes of 8x4=32 bytes.

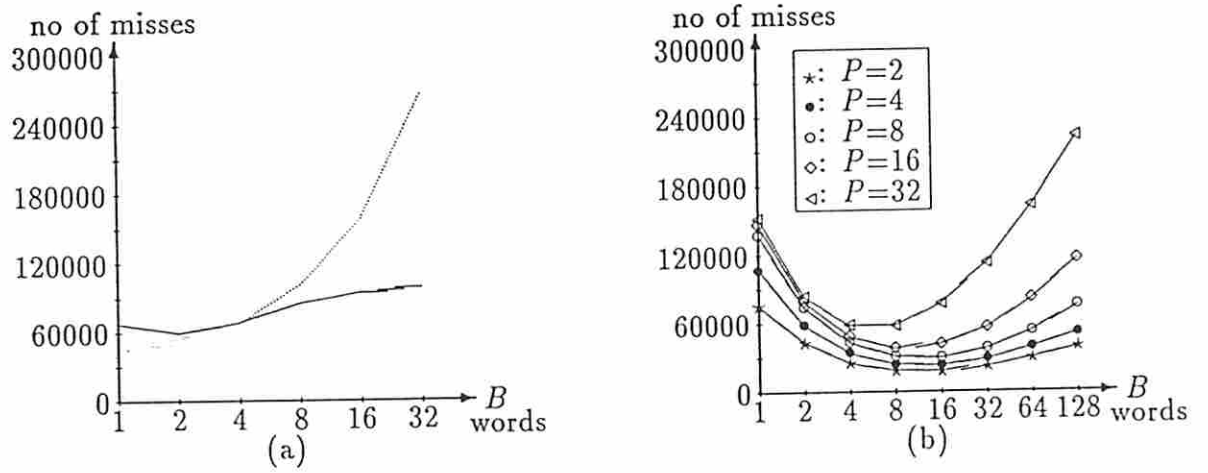


Figure 3: Total number of misses for S.O.R (a) and quicksort (b) algorithms.

In a system where we want to support efficiently both single and multiple thread programs, we are left with a *dilemma*. Namely, single thread programs benefit from bigger block sizes (for example, the block size in the IBM RS/6000 workstation is 128 bytes), while those bigger blocks are detrimental to the performance of parallel threads.

There have been several other proposed solutions to the problem of false sharing and the relatively low spatial locality of shared data accesses [LYL87][TLH90]. The first one consists in not caching shared writable data. This solution usually requires a T.L.B. (Translation Lookaside Buffer) to discriminate dynamically between cacheable and non-cacheable blocks. The problem with this solution is that entire data structures must be deemed non-cacheable if any item in the structure is shared and can be modified. Another proposition would be to use two data caches: one for shared data, and one for private data. These two caches could have different block sizes. Theoretically, caches could have two block sizes: one for shared data and one for private data, but the complexity of the implementation of such proposal has never been investigated. Bitar and Despain [BiDe86] have proposed to allocate one cache block per shared data item. This scheme would probably have to rely on the compiler to expand the shared data structure with dummy items, and compiling the code accordingly. This will result in significant waste of memory and cache.

The compiler and programmer can try to reduce false sharing through wise data placements. For example, in the S.O.R. algorithm above the compiler could try to allocate an integer number of blocks per row, if it *knows* that there is a potential false sharing problem; this would remove the problem for blocks such as block 2 in Figure 2. However, for blocks such as block 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. Real applications, while exhibiting the type of sharing present in the S.O.R. Poisson solver, are seldom as simple [Youn71]. Usually, the boundary calculations are complex, and the region is not square but is irregularly shaped and has internal cavities; sometimes the grid mesh size is different for different areas of the grid. Compilers may have problems dealing with false sharing in those cases, even for blocks such as block 2. Reduction of false sharing by the compiler or the programmer is also difficult in the case of dynamic data partitioning, such as in quicksort. In this case, the user would have to take into account the block size in the computation of the pivot, and this will result in complex algorithms, optimized at the source code level for one machine but not another. Some simple compiling techniques, which in some cases can reduce the effect of false sharing, are proposed in [TLH90] and evaluations based on 16 and 32 processor systems look encouraging.

## 2.2 Latency Tolerance

The miss penalty in current systems is of the order of 10 to 20 processor cycles. In the future we can expect miss penalties in excess of 100 cycles [MBBD91]. Latency-tolerance mechanisms can reduce these penalties. There are two approaches. The traditional approach, pioneered in the HEP multiprocessor [Kowa85], consists in having multiple resident contexts in each processor, and switching from one context to the next at the occurrence of an access to the global memory. This approach has been adopted in the TERA [ACCK90] and Alewife [ALKK90] multiprocessor projects. There are several problems with this approach. First, special processors must be designed for multiprocessors, with several processor-resident register sets. Second, switching from one thread to the next becomes more complex as uniprocessors achieve higher speed through increased internal concurrency. Third, the hit rate in the cache is reduced, unless the threads running concurrently on a processor are closely related. Finally, the mechanism does not speed up single-threads of code, and therefore is not good for single thread programs, programs with limited

parallelism, or programs with serial bottlenecks [Ahmd67].

The second approach advocates mechanisms which can hide penalties in the execution of a single thread. One such approach is non-blocking or *lockup-free* caches [Krof81][ScDu91] which reduce the penalty of cache misses. When a miss occurs in a non-blocking cache, the cache does not block the processor and services the miss concurrently with processor accesses. Several misses can be pending. Non-blocking caches are especially effective for second-level shared caches, for superscalar processors [SoFr91], and for programs in which in-cache prefetching can be done effectively[MoGu91]. Penalties due to Stores in systems with write-through caches can also be reduced effectively with a Store buffer between the processor and the cache. Delayed consistency is a latency tolerance mechanism that belongs to the second approach.

## 2.3 Other Related Work

In the IBM 3033, a multiprocessor with write-through caches, a mechanism called the *BIAS filter* was implemented [BLPS79]. In this buffer, invalidations with the same block address and coming from other processors are filtered before they reach the cache, in order to reduce the number of cache cycles needed by invalidations. In [DuSc90a], buffering of memory accesses at both the sending and the receiving processors was studied in the context of *sequential consistency* [Lam79] and weakly ordered protocols. For sequential consistency to hold it was required that buffered received invalidations have higher priority than accesses from local processors.

In [Sche89], the notion of *isolated caches* was introduced, and it was informally shown that received and buffered invalidations may have lower priority than local processor accesses, in the context of sequential consistency. In [ABM89], a theoretical model called *lazy caching* is developed, and it is proved that consistency can be send delayed and receive delayed in sequentially consistent systems. In this scheme, the sending of a coherence update can be delayed until the next Read (hit or miss) in the local processor. Moreover, the receiving of a coherence update in a cache may be delayed until the next miss in that cache. These conditions are very restrictive, especially for the sending of updates.

Under weak ordering [DuSc90a][AdHi90], protocols can be delayed further, and Distributed Shared Memory system proposals try to take advantage of this. The Plus [BiRa90] system has *delayed operations*. In *Munin*, a distributed shared memory system under development at Rice University [BCZ90a][BCZ90b], delayed consistency is advocated at the software level. Namely, updates by a thread of replicated objects are delayed until such a time that they can be detected by another thread, under a weakly ordered model of consistency called *loose coherence*. A *delayed update queue* is maintained for each object and updates on an object are propagated at synchronizations. In [BoHe90], a delayed consistency protocol implemented on pages at the software level is outlined. In these papers delayed consistency is linked to the problem of false sharing for the first time.

Finally, the DASH multiprocessor implements a form of weak ordering protocol based on *Release Consistency* [LLGG90] and attempts to overlap Stores with computation in a system with write-through caches, mostly by using a Store buffer. However, the second level cache locks out

the processor when a Store requires ownership. The DASH protocol is neither send delayed nor receive delayed, in the sense that we mean it here: the cache locks out the processor on a coherence update and there is no provision for delaying the reception of invalidations. In the DASH multiprocessor as long as the first level cache [LLGG90] hits on Read accesses while the second level cache gains ownership for a previous Store, consistency is in fact delayed. Some of the benefits of delayed consistency may be obtained through this two-level organization. Since the DASH protocol relies mostly on a Store buffer associated with the processor, the propagation of invalidations are time-critical and therefore results in complex deadlock-prone sequences of transfers between memory and caches.

## 3.0 Delayed Protocols

In delayed consistency protocols, advocated in this paper, the design is simplified by decoupling totally cache operation and propagation of coherence updates. The cache is never blocked on a Write hit, by associating a buffer for sending invalidations with each cache. Therefore, the timing of these invalidations is not critical as in the DASH protocol. Moreover, delayed consistency alleviates the performance degradations due to false sharing transitions in multiprocessors. These false sharing effects will be more pronounced in systems with low granularity of parallelism and large number of processors.

In a system with delayed consistency, 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 are not subject to delays, so that an On-the-Fly protocol is enforced on these variables. Therefore, the processor must have the ability to distinguish between synchronization variables and other variables. This is usually easy to do.

In the following we specify three protocols. The specification of each protocol is in three parts. The first part is a specification of block states in the caches and the system directory. The second part is a description of transactions between caches and memory to implement the protocol. The third part is the algorithm for the control of the cache.

## 3.1 On-the-Fly Protocol

This is the non-delayed protocol. Coherence actions are taken immediately and, during their propagation, the cache controller does not accept any request from the processor. This protocol was first introduced by Censier and Feautrier [CeFe78]. We reproduce the specification of this protocol for future reference in the paper.

**Block states**

*(a) Cache states.*

If a cache block is Valid, then the cache may be an Owner ( i.e. the copy is unique in the system and it is dirty) or a Keeper (i.e. there may be copies in other caches and all copies

are clean).

*(b) System Directory states.*

There is one Modified bit and P Presence bits (one per processor) per block in memory. A Presence bit is set only if the corresponding cache is an Owner or a Keeper of the block. If a cache is the Owner of the block (in which case only one P bit is set), then the Modified bit is also set.

**Memory commands**

*(a) Commands issued by the memory controller to the caches*

. Inv (Invalidate): the receiving cache is either a Keeper (the cache controller must invalidate its copy of the block), or the Owner (the cache controller must invalidate its copy of the block and send it to memory).

. UpdM (Update Memory): the receiving cache must be an Owner. The copy of the block is sent to memory and the cache becomes a Keeper of the block.

*(b) Commands issued by the cache to a memory controller*

. ReqO (Request Ownership): the memory controller sends an Inv command to all caches with a copy and the requesting cache becomes the Owner.

. ReqOC (Request Owner Copy): same as ReqO, but the memory copy of the block is also sent to the requesting cache.

. ReqKC (Request Keeper Copy) : if there is an Owner, the memory controller sends an UpdM command to the Owner. The memory copy of the block is then sent to the requesting cache (which becomes a Keeper).

. WB: the block is written back to memory.

**Cache algorithm**

For the various types of cache accesses, the cache controller takes the following actions (Figure 4).

. Read hit: no action is taken.

. Write hit: if the cache is Keeper, a ReqO command is sent to the memory controller, otherwise no action is taken.

. Read miss: a ReqKC command is sent to the memory controller.

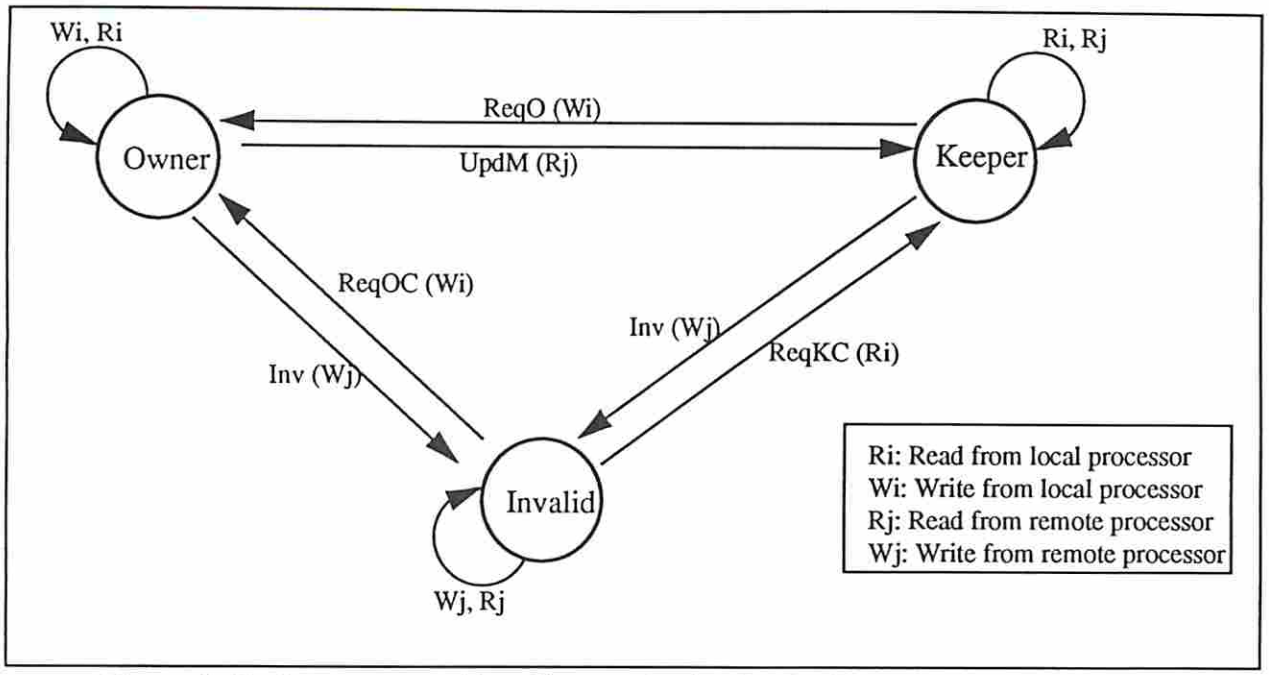


Figure 4: Cache states transition diagram for the On-the-Fly protocol

. Write miss: a ReqOC command is sent to the memory controller.

. Replacement: if the cache is the Owner, a WB command is issued to memory.

## 3.2 Receive Delayed Protocol

When an Inv signal is received by a cache, the invalidation does not need to reach the cache until the next Lock instruction executed by the local processor. The behavior is still correct because the programming model in weakly-ordered systems forbids accesses to a shared writable data outside a critical or semi-critical section [AdHi90]. Therefore, between the times when the Inv is received and the next Lock instruction is executed in the local processor, any Read to the block must be for a different word or byte in the block than the one modified. We say that the block copy is *Stale*. Right after the processor acquires a Lock, all the Stale blocks in the cache must be invalidated; otherwise, Stale blocks are treated the same way as Fresh (non-Stale) blocks in the cache, and the protocol is very similar to the above On-the-Fly protocol.

### Block states

*(a) Cache states.*

As for the On-the-Fly protocol, a cache may be a Keeper or the Owner of a block. However, in addition, a Valid block may also be Stale (i.e. Valid locally but Invalid in the system directory).

*(b) System Directory states.*

Same as for the On-the-Fly protocol.

**Memory commands**

*(a) Commands issued by the memory controller to the caches*

The commands are Inv and UpdM as in the On-the-Fly protocol. However, the cached copy becomes Stale instead of Invalid when an Inv command is received by a cache.

*(b) Commands issued by the cache to the memory controller*

The commands are ReqO, ReqOC, ReqKC, and WB as in the On-the-Fly protocol. The only difference is that the memory controller sends Inv and UpdM commands only to copies that are Fresh (non-Stale).

**Cache algorithm**

Similar to the On-the-Fly algorithm. However a Write hit on a Stale copy triggers a ReqOC command to memory and a block reload. Also, upon execution in the processor of a Lock instruction all Stale blocks become Invalid (see Figure 5).

## 3.3 Send-and-Receive Delayed Protocol

Receive Delayed protocols are effective at reducing the number of false sharing misses. They work well for blocks that are read by several processors, and modified by a small subset of these processors: the modifying processors experience false sharing transitions on each Write, but the reading processors do not (between the execution of two Lock instructions). Usually, however, between two Lock instructions, processors may Read and Write different words of the same block, or even sometimes only Write into the same block. If two Writes to the same block can occur simultaneously, then they must be for different parts of a block [AdHi90]. Therefore, such Writes can be executed without acquiring a unique copy. Writes executed on a non-unique copy must be propagated at the next execution of an Unlock instruction [LLGG90].

We have to keep track of all partial modifications of a non-owned block copy so that they can be written back to memory at the next Unlock instruction. Copies of these modifications must be kept in an Invalidation Send Buffer (ISB). At most this buffer must have as many entries as there are block frames in the cache. However, the optimum size of this buffer is probably much less than that. First of all, if the buffer is too large, Unlock instructions are very costly because a large number of memory updates must be propagated. Second, because of the locality of accesses to shared blocks, we can expect that accesses to a shared block by a given processor occur in runs or bursts. Therefore, we will observe diminishing returns in the miss rate curves as the buffer size is increased. The ISB should be a small, fully associative buffer, capable of containing a few blocks.

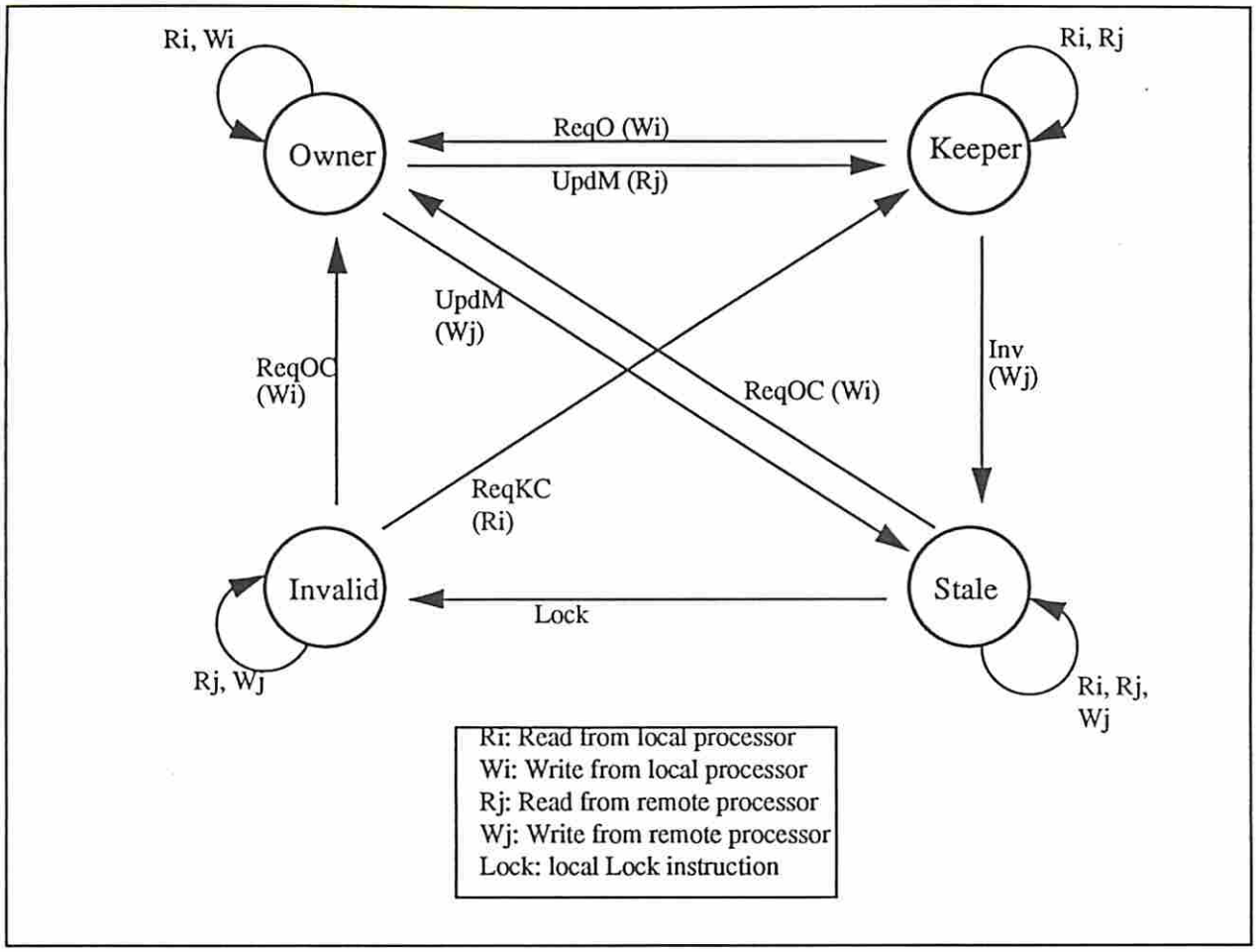


Figure 5: Cache state diagram for the Receive Delayed protocol.

Besides being receive delayed, the following protocol is also send delayed in the sense that ownership is acquired either on a Write miss or when a modified block is removed from the ISB. While the block is in the ISB, multiple Writes can be done locally and other processors can also Read and Write the block.

### Block states

*(a) Cache states.*

Similar to the states for the Receive Delayed protocol. The major difference is that a Keeper copy or a Stale copy may be Modified locally. In these cases, there is a copy of the modifications to the block in the ISB.

*(b) System Directory states.*

Same as for the On-the-Fly protocol.

### Memory commands

*(a) Commands issued by the memory controller to the caches*

Same as for the Receive Delayed protocol.

*(b) Commands issued by the cache to the memory controller*

Commands ReqO, ReqOC, ReqKC and WB are needed and trigger the same actions as in the Receive Delayed protocol. However, a new command must be introduced to notify the possible Owner or Keepers in case a Modified Keeper or Stale copy is removed from the ISB or is replaced in the cache. This new command is called ReqU (Request Update). Besides a partial update of memory (based on the words modified in the ISB), Inv signals are sent to all Fresh copies in the system.

### Cache algorithm

For different types of cache accesses, the cache controller takes the following actions (Figure 6).

. Read hit: no action is taken.

. Write hit: if the copy is Owned, no action is taken. If the cache is a Keeper or if the copy is Stale, then an entry is allocated to the block in the ISB (unless an entry is already present); the new values are stored in the buffer. Therefore, a Write hit is always a local operation.

. Read miss: a ReqKC command is sent to the memory controller.

. Write miss: a ReqOC is sent to the memory controller.

. Removing an entry from ISB: the Keepers or the Owner (if any) must be notified; if the block was Invalid or Stale in the cache then a ReqU request is sent to memory and the block stays Invalid or Stale in the cache. If the cache was a Keeper then a ReqO command is sent to the memory controller (in this case, the part of the local copy that has not been modified is consistent with memory).

. Executing a Lock instruction: all Stale blocks become Invalid right after the successful acquisition of the Lock.

. Executing an Unlock instruction: all entries must be removed from the ISB right before releasing the Lock.

. Replacement: if the cache is the Owner, then a WB command is sent to memory. A ReqU command is sent to the memory if the copy is a Stale or a Keeper copy and it has been Modified; in this case, the modifications are stored in the ISB.

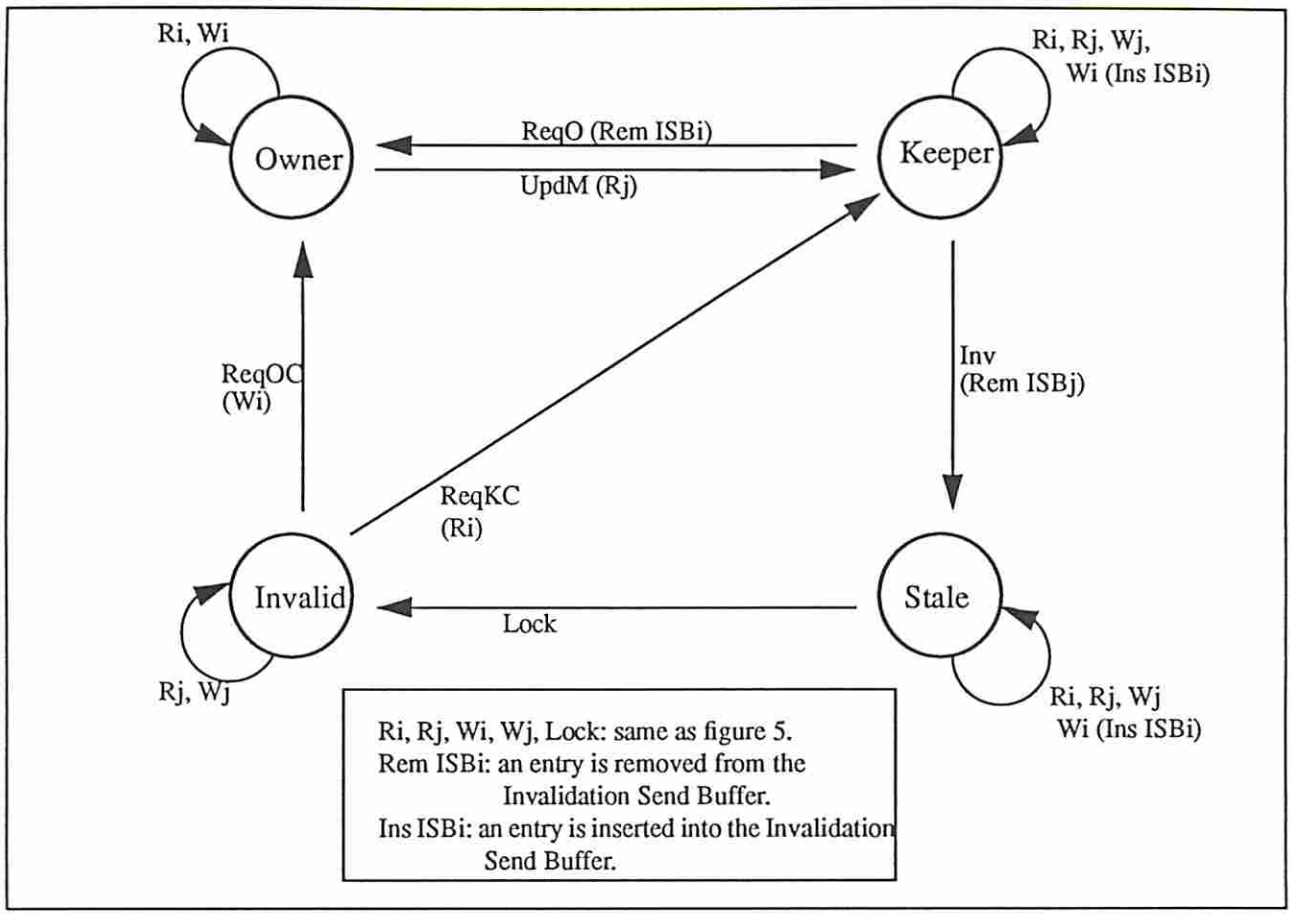


Figure 6: Cache state diagram for the Send-and-Receive Delayed protocol.

## 3.4 Hardware Implementation

At the system level, the Send-and-Receive Delayed protocol is very similar to the On-the-Fly protocol. The only major difference is the ReqU command, which uses controls needed for the ReqO command. Added complexity is essentially due to the implementation of the Stale state in the cache, of the Invalidation Send Buffer, and of the partial updates of memory.

Each block frame in the cache requires a Valid bit (V) and an Ownership bit (O). A Stale bit (S) is also needed to distinguish between a Stale and a Fresh block. When an invalidation reaches the cache, the S-bit is set and the V-bit is reset to mark the block as Stale. When a Stale block is accessed, the reset V-bit is masked by the S-bit, i.e. if the S-bit is set then the value of the V-bit is ignored. After the successful execution of a Lock instruction, all S-bits must be reset in the cache (at this point all Stale blocks become Invalid). A simple way to do this is to store the S-bits in a clearable SRAM chip (for example, part No. SN 74ACT2154 in [TI89]).

The ISB should contain no more than a few entries (2, 4 or 8). Each entry contains the block address, a Valid bit and a Dirty bit for each word of a block (see Fig. 7). The buffer is accessed associatively with the block address when a Store is done in the cache. If the copy in cache is a Keeper or a Stale copy, as indicated by the S and O bits, the ISB is consulted. If an entry already

exists for that block the dirty bit corresponding to the word is set. Otherwise an entry must be allocated in the buffer. A Keeper copy or a Stale copy are modified if there is an entry for them in the ISB. It is important to manage the ISB efficiently to avoid slowing down the processor on a Write hit. In particular, there should always be at least one free entry in the ISB at any time.

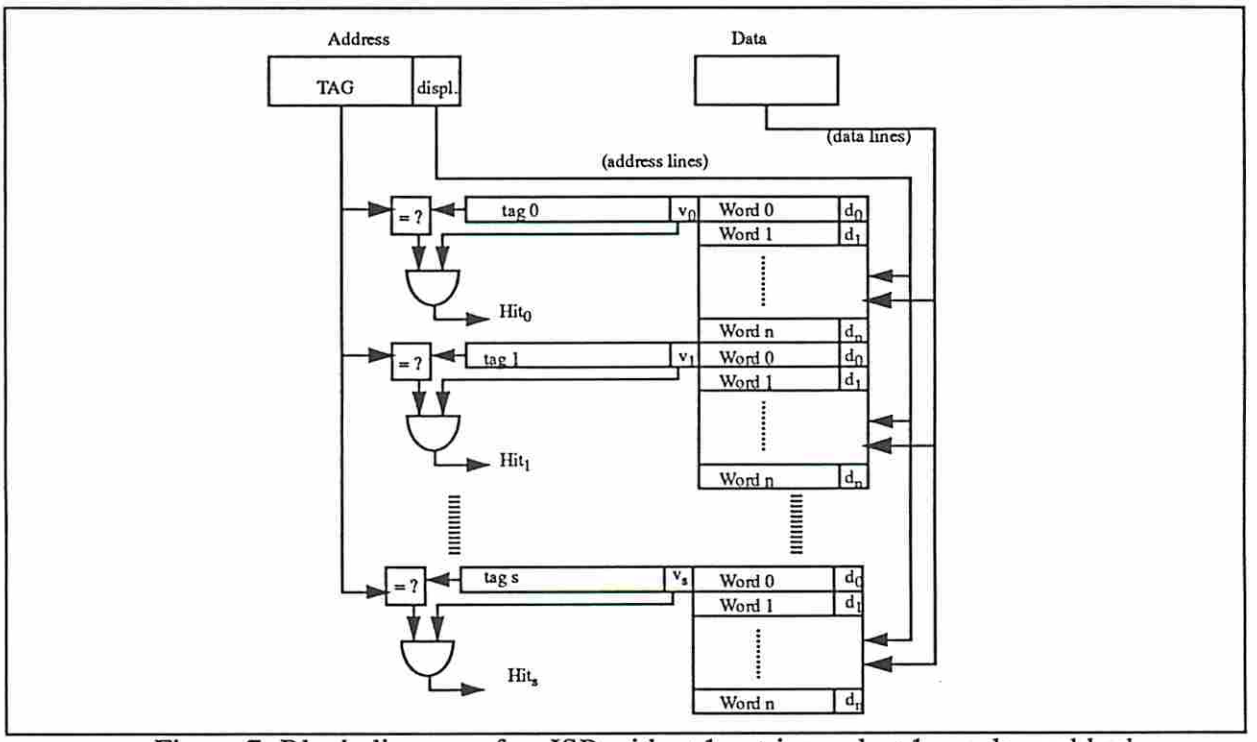


Figure 7: Block diagram of an ISB with s+1 entries and n+1 words per block

When an non-owned block copy updates memory the block in the ISB (containing both modified and "empty" words) is sent to the memory controller. The memory controller can use the dirty bits to enable/disable the Store of each word of the block.

To conclude, the implementations of the Stale state in cache and of the ISB are quite simple and should not in any way slow down the processor on a hit or in the storing of blocks to memory.

### 3.4.1 Alternate Designs

Now, we propose some refinements which could further improve the performance of the Send-and-Receive Delayed protocol.

In the present protocol, when a Keeper copy is removed from the ISB, the cache gains ownership for the block, in order to reduce the overhead for private, non-shared blocks. On the other hand, when a Stale block is removed from the ISB, all Fresh copies are notified and the block remains Stale. Besides leaving it Stale, we could envision two other strategies: invalidating the block, or acquiring ownership for the block. It may be useful to have different strategies, depending on *when* the block is removed from the ISB.

The ISB must be managed in such a way that Write hits are never slowed down, and synchronization points are executed efficiently. We propose that, in normal operation, one or two free entries be always maintained in the ISB, and that blocks be removed when the number of free entries falls below one or two. However, before an Unlock instruction, a special instruction "Prepare to Synchronize" could be issued by the compiler to reduce the number of occupied entries to one.

In the delayed protocol, the cache never blocks on a Write hit. A simple extension is to avoid blocking on Write misses as well. A Write miss simply fills a word in an entry of the ISB (such a copy, which exists in the ISB but not in the cache should be considered as Stale). The block is only loaded at the first following Read miss.

## 4.0 Effect on Miss Rates

All simulation results reported in this section were derived for infinite cache sizes and for an ISB of size two blocks (no gain was observed beyond an ISB of size 2 blocks). We only show data miss rate reduction figures. The relative reduction of the miss rates on data with respect to the On-the-Fly protocol can be seen in Tables 1 through 5, for the set of programs analyzed. In the discussion we focus on block sizes of 16 words (64 bytes), which is the block size adopted in the SCI (Scalable Coherence Interface) protocol [JLGS90]. It is also a good choice for uniprocessor caches.

The four parallel programs used in our performance studies are described below. Note that these programs exhibit different block sharing characteristics.

1) SOR: 100 iterations of the Successive Over Relaxation iterative algorithm to solve Poisson's equation on a square domain [Youn71] with single precision (32 bit) floating point numbers. The grid has size 128x128 (actually 130x130 because of boundary conditions) and is partitioned in four quadrants. Each quadrant is allocated to one processor. There are 5 Reads for each Write, and this sharing could be classified as sharing of Read/Write object [WeGu89]. Partitioning of data is static. The size of one word is 32 bits. The miss rate due to false sharing is very sensitive to the actual timing of processor execution (see Section 2). Results are reported for the worst and best cases of false sharing.

| Block size (bytes) | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Receive Delayed | 14% | 9% | 4% | 6% |
| Send-and-Receive Delayed | 14% | 14% | 12% | 10% |

Table 1: SOR (best case) - reduction on the miss rates.

2) QSORT: Quicksort [Sedg80] of a 32K file of 32-bit integers. Results are the average of the result of 10 runs for 10 different random files. Data sharing in this program can be classified as sharing of migratory objects [WeGu89], and partitioning of the data is dynamic. The number of processors

| Block size (bytes) | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Receive Delayed | 14% | 17% | 27% | 34% |
| Send-and-Receive Delayed | 14% | 24% | 45% | 65% |

Table 2: SOR (worst case) - reduction on the miss rates

is 16. The size of one word is 32 bits.

| Block size (bytes) | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Receive Delayed | 0% | 5% | 18% | 31% |
| Send-and-Receive Delayed | 0% | 23% | 41% | 63% |

Table 3: QSORT - reduction on the miss rates.

3) FLOYD: Single source shortest path problem using the Floyd-Warshall algorithm [DPL80]. There is a *path* and a *cost* array. Each graph is a random graph of 128 nodes with maximum connectivity of 96. Each result is an average over 10 runs of the algorithm on all nodes of 10 different random graphs. The two arrays are frequently read but rarely modified, and belong to the category of mostly read objects. Data partitioning is done dynamically. The number of processors is 16. All data are 32-bit words, and the size of a word is also 32 bits.

| Block size (bytes) | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Receive Delayed | 13% | 13% | 13% | 15% |
| Send-and-Receive Delayed | 13% | 13% | 13% | 15% |

Table 4: FLOYD - reduction on the miss rates

4) INTERPOLATE: Picture interpolation program. Only one out of every 9 pixels in a picture has initial values. Based on a linear interpolation the missing pixel values are computed. The size of the picture after interpolation is 96x96. The picture is divided into 8 rectangles and each rectangle is assigned to one processor. Partitioning is static. Data is either write-only or read-only. All data are 8-bit pixels, and the size of a word is also 8 bits.

Of all these programs FLOYD exhibits the least number of false sharing transitions. The reason is that the two arrays are accessed mostly randomly and are read most of the time. INTERPOLATE exhibits large false sharing effects. It is not iterative like the other three

| Block size (bytes) | 16 | 32 | 64 | 128 |
|---|---|---|---|---|
| Receive Delayed | 75% | 88% | 90% | 93% |
| Send-and-Receive Delayed | 75% | 88% | 90% | 93% |

Table 5: INTERPOLATE - reduction on the miss rates

algorithms. Processes read the known pixel values in one array, compute each unknown value and store it in a second array. Therefore one array is read-only, while the other is write-only. There is no Read/Write sharing of data, and consequently there is no locking in the whole program, except at the beginning (to fork the processes) and at the end (to terminate). There is some Read sharing, but all the coherence activity is due to false sharing transitions caused by stores. This type of sharing is very frequent. It occurs for example in a Doall loop where successive iterations of a loop compute the components of a vector or an array, and are allocated to different processors. A classical example is the Doall loop computing the product of two matrices A and B and storing the result in a matrix C.

The results above show that delayed consistency is effective at reducing the number of misses when the effect of false sharing transitions is large. FLOYD has very few false sharing transitions and therefore it shows little gain. The best cases of SOR have few false sharing transitions, and most of the false sharing transitions occur across barrier synchronizations, so that the Stale bit and the ISB are not very effective (a reduction of misses of only 12% is observed for a block size of 64 bytes); for the worst case of SOR, delaying consistency is extremely effective in the case of a block size of 64 bytes, because false sharing effects are intense and occur mostly between barrier synchronizations (within the same sweep). The gains in QSORT (P=16) only occur for block sizes larger than 16 bytes, because the false sharing effects are very small for smaller blocks. For a block size of 64 bytes (file size of 32K), the reduction in the number of misses approaches 41% for a Send-and-Receive Delayed protocol. Finally, as expected, INTERPOLATION benefits the most from delayed consistency (90% reduction in the number of misses for block size 64), because all misses (except initial loading misses) are due to false sharing and because the algorithm does not require any synchronization during most of its execution.

## 5.0 Conclusion

Both write-invalidate and write-broadcast protocols may be send delayed or receive delayed. In this paper, we have introduced two write-invalidate delayed consistency protocols, built as extensions of an existing protocol. Delays are obtained through one buffer and the addition of the Stale bit in the cache state. Implementation of an ISB allows the overlapping of cache activity and sending of invalidations.

Delayed consistency also reduces the number of false sharing transitions. This is important in systems supporting efficiently both parallel and single thread processes. Significant reductions in the miss rate on data can be obtained with a small ISB. This reduction is obtained with no assistance from the programmer and the compiler, and therefore it is particularly useful for

general-purpose multiprocessors. The only restriction is that parallel programs must access shared-writable data in critical or semi-critical sections.

## 5.0 References


[ABM89] Y. Afek, G. Brown, and M. Merritt, "A Lazy Cache algorithm," *Proc. of the 1st ACM Symp. on Parallel Algorithms and Architectures*, pp. 209-223, Jun 1989.

[ACCK90] Alverson at al., "The TERA Computer System," *Proc. of the ACM Int Conf on Supercomputing*, pp. 1-6, Jun 1990.

[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.

[Ahmd67] G.M. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," *Proc AFIPS*, Vol. 30, pp.483-465, 1967.

[ALKK90] A. Agarwal et al., "APRIL: A Processor Architecture for Multiprocessing," *Proc. of the 17th Annual Int. Symp on Comp. Arch.*, pp. 104-114, Jun 1990.

[BCZ90a] J.K. Bennett, J.B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proc. of the 2nd ACM Symposium on Principles and Practice of Parallel Programming, SIGPLAN Notices* 25:3, Mar 1990, pp. 168-176

[BCZ90b] J.K. Bennett, J.B. Carter, and W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures," *Proc. of the 17th Annual Int. Symposium on Computer Architecture*,Jun 1990, pp. 125-134.

[BiDe86] P. Bitar and A. Despain, "Multiprocessor Cache Synchronization: Issues, Innovations, Evolution," *Proc. of the 13th Annual Int. Symp. on Comp. Architecture*, June 1986, pp. 424-433.

[BiRa90] R. Bisiani, and M. Ravishankar, "PLUS: A Distributed Shared-Memory System," *Proc. of the 17th Annual Int. Symposium on Comp. Architecture*, pp. 115-124, June 1990..

[BLPS79] B.M. Bean et al., "Bias Filter Memory for Filtering Out Unnecessary Interrogations of Cache Directories in a Multiprocessor System," U.S. Patent 4,142,234, Feb 1979.

[BoHe90] L. Borrmann and M. Herdieckerhoff, "A Coherency Model for Virtual Shared Memory," *Proc. of Int. Conf. on Parallel Processing*, Vol.2, pp.252-257, June 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.

[DPL80] N. Deo, C.Y. Pang, and R.E. Lord, "Two Parallel Algorithms for Shortest Path Prob-

lems," *Proc. of the Int. Conf. on Parallel Proc.*, pp. 244-253, Aug. 1980.

[DuSc90a] M. Dubois and C.Scheurich, "Memory Access Dependencies in Shared Memory Multiprocessors," *IEEE Transactions on Software Eng.*, 16(6), pp. 660-674, June 1990.

[JLGS90] D.V. James et al., "Scalable Coherent Interface," *IEEE Computer*, Vol. 23, No. 6, pp. 74-77, June 1990.

[Kowa85] J.S. Kowalik, editor, "Parallel MIMD Computation: HEP Supercomputer and Its Applications", The MIT Press, 1985.

[Krof81] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization," *Proc. of the 8th Ann. Int. Symp. on Comp. Arch.*, pp.81-87, 1981.

[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.

[LLGG90] D. Lenoski et al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc of the 17th Annual Int Symposium on Comp Arch*, pp. 148-159, Jun 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, June 1987.

[MBBD91] Mudge, T. N. et al., "The Design of a Microsupercomputer", *IEEE Computer*, January 1991, pp.57-64.

[MoGu91] Mowry, T. and Gupta, A., "Tolerating Latency through Software-Controlled Prefetching in Scalable Shared-Memory Multiprocessors", to appear in the *Journal of Parallel and Distributed Computing*, 1991.

[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, June 1984.

[Sche89] C. Scheurich, "Access Ordering and Coherence in Shared-Memory Multiprocessors," PhD thesis, University of Southern California, May 1989.

[ScDu91] C. Scheurich and M. Dubois, "Lockup-free Caches in High-Performance Multiprocessors," *Journal of Parallel and Distributed Computing*, January 1991.

[Sedg80] R. Sedgewick, "Quicksort", New York: Garland Publishing, Inc., 1980.

[SoFr91] Sohi, G. and Franklin, M., "High-Bandwidth Data Memory Systems for Superscalar Processors", to appear in *APLOS IV*,1991.

[Sten90] P. Stenstrom, "A Survey of Cache Coherence Scheme for Multiprocessors," *IEEE*

*Computer*, Vol. 23, No. 6, Jun 1990.

[TI89] Texas Instrument MOS Memory Data Book, pp. 7-135 to 7-147, 1989.

[TLH90] J. Torrellas, M.S. Lam, and J.L. Hennessy, "Shared Data Placement Optimizations to Reduce Multiprocessor Cache Misses," *Proc. of the 1990 Int. Conf. on Parallel Proc.*, Aug 1990, pp. 266-270.

[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.

[WeGu89] W-D Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors," *Proc. of the 3rd Int. Conf. on Arch. Support for Prog. Languages and Systems(ASPLOS)*, pp.243-256, Apr 1989.

[Youn71] D. Young, "Iterative Solution of Large Linear Systems", Academic Press: New York, 1971.