Appears in: "Proceedings of the 22nd Annual International Symposium on Computer Architecture"
Reprinted by permission of ACM

Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors

Alvin R. Lebeck and David A. Wood
Computer Sciences Department
University of Wisconsin-Madison
Madison, Wisconsin 53706 USA
{alvy,david}@cs.wisc.edu

Abstract

This paper introduces dynamic self-invalidation (DSI), a new technique for reducing cache coherence overhead in shared-memory multiprocessors. DSI eliminates invalidation messages by having a processor automatically invalidate its local copy of a cache block before a conflicting access by another processor. Eliminating invalidation overhead is particularly important under sequential consistency, where the latency of invalidating outstanding copies can increase a program's critical path.

DSI is applicable to software, hardware, and hybrid coherence schemes. In this paper we evaluate DSI in the context of hardware directory-based write-invalidate coherence protocols. Our results show that DSI reduces execution time of a sequentially consistent full-map coherence protocol by as much as 41%. This is comparable to an implementation of weak consistency that uses a coalescing write-buffer to allow up to 16 outstanding requests for exclusive blocks. When used in conjunction with weak consistency, DSI can exploit tear-off blocks—which eliminate both invalidation and acknowledgment messages—for a total reduction in messages of up to 26%.

This work is supported in part by NSF PYI Award CCR-9157366, NSF Grants CDA-9024618 and MIP-9225097, donations from Thinking Machines Corp., Digital Equipment Corp., Xerox Corp., and by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant F33615-94-1-1525 and ARPA order no. B550
The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

1 Introduction

Shared-memory multiprocessors simplify parallel programming by providing a single address space even when memory is physically distributed across many workstation-like processor nodes. Most shared-memory multiprocessors use cache memories to automatically replicate and migrate shared data and implement a coherence protocol to maintain a consistent view of the shared address space [8,21,26,29].

Write-invalidate protocols allow multiple processors to have copies of shared-readable blocks, but force a processor to obtain an exclusive copy before modifying it [8,22,29]. Directory-based protocols invalidate outstanding copies by sending explicit messages to the appropriate processor nodes [3,7,39]. When a node receives an invalidation message, it invalidates its local copy and sends an acknowledgment message back to the directory. (This message also contains the data for exclusive blocks).

The performance of these protocols might improve significantly if we could eliminate the invalidation messages (without changing the memory semantics). An oracle could do this by simply making the processors replace blocks just before another processor makes a conflicting access. Thus, the processors would *self-invalidate* their own blocks instead of waiting for the directory to send explicit invalidation messages. This would improve performance by reducing the latency and bandwidth required to satisfy conflicting memory requests.

The principal contribution of this paper is a practical approach for *dynamic self-invalidation (DSI)* of cache blocks. We show how the directory can dynamically identify which blocks should be self-invalidated, convey this information back to the cache in response to a miss, and how the cache controller can later self-invalidate the selected blocks at an appropriate time.

Self-invalidation cannot make a correct program incorrect, since it has exactly the same semantics as a cache replacement. However, self-invalidating blocks too early can cause unnecessary cache misses, hurting rather than

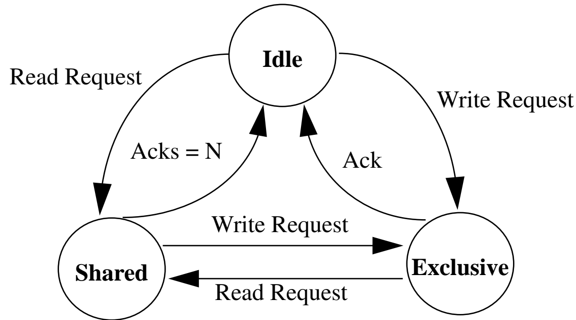


Figure 1. Cache Block Directory States

helping performance. Therefore, the DSI implementation must minimize the number of explicit invalidations without significantly increasing the number of misses.

This paper introduces and evaluates two methods for identifying which blocks to self-invalidate: additional directory states and version numbers. Our results indicate that a 4-bit version number generally performs better than the additional state method. We also investigate two techniques for the cache controller to self-invalidate the blocks: a FIFO buffer and by selective cache flushes at synchronization operations. Simulations show that selectively flushing is more effective because the FIFO's finite size can cause self-invalidation to occur too early.

The benefit of DSI is significant when coherence traffic dominates communication. For most of our benchmarks, a sequentially consistent memory system with DSI performs comparably to a weakly consistent implementation that allows up to 16 outstanding requests for exclusive blocks, but stalls on read misses. Execution times with the sequentially consistent protocol improve by up to 41%, depending on the cache size and network latency. When used with weak consistency, DSI can eliminate both invalidation and acknowledgment messages by allowing nodes to obtain copies of a cache block without updating the directory state. Our results show that while DSI improves the performance of one benchmark by 18%, it has little effect on execution time for most programs. However, combining DSI and weak consistency can eliminate 50–100% of the invalidation messages, reducing the total number of messages by up to 26%. Our results indicate that DSI will have the greatest impact in systems with large, multi-megabyte caches, e.g., a portion of main memory [21,33], since data is seldom replaced, or with relatively slow networks, such as networks of workstations [5].

This paper is organized as follows. Section 2 reviews invalidation-based coherence protocols and discusses related work. Section 3 presents dynamic self-invalidation and discusses the design space. Section 4 describes our implementations of dynamic self-invalidation protocols, Section 5 evaluates their performance, and Section 6 concludes our paper.

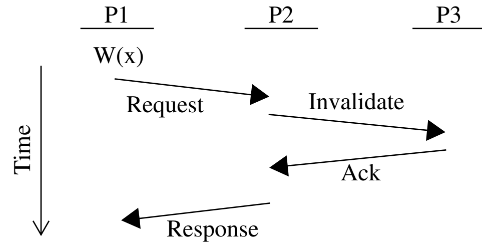


Figure 2. Coherence Overhead

2 Background and Related Work

DSI techniques are applicable to hardware [29], software [35], and hybrid systems [8,26,33]. In this paper we evaluate DSI in the context of a full-map directory-based hardware cache coherence protocol [3]. We assume a typical write-invalidate protocol with three states (see Figure 1): no outstanding copies (Idle), one or more outstanding shared-readable copies (Shared), or exactly one outstanding readable and writable copy (Exclusive). A processor must obtain an exclusive copy of a block before modifying it; the directory enforces this by sending explicit invalidation messages to eliminate any outstanding copies.

The overhead of these invalidation messages is particularly significant under *sequential consistency* [27], the programming model most programmers implicitly assume. A multiprocessor is sequentially consistent if the execution corresponds to some interleaving of the processes on a uniprocessor. Conventional directory-based write-invalidate coherence protocols maintain sequential consistency by stalling a processor on a write miss until it receives acknowledgment that all cached copies have been invalidated,[1] as shown in Figure 2. Unfortunately, the latency of sending invalidations and collecting acknowledgments may lie on the program's critical path, and therefore degrade performance.

Self-invalidation techniques can eliminate the invalidation and acknowledgment messages from the sequence illustrated in Figure 2, significantly reducing the latency required to obtain a cache block. When self-invalidation is performed perfectly, read requests always find the block in state Idle or Shared and write requests always find the block in state Idle.

Previous self-invalidation techniques rely on memory system directives inserted by the compiler, profile-based tools, or the programmer. *Compiler-directed coherence* [9,14,16,30] eliminates the directory, placing the entire burden of maintaining cache coherence on the compiler.

1. It is possible to have P2 respond immediately with the data, and P3 send an acknowledgment directly to P1 [29]. However, P1 stalls until the acknowledgment is received.

Unfortunately, this technique requires sophisticated analysis, and has only been demonstrated to work well for regular scientific applications and one-word cache blocks.

Other self-invalidation techniques combine memory system directives with a conventional directory-based write-invalidate protocol. In CICO, the programmer [22,40] or a profile-based tool [10] annotates the program with `check_in` directives to inform the memory system when it should invalidate cache blocks. In contrast to compiler-directed coherence, `check_in` directives are only performance hints to the memory system; the directory hardware is still responsible for correctness.

Self-invalidation can be used with other techniques that reduce the impact of coherence overhead. Prefetching cache blocks before their expected use hides the latency to obtain a cache block [31,20]. Multithreading [37,20] tolerates latency by rapidly switching to a new computation thread when a remote miss is encountered. Migratory data optimizations [12,38] speculate about future write requests by the same processor when responding to a read request. Self-invalidation is complementary to these optimizations and could be combined with them. For example, the SPARC V9 `prefetch-read-once` instruction [6] indicates that a block should be prefetched, but then self-invalidated after the first reference.

Weak consistency models [2,17,19] also reduce the impact of coherence overhead. A system that provides weak consistency appears sequentially consistent provided that the program satisfies a particular synchronization model [2]. Weak consistency models allow the use of memory access buffering techniques—e.g., write buffers. They also allow the directory to respond with the data in parallel with the invalidation of outstanding copies, and the processor can proceed as soon as it receives the data. The acknowledgments can be sent directly to the requesting processor [29], or collected by the directory which forwards a single acknowledgment. The processor stalls at synchronization operations, depending on the specific consistency model, until all preceding writes are acknowledged. Adve and Hill proposed a similar scheme for sequential consistency [1]; however, they do not provide any quantitative results. As discussed in Section 3.3, self-invalidation can eliminate acknowledgment messages when combined with weak consistency.

3 Dynamic Self-Invalidation

In this section we present a general framework for performing dynamic self-invalidation (DSI). Similar to other forms of self-invalidation, DSI attempts to ensure that data is available at the home node when another processor requests access. However, DSI does not rely on programmer intervention; instead, self-invalidation is performed automatically by the coherence protocol.

Write-invalidate coherence protocols generally involve the following operations:

1. identify that a cache block requires invalidation,
2. perform the invalidation, and
3. acknowledge the invalidation, if necessary.

Conventional protocols tightly couple the identification of a block for invalidation (step 1) with its invalidation (step 2). The directory explicitly invalidates outstanding copies when servicing cache misses. In contrast, DSI decouples these steps, speculatively identifying which blocks to invalidate when they are brought into the cache, but deferring the invalidation itself to a future time.

The remainder of this section discusses the dynamic self-invalidation design space. Section 3.1 discusses techniques for the directory controller, cache controller, or software to identify blocks for self-invalidation. Section 3.2 discusses performing self-invalidation with the cache controller or software, and Section 3.3 discusses the acknowledgment of invalidation messages.

3.1 Identifying Blocks

Identifying blocks for self-invalidation requires speculating the likelihood that a block will be invalidated in the near future. This identification can be implemented by the directory controller, the cache controller, software, or any combination of the three.

Software approaches issue directives to the memory system to identify which blocks to self-invalidate. Unfortunately, these techniques require either programmer annotations, a sophisticated compiler, or a profile-based tool. Furthermore, implementing these directives either requires special instructions not present in all instruction sets, or additional memory mapped loads and stores.

In this paper we focus on hardware techniques that automatically identify blocks for self-invalidation. A directory controller can identify a block for self-invalidation by maintaining a history of its sharing pattern. When servicing a request for a cache block, the directory uses this extra information to predict if the block is likely to be invalidated in the future, and conveys this information to the caching node with the response.

Similarly, a cache controller can identify blocks for self-invalidation by maintaining information for recently invalidated blocks [15] (e.g., the number of times a block is invalidated). When servicing a cache miss, this history information is used by the controller to decide if it should self-invalidate this block at a later time.

3.2 Performing Self-Invalidation

Software, hardware, or a combination can be used to perform self-invalidation. The caching node must record

the identity of the blocks selected for self-invalidation and invalidate them at a point in the future that maximizes performance.

Systems that maintain cache coherence in software [24,26,33,35] can use arbitrary data structures to store block identities. The blocks are self-invalidated using the same primitives required to process explicit invalidation messages. Alternatively, hardware managed caches can maintain a hardware data structure, such as an auxiliary buffer or an extra bit in the cache tag. Software examines this hardware data structure, self-invalidating the blocks by issuing directives to the memory system.

There are many alternatives for performing self-invalidation entirely in hardware. In Section 4 we present two hardware methods that can be implemented in the (second-level) cache controller. The first scheme uses a first-in-first-out (FIFO) buffer; blocks are self-invalidated when they fall out of the buffer. The second technique performs self-invalidation at synchronization operations using custom hardware.

3.3 Acknowledging Invalidation Messages

In conventional directory-based write-invalidate protocols, the directory records—or *tracks*—the identity of nodes holding copies of a cache block. By tracking blocks, the directory can always explicitly invalidate cached copies when necessary. Thus self-invalidation is semantically equivalent to a cache replacement and places no restrictions on the memory consistency model. Self-invalidation of tracked blocks can reduce latency and eliminate invalidation messages. However, acknowledgment messages are still required to inform the directory that a node has invalidated its copy of the block.

We can eliminate both invalidation and acknowledgment messages by guaranteeing to self-invalidate blocks at specific points according to the memory consistency model. For these blocks—called *tear-off* blocks—the directory does not track the outstanding copy.

Scheurich observed that the invalidation of a cache block could be delayed until the subsequent cache miss and still maintain sequential consistency [34]. The intuition behind this observation is that a processor can continue to access data until it "sees" new data generated by another processor. To maintain sequential consistency the cache controller must invalidate tear-off blocks at subsequent cache misses. Therefore, a cache may contain at most one tear-off block. Note that tear-off blocks are only useful for shared-readable blocks, since the acknowledgment for exclusive blocks is generally coupled with the transfer of modified data.

A further caveat is that using tear-off blocks with sequential consistency does not guarantee forward progress. If a processor obtains a tear-off block containing a spin lock [4], it may never experience a subsequent cache miss. The spin lock will never be invalidated and the processor will not proceed. To overcome this, the tear-off block could be self-invalidated periodically, e.g., at context switches.

Tear-off blocks are potentially much more significant under weaker consistency models. A processor can cache multiple tear-off blocks since the model does not guarantee that a processor can "see" data generated by another processor until it performs a synchronization operation. By self-invalidating its local tear-off blocks at each synchronization point, a processor ensures that it can see all other processors' modifications to shared data.

4 Implementation

In this section we present several different implementations of DSI. We focus on techniques where the directory identifies which blocks should be self-invalidated and the cache controller performs the self-invalidation. The directory conveys self-invalidation information to the cache when responding to a miss. Blocks that are not self-invalidated are explicitly invalidated in the conventional manner. We describe two methods for the directory to identify blocks for self-invalidation, followed by two techniques for the cache to perform the invalidations. We evaluate these implementations in Section 5.

4.1 Identifying Blocks

The directory controller provides a single point for monitoring a cache block's sharing patterns. This section presents two techniques for the directory controller to identify which blocks should be self-invalidated: additional states and version numbers. Both implementations are extensions to a standard three state full-map directory-based write-invalidate protocol, such as Dir_nNB [3].

Both implementations use the sharing history to speculate about the future: blocks that have recently had conflicting accesses—and hence would have needed invalidations—are candidates for self-invalidation. Thus, shared-readable blocks are marked for self-invalidation if they have been modified since the last reference by the processor. Likewise, exclusive blocks are marked for self-invalidation if they have been read or modified by a different processor since the writing processor's last access.

Through experimentation we found two special cases where it is better to avoid self-invalidation. First, blocks are not self-invalidated from the home node's cache. Second, under sequential consistency, exclusive blocks are not marked for self-invalidation if the writing processor had a shared-readable copy and there are no other outstanding copies. This upgrade case can cause unnecessary self-invalidation of exclusive blocks, degrading performance

for some programs under sequential consistency. This special case is not needed under weak consistency, since the write buffer hides the latency of the additional write misses.

Additional States

Our first implementation uses four additional states to identify which blocks should be self-invalidated. When servicing a read request, the directory responds with a self-invalidate block if the current state is exclusive. These blocks enter a new state (Shared_SI) that causes all subsequent read requests to obtain a block marked for self-invalidation. We also add two new states (Idle_X, Idle_S) to detect transitions into the idle state from the exclusive or shared-readable state resulting from self-invalidation. Finally, we add one state (Idle_SI) to detect transitions into the idle state resulting from the cache replacement of a self-invalidate block.

The directory responds to a write request with a self-invalidate block if the current state is: Shared, Shared_SI, Exclusive, Idle_S, Idle_SI, or Idle_X where a different processor had the block exclusive. Read requests obtain a self-invalidate block if the current state is: Exclusive, Idle_X, Shared_SI or Idle_SI.

If tear-off blocks are supported, each directory entry requires one additional bit to indicate that there is more than one outstanding tear-off block. This bit allows correct identification of exclusive blocks for self-invalidation when servicing a write request from a processor that had a tear-off block.

Version Numbers

Version numbers provide an alternative scheme that identifies when blocks are modified by different processors. This additional information allows processors to decide independently whether to obtain a self-invalidate block. In contrast, all processors make the same decision using the state method.

The directory maintains a version number for each block and increments it each time any processor requests an exclusive copy. This scheme requires the cache controller to store the version number with the associated block. On a miss, if there is a tag match but the block is invalid, the corresponding version number is sent with the request for the block. The directory responds with a self-invalidate block if the current version number is different from the version number of the request. If the cache controller does not provide a version number (i.e., there is not a tag match), the directory responds with a normal block. Since the version number is only a performance hint, we can use a small number of bits and allow wrap-around without violating correctness.

Identifying exclusive blocks for self-invalidation requires additional information, since the version numbers may match yet another processor has read the block. To address this situation, we add two bits to each directory entry that count the number of shared-readable copies distributed for the current version of the block. Each time the directory responds with a shared-readable block, a ‘one’ is shifted into the low-order bit. Both bits are cleared when the version number is incremented. Therefore, write requests obtain a self-invalidate exclusive block if either the version numbers do not match, or the current version has been read by at least two processors (which may include a previous read by the writing processor).

4.2 Performing Self-Invalidation

In this section we present two techniques for the cache controller to self-invalidate blocks using information readily available from many commodity processors.

The first implementation uses a first-in-first-out (FIFO) policy for self-invalidation blocks. When the cache controller receives a self-invalidate block, it records the identity of the block in the FIFO. Blocks are self-invalidated when an entry in the FIFO is replaced. In addition, if we can identify synchronization operations, such as `test&set` or `swap`, then we can also flush the FIFO at those points.

Implementing the FIFO requires the addition of a small memory to store the identity of the blocks to self-invalidate. This buffer—similar to a victim cache [23] or the HP PA7200 assist cache [25]—is unlikely to exceed 64 entries. Nonetheless, this is an attractive approach since it does not rely on any information from the processor.

If the cache controller can identify synchronization operations, then there are other schemes for performing self-invalidation. In particular, we can eliminate the FIFO and flush all self-invalidate blocks from the cache after one or more synchronization operations [11,18]. In this paper, we focus on invalidating blocks at each synchronization point.

The precise implementation depends on the specific DSI protocol. All the implementations require an additional bit, *s*, associated with each cache tag. The *s* bit indicates that the block should be self-invalidated, which is accomplished by clearing the corresponding valid bit. When a new block is brought into the cache, the *s* bit is set if the block has been selected for self-invalidation.

Self-invalidation of tracked blocks requires the cache controller to send an acknowledgment (or notification) message to the directory. The control logic must find which blocks to self-invalidate—marked by the *s* bit—and recreate the full addresses by concatenating the cache index with the cache tag.

The naive implementation sequentially examines each cache frame, self-invalidating the block and sending a message, if necessary. However, the overall latency will be proportional to the number of cache frames, even though many blocks may not be self-invalidated.

We can reduce this latency using a circuit that sequences through only the blocks that must be self-invalidated. One implementation uses a modified flash clear circuit [28] to determine the next cache set that contains a block to self-invalidate, and requires an encoder to recreate the cache index. This encoder is roughly the same size as the set index decoder, and, for set-associative caches, it can be shared by all cache frames in the same cache set.

Alternatively, we could use a hardware linked list, which adds a pointer to each cache set, and maintains a head and a tail pointer. The pointers store the cache index of the next block to self-invalidate. When a self-invalidate block is brought into the cache, its corresponding pointer is assigned to the current value of the tail, and the tail is updated to point to the new block. At synchronization operations, the list is traversed from tail to head. Set-associative caches require only one pointer per cache set. A set is inserted in the list when it receives its first self-invalidate block; during self-invalidation the set must be searched for all blocks with the *s* bit equal to one.

These implementations achieve similar performance, processing only blocks that require self-invalidation. Note that self-invalidation of tracked blocks can overlap with the execution of the processor, staging out the messages and possibly avoiding severe network congestion or synchronization delays. However, the quantitative results in this paper assume the processor does not proceed past synchronization points until all blocks are self-invalidated and that messages are injected as rapidly as the network can accept them.

Self-invalidating tracked blocks always requires messages to the directory, and the latency to perform self-invalidation is proportional to the number of blocks self-invalidated. However, when both tear-off blocks and exclusive blocks are self-invalidated, only the exclusive blocks require a message to the directory. The tear-off blocks can be self-invalidated in a single cycle using a simple flash clear circuit; the exclusive blocks must be sequentially self-invalidated using one of the techniques described above.

5 Performance Evaluation

In this section we evaluate the effectiveness of DSI by comparing it to a full-map protocol [3]. Section 5.2 evaluates the detection and self-invalidation mechanisms under sequential consistency. In Section 5.3, we evaluate the benefit of adding dynamic self-invalidation to a weak consistency implementation that allows up to 16 outstanding requests for exclusive blocks.

Name	Input Data Set
Barnes	2048 bodies, 5 iterations
EM3D	192,000 nodes, degree 5, 5% remote
Ocean	98x98, 1 day
Sparse	512x512 dense, 5 iterations
Tomcatv	512x512 5 iterations

TABLE 1. Application Programs

This table describes the benchmarks used in this paper. Sparse is locally-written[40], EM3D is from the Berkeley Split-C group [13], Barnes and Ocean are from the Stanford SPLASH suite [36], and Tomcatv is a locally written, parallel version of the SPEC benchmark.

5.1 Methodology

We use a modified version of the Wisconsin Wind Tunnel [32] to simulate 32-processor systems with 256K-byte and 2M-byte 4-way set-associative caches with 32-byte blocks. Cache misses occupy the cache controller for 3 cycles and the directory controller for 10 cycles, plus message injection time. The message injection overhead is 3 cycles, with an additional 8 cycles if a cache block must be sent. We assume a constant 100 cycle network latency, and do not model contention in the switches. However, contention is accurately modeled at the directory, cache and network interface. Instruction execution time is obtained by modeling the SuperSPARC processor, which can issue up to three instructions per cycle. We assume that SPARC `swap` instructions and a hardware barrier, with a 100 cycle latency from the last arrival, are visible to the memory system.

The base cache coherence protocols are all full-map protocols. The sequentially consistent implementation stalls the processor on all misses. The directory invalidates outstanding copies and collects acknowledgments before forwarding the block to the requesting processor.

For weak consistency, we use a 16-entry coalescing write buffer. Each entry in the write buffer contains an entire cache block, and write misses that match an outstanding request are merged into the existing entry. The directory in our weak consistency protocol grants exclusive access to a block in parallel with the invalidation of outstanding shared-readable blocks. A single acknowledgment is sent to the owning processor after the directory collects the invalidation acknowledgments. The processor stalls at `swap` and `barrier` operations until all previous

writes are acknowledged. The processor also stalls on read misses until the block is obtained.

We present results from five benchmarks in our evaluation of DSI, see Table 1. We focus specifically on the parallel portion of the programs, clearing all statistics after initialization.

5.2 Sequential Consistency Results

In this section we evaluate DSI in the context of sequential consistency. We begin with an evaluation of the detection mechanisms, described in Section 4.1, and assume we have custom hardware to perform the self-invalidation at synchronization operations. This is followed by a discussion of performing self-invalidation with a FIFO buffer.

The main results from this study are that:

1. DSI can give sequential consistency performance comparable to an implementation of weak consistency.
2. Version numbers are more effective than additional states for detecting which blocks to self-invalidate.
3. Performing self-invalidation at synchronization operations is better than using a finite-size FIFO.
4. DSI is most effective when coherence overhead dominates communication.

DSI improves execution time by up to 41%, depending on the cache size and network latency. For all but one of our benchmarks, these execution times are comparable to our weakly consistent implementation. Furthermore, the benefit of DSI is much larger when coherence overhead is high. When coherence overhead is low neither weak consistency nor DSI have much effect on execution time.

Detection Mechanisms

In this section we examine the performance of detecting blocks using additional states and 4-bit version numbers. Figure 3 shows execution time normalized to the base sequentially consistent protocol. The left most bar is the base sequentially consistent protocol (SC), followed by the weakly consistent (W) and DSI protocols with additional states (S) and version numbers (V), respectively. Based just on total execution time, the results indicate that sequentially consistent DSI achieves performance roughly comparable to the base weakly consistent implementation for all programs except `ocean`.

To look further, we refine execution time into computation, synchronization, read invalidation, read other, write invalidation, write other, and other (e.g., TLB misses and I/O). Read (write) invalidation is the time spent waiting at the directory for outstanding copies to be invalidated, and represents the maximum time DSI can eliminate. For weak consistency, we also include the time spent waiting: at synchronization points for the write buffer to drain (synch wb), on read misses for which there is already an outstanding write miss (read wb), and when the write buffer is full (wb full). For the DSI protocols we include the time spent waiting for the self-invalidation to complete (DSI). Although, our simulations show that this time is too small to perceive.

This breakdown shows that `Barnes` has a large synchronization component in its execution time, due primarily to fine-grain locking and load imbalance for this small data set. Neither weak consistency nor DSI yield significant performance improvements.

`EM3D` spends most of its time waiting for cache misses. DSI reduces the write invalidation time, producing improvements within 5% of the weakly consistent protocol, which eliminates all write latencies. For the 256K-byte cache, execution time improves by 25% for weak consistency, 15% for DSI using states, and 13% for the version number implementation. For the 2M-byte cache, improvements are 32%, 27%, and 27%, respectively. DSI does not reduce the read invalidation time because `EM3D` uses local allocation and all modifications to shared data occur on the home node.

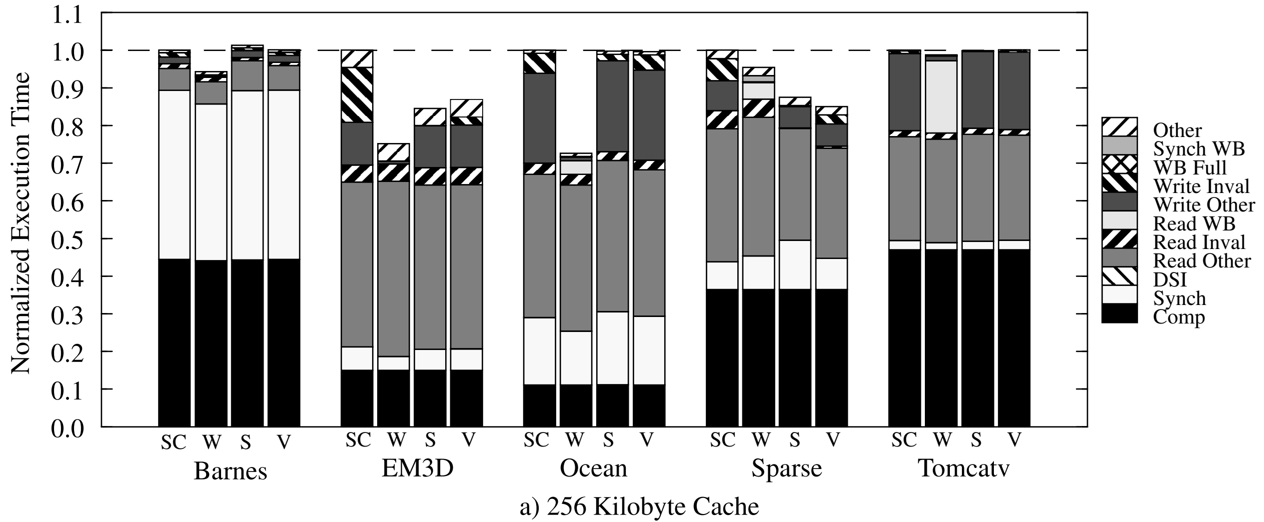
DSI has little effect on the execution time of `ocean` for either cache size because of un-synchronized accesses to shared data. In contrast, weak consistency reduces execution time by 27% and 32% for the two cache sizes.

For `sparse`, DSI reduces both read invalidation and write invalidation delays, *outperforming* weak consistency by as much as 10%. Weak consistency improves performance by 5% for the 256K-byte cache and 9% for the 2M-byte cache. DSI provides 13% and 10% improvements using additional states and 15% for both cache sizes using version numbers.

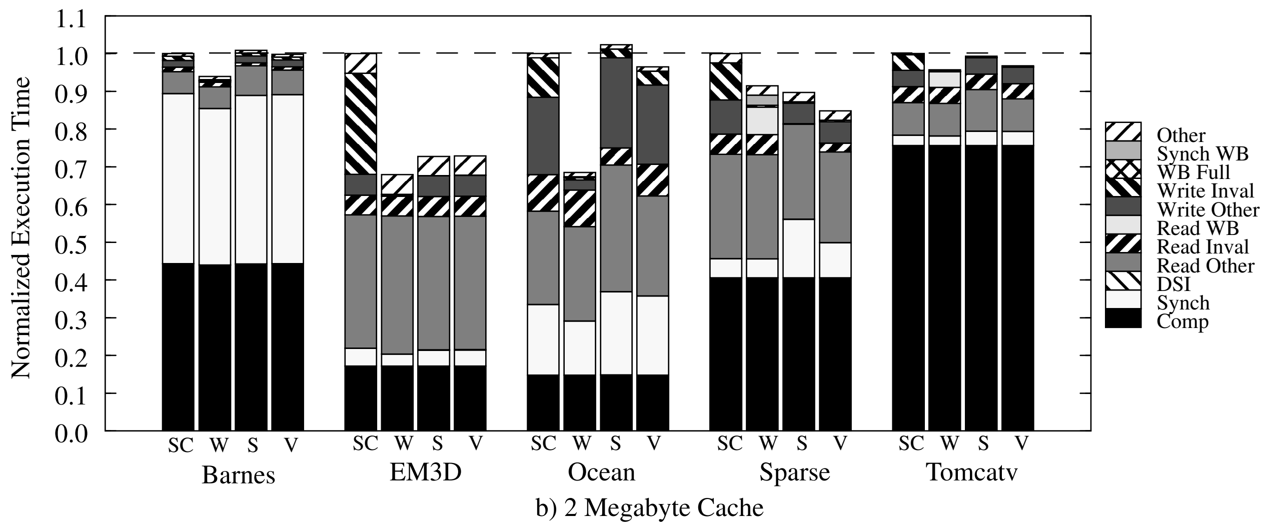
For the 256K-byte cache, `tomcatv` shows no change in execution time for any protocol, since its data set is too large for the cache. Weak consistency eliminates the write stall time, but read stalls increase because there is a read miss for a block with an outstanding write miss. For the larger cache, `tomcatv`'s execution time is dominated by computation, weak consistency and DSI with version numbers improve execution time by only 4% and 3% respectively.

Impact of Network Latency

As processor cycle times continue to decrease relative to network latencies, the impact of coherence overhead increases. To evaluate the benefit of DSI under these conditions we increased the network latency to 1000 cycles (10μs @ 100 MHz). This generally increases the benefit of both DSI and weak consistency. With a 256K-byte cache weak consistency reduces execution time by 8% for `barnes`, 33% for `EM3D`, 32% for `ocean`, 15% for `sparse`, and 1% for `tomcatv`. DSI improves `EM3D`'s performance



a) 256 Kilobyte Cache



b) 2 Megabyte Cache

Figure 3. Performance of Dynamic Self-Invalidation Under Sequential Consistency

SC = Sequential Consistency, W = Weak Consistency, S = DSI using additional states, V = DSI using version numbers

by 32% using states and 26% using version numbers. `Barnes` and `tomcatv` show very little change from the 100 cycle network. DSI provides less benefit for `sparse` with the higher network latency; improving performance by only 2% using states and 9% using version numbers.

For a 2M-byte cache size (see Figure 4) DSI using version numbers reduces execution time by 41% for `EM3D`, 5% for `ocean`, 21% for `sparse`, and 12% for `tomcatv`. Using additional states to detect self-invalidate blocks improves `EM3D`'s execution time by 41% and `tomcatv`'s by only 4%. `Ocean`'s execution time is unaffected, while this method actually increases the execution time of `barnes` and `sparse`. Thus, 4-bit version numbers generally perform better than additional states.

The results in this section show that DSI can improve the performance of a sequentially consistent full-map directory-based protocol by eliminating invalidation latencies. For all but one of our benchmarks, DSI achieves performance comparable to an implementation of weak consistency. The benefit of DSI is most pronounced when coherence activity dominates communication. When a program's data set does not fit in the cache, coherence overhead is low and the benefit of DSI decreases. These results suggest that systems using main memory as a cache for remote data, e.g., COMA, [21,33] may benefit significantly from self-invalidation.

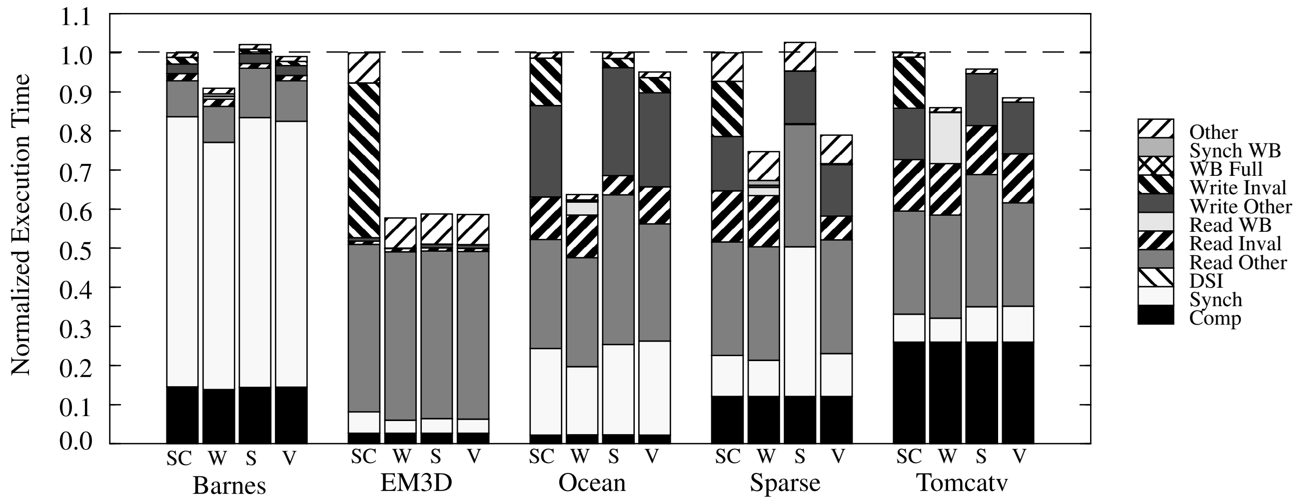


Figure 4. Impact of Network Latency

2M-byte cache, 1000 cycle network latency

SC = Sequential Consistency, W = Weak Consistency, S = DSI using additional states, V = DSI using version numbers

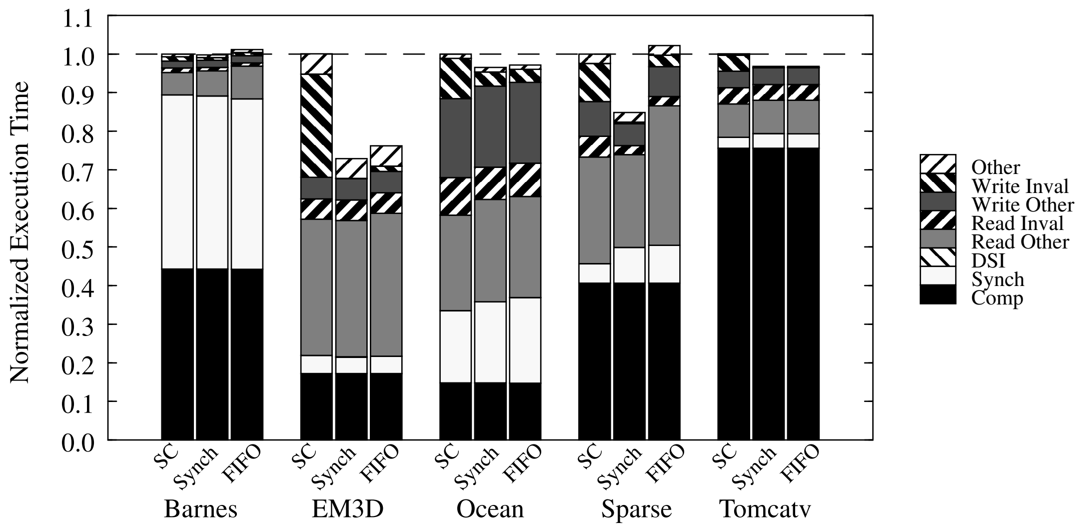


Figure 5. Self-Invalidation Mechanisms

2M-byte cache, 100 cycle network latency, DSI with version numbers

Self-Invalidation Mechanisms

In this section we evaluate the two techniques for self-invalidating blocks at the cache. The FIFO buffer has 64 entries and is flushed at each synchronization operation. The selective flush hardware uses a linked list to invalidate marked blocks at each synchronization point. The directory uses version numbers to determine which blocks should be self-invalidated.

The results, shown in Figure 5, are for a 2M-byte cache; however, there is no qualitative difference for a 256K-byte cache. For `barnes`, `EM3D`, `ocean`, and `tomcatv` there is little difference between the self-invalidation schemes. However, `sparse` exhibits a dramatic difference, with self-invalidation at synchronization operations significantly outperforming the FIFO buffer. The FIFO is unable to contain all the self-invalidate blocks in the program's working set. Blocks are self-invalidated too early, causing a subsequent miss which obtains a normal cache block. This is a fundamental problem with a finite size buffer, and can significantly undermine the benefit of DSI.

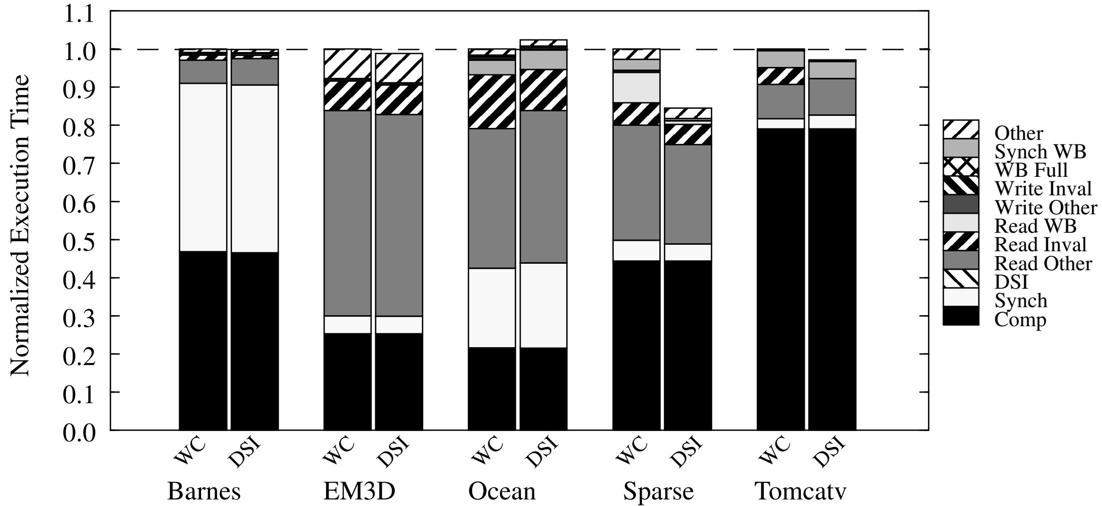


Figure 6. DSI and Weak Consistency
2M-byte cache, 100 cycle network latency, DSI with version numbers

Benchmark	100 cycle network		1000 cycle network	
	256 KB	2 MB	256 KB	2 MB
Barnes	1.01	1.00	1.00	1.00
EM3D	0.99	0.99	1.00	1.00
Ocean	1.00	1.02	0.99	1.04
Sparse	**0.82**	**0.84**	**0.90**	0.96
Tomcatv	1.00	0.97	1.00	**0.86**

TABLE 2. Weakly Consistent DSI Normalized Execution Time

Benchmark	Total Messages		Invalidation Messages	
	256 KB	2 MB	256 KB	2 MB
Barnes	5%	6%	45%	51%
EM3D	17%	26%	85%	100%
Ocean	4%	12%	32%	52%
Sparse	7%	1%	54%	66%
Tomcatv	0%	21%	45%	100%

TABLE 3. DSI Message Reduction

5.3 DSI and Weak Consistency

This section evaluates the benefit of DSI in the context of weak consistency. DSI and weak consistency both reduce the impact of coherence overhead, and the results in Section 5.2 show they often achieve comparable reductions in execution time. When DSI is used in conjunction with weak consistency the directory can utilize tear-off blocks, as described in Section 3.3, to eliminate acknowledgment messages. Furthermore, the write buffer can mitigate the effects of self-invalidating exclusive blocks incorrectly, and we can eliminate the special case for exclusive blocks described in Section 4.1.

For most of our programs there is very little effect on execution time, as shown in Table 2 and Figure 6. `Sparse` is the exception, where DSI with weak consistency improves performance by up to 18% over weak consistency alone. `Tomcatv` shows a 14% reduction in execution time for the 2M-byte cache with a 1000 cycle network. This is a direct consequence of eliminating the special case for exclusive blocks; DSI eliminates both write invalidation and read invalidation latencies.

DSI with tear-off blocks eliminates both invalidation and acknowledgment messages. Tear-off blocks potentially reduce both the total message traffic and the directory controller occupancy. The latter may have a significant effect on systems that cannot process local memory accesses in parallel with protocol events (e.g., FLASH [26]). The results in Table 3 show that DSI reduces the total number of messages by up to 17% for a 256K-byte cache and 26% for a 2M-byte cache. To the first order, directory controller occupancy will be reduced by the same amount.

6 Conclusion

Coherence overhead in directory-based write-invalidate protocols can significantly degrade performance. In this

paper, we presented dynamic self-invalidation (DSI), a new technique for reducing coherence overhead. DSI eliminates invalidation messages by having processors automatically invalidate local copies of cache blocks before another processor accesses the block. Therefore, the directory can immediately respond with the data when processing a request for the block.

We evaluated DSI in the context of a full-map hardware cache coherence protocol. In our implementations, the directory identifies cache blocks for self-invalidation and the cache controller performs the self-invalidation. Under sequential consistency—where the latency of invalidating outstanding copies may lie on a program's critical path—DSI reduces execution by up to 41%, depending on the cache size and network latency. Under weak consistency, DSI generally had little effect on execution time, although one benchmark improved by 18%. However, combining DSI and weak consistency permits exploitation of *tear-off blocks*, where the directory does not track the outstanding copies. This eliminates both invalidation and acknowledgment messages, reducing the total number of messages by up to 26%.

We presented two techniques for the directory to identify which blocks should be self-invalidated: additional states and version numbers. Our simulations reveal that version numbers generally outperform additional states. We also evaluated two approaches for the cache controller to perform the self-invalidation: a FIFO buffer, and at synchronization operations using custom hardware. Self-invalidation at synchronization operations utilizes the full capacity of the cache, and significantly outperforms the finite-size FIFO for some applications.

DSI is a general technique, applicable to hardware, software, and hybrid cache coherent shared-memory multiprocessors. Current trends in parallel architectures, e.g., faster processors and larger caches, can make coherence overhead a significant fraction of execution time. If this trend continues, DSI should be of increasing benefit.

Acknowledgments

We thank Alain Kägi for his modifications to the Wisconsin Wind Tunnel that facilitated our implementation of weak consistency. Mark Hill, James Larus, Guri Sohi, Anne Rogers, Thea Sklenar, Steve Reinhardt and Rahmat Hyder provided helpful comments on this paper. We also thank the members of the Wisconsin Wind Tunnel project for their support and the referees for many useful suggestions.

References

[1] Sarita V. Adve and Mark D. Hill. Implementing Sequential Consistency In Cache-Based Systems. In *ICPP90*, pages I47–I50, August 1990.

[2] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[4] Thomas E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing (Vol. II Software)*, pages II170–II174, August 1989.

[5] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*. To appear.

[6] Brian Case. SPARC V9 Adds Wealth of New Features. *Microprocessor Report*, 7(9), February 1993.

[7] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[8] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[9] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *IEEE Computer*, 23(6):39–48, June 1990.

[10] Trishul M. Chilimbi and James R. Larus. Cachier: A Tool for Automatically Inserting CICO Annotations. In *Proceedings of the 1994 International Conference on Parallel Processing (Vol. II Software)*, pages II–89–98, August 1994.

[11] Lynn Choi and Pen-Chung Yew. A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality. In *Proceedings of Supercomputing 94*, pages 773–782, Nov 1994.

[12] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[13] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing 93*, pages 262–273, November 1993.

[14] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic Management of Programmable Caches. In *Proceedings of the 1988 International Conference on Parallel Processing (Vol. II Software)*, pages 229–238, Aug 1988.

[15] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Combined Performance Gains of Simple Cache Protocol Extensions. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 187–197, April 1994.

[16] Ervan Darnell and Ken Kennedy. Cache Coherence Using Local Knowledge. In *Proceedings of Supercomputing 93*, pages 720–729, Nov 1993.

[17] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[18] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 201–213, November 1994.

[19] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[20] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[21] E. Hagersten, A. Landin, and S. Haridi. DDM—A Cache-Only Memory Architecture. *IEEE Computer*, pages 44–54, September 1992.

[22] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in ASPLOS V, Oct. 1992.

[23] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[24] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwanenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operations Systems. Technical Report COMP TR93-214, Department of Computer Science, Rice University, November 1993.

[25] Gordon Kurpanek, Ken Chan, Jason Zheng, Eric Delano, and William Bryg. PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface. In *Compcon*, pages 375–382, 1994.

[26] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[27] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[28] A. R. Lebeck. *Tools and Techniques for Memory System Design and Analysis*. PhD thesis, University of Wisconsin at Madison, expected August 1995. Computer Sciences Department.

[29] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[30] Sang Lyul Min and Jean-Loup Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):25–44, January 1992.

[31] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[32] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[33] Steven K. Reinhardt, James R. Larus, and David A. Wood. Typhoon and Tempest: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[34] Christoph Ernst Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989. Also available as technical report No. CENG 89-19.

[35] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steve K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. Submitted for publication, March 1994.

[36] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[37] B. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proceedings of the Int. Soc. for Opt. Engr*, pages 241–248, 1982.

[38] Per Stenstrom, Mats Brorsson, and Lars Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[39] C. K. Tang. Cache System Design in the Tightly Coupled Multiprocessor System. In *Proc. AFIPS*, pages 749–753, 1976.

[40] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993.