# A Comparative Evaluation of Cache Coherence Schemes Based on Virtual Memory Support

Karin Petersen Kai Li
Department of Computer Science
Princeton University

**Abstract**

This paper presents an evaluation of a new class of software cache coherence schemes that use virtual memory (VM) support to maintain multiprocessor cache coherence. Traditional VM translation hardware in each processor detects memory access attempts that would violate cache coherence and system software is used to enforce coherence. The implementation of this class of coherence schemes is extremely economical: it requires neither special multiprocessor hardware nor compiler support, and easily incorporates different consistency models.

We have evaluated four consistency models for the VM-based approach: sequential consistency, single-writer release consistency, release consistency, and lazy release consistency. Our trace-driven simulation results show that the VM-based cache coherence schemes are practical for small-scale multiprocessors and that the performance of lazy release consistency for multi-threaded parallel programs is close to the snoopy-cache invalidation-based coherence approach.

**Keywords:** shared memory, multiprocessors, cache coherence, memory management unit, virtual memory, consistency models.

## 1 Introduction

Most previously proposed and implemented methods to maintain cache coherence in multiprocessor systems fall into two categories: hardware-based approaches, that require complete architectural support to implement the coherence protocols, and application-level software methods that require compiler support and often additional architectural support to improve performance. In this paper we present our evaluation of a new approach that requires neither special multiprocessor hardware, nor compiler support. It uses VM system level software to keep caches coherent.

The most common hardware-based solutions use either snoopy caches [LT88, McC84, PP84, RS85, TS87] or directories [ASHH88, LLJ+92]. Snoopy-cache protocols rely on a bus-based interconnection of the nodes in the system. Cache controllers listen to all transactions on this bus and take appropriate actions to keep caches coherent. These schemes do not scale well due to increasing bus traffic when the number of processors connected to the bus increases. Directory-based schemes use directory entries to keep track of cached copies for each shared memory block, and invalidate or update all copies upon updates to the memory block. Since directory-based schemes allow arbitrary interconnection networks, they scale better than the bus-based snoopy cache schemes. However, all hardware-based methods suffer from the same intrinsic problem: they require special purpose hardware that is expensive both in design time and chip real estate.

Application level software-based methods require compiler support to identify accesses to shared variables that might cause cache incoherencies. [Vei86] presents an algorithm executed by the compiler which inserts cache invalidation instructions at parallel loop boundaries. In [LYL87] the compiler divides programs into segments called *epochs*, and tags variables within an epoch as non-cacheable if they are shared and written, or cacheable otherwise. At the end of each epoch caches are invalidated, and write-back is used to keep main memory updated. A similar technique is used for the NYU Ultracomputer [EGK+85] and RP3 [BMW85] multiprocessor. Several forms of special hardware support have been proposed to improve the peformance of these compiler assisted schemes such as using hardware to selectively invalidate cache lines [CV88], to support efficient invalidations [Smi85], and to reduce unnecessary invalidations [MB89].

An important advantage of these software-based methods is that they are economical in terms of hardware: it need only support a few instructions used by compilers to invalidate and flush the cache. Previous research [AAHV91, OA89] shows that the performance of compiler-based software schemes is comparable to hardware approaches for programs that do not suffer a significant cache hit reduction caused by inaccurate static predictions of memory access conflicts. On the other hand, compiler-based solutions are expensive in terms of having to be conservative when determining the interference of shared memory accesses, and by requiring special hardware support for performance.

The new class of software coherence schemes we are investigating requires neither special hardware support, nor compiler support. The multiprocessor architecture required to implement these schemes is extremely simple: a network connecting a number of off-the-shelf uniprocessors, uniprocessor caches, and memory modules. The virtual memory translation hardware on each processor detects shared accesses that would leave memory incoherent, and the VM page fault handlers execute the appropriate actions to maintain cache coherence.

This paper reports the trace-driven simulation results of four memory consistency models including sequential consistency, single-writer release consistency, release consistency, and lazy release consistency. We compare them with the Illinois invalidation-based snoopy-cache protocol [PP84], and to do so, we simulate a bus-based architecture and experiment with several unmodified programs written for snoopy-cache multiprocessors. Our simulation results show that the new software cache coherence methods are very practical for small-scale multiprocessors, and that the performance of lazy release consistency for multi-threaded parallel programs is close to the snoopy-cache coherence mechanism.

## 2 VM-based Cache Coherence

The idea of the VM-based cache coherence schemes stems from previous work on shared virtual memory [LH89]. This section describes the basic mechanism required in VM-based cache coherence schemes and the algorithms for implementing three memory consistency models.

### Architecture and Mechanisms

The multiprocessor architecture for VM-based cache coherence schemes is extremely simple. An interconnection network to connect off-the-shelf uniprocessors, uniprocessor physical caches, and standard memory modules as shown in Figure 1 will be sufficient. The only hardware support needed is an interprocessor interrupt mechanism over the network.

The VM-based schemes require support from the memory management unit (MMU) and the virtual memory system on each uniprocessor:

- The MMU allows for at least three different protection states associated with a page: *read-only* (READ), *read/write* (WRITE), and *no access* (NIL).

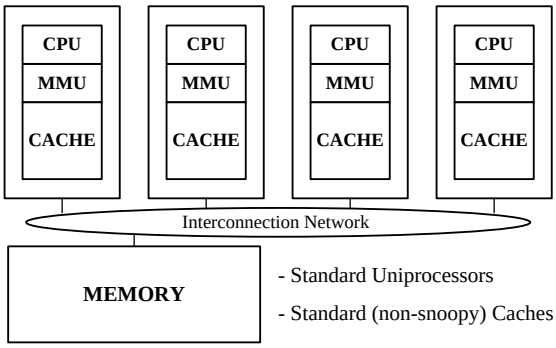


Figure 1: A simple multiprocessor using uniprocessor CPUs, caches, and memory modules.

- The caches are physical caches with a *write-through* policy and support for selective invalidation of cached lines from software, as supported by architectures like the SPARC-1, the Intel i860, the MC68040 and the IBM RP3 [BF90, Int89, ELM90, BMW85]. We will also explain how to implement the VM-based schemes with *write-back* caches.
- For each virtual address space, each processor has its own page table which resides in non-cached memory (i.e., they directly go to TLB entries and bypass the physical cache). The base addresses of these page tables are known to all processors.

Most stock uniprocessors satisfy these requirements.

The VM-based cache coherence software maintains cache coherence at the virtual memory address space level. When creating a shared address space, the system initializes a page table for each processor. In these tables, the entries for shared pages have the same virtual-to-physical address translations. According to a specific memory consistency model, the system initializes the protection bits of the page table entries and maintains them at run time. A memory reference causes a page access fault when the memory location is in a protected page. A fast page access fault handler fields the fault and perform appropriate actions to maintain cache coherence.

## Sequential Consistency

Sequential consistency (SC) [Lam79], the strictest of all memory consistency models, requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. At a given point in time, sequential consistency allows only one writer per data item. In the absence of a writer, it allows multiple readers.

The algorithm of implementing SC for the VM-based cache coherence approach is:

**Initial state:** Set the protection bits of all page table entries (PTEs) of each processor to NIL.

**Read fault on page $i$:** Loop through every processor's PTE for page $i$. If a PTE has WRITE protection, interrupt its processor and change its PTE to READ. After checking all the PTE's, the faulting processor changes its PTE to READ and restarts.

**Write fault on page $i$:** Loop through every processor's PTE for page $i$. If a PTE has READ or WRITE protection, interrupt its processor, make it invalidate its cache lines for page $i$, then change its PTE to NIL. After checking all the PTE's, the faulting processor updates its PTE to WRITE, and restarts.

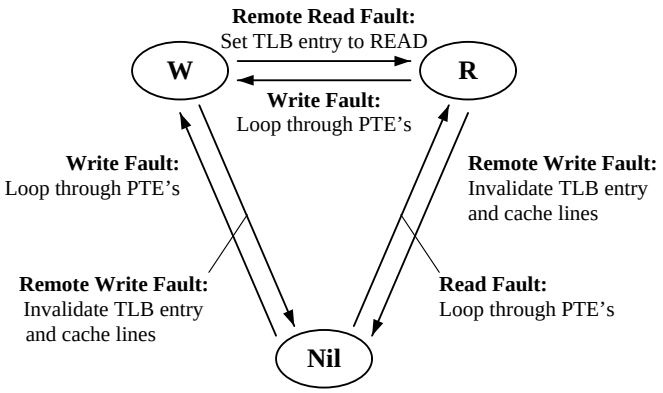


Figure 2: Transition diagram for sequential consistency.

Figure 2 illustrates this algorithm with a transition diagram for the state of a page. The diagram shows six possible transitions of a page's protection setting, along with the actions taken on each transition. In the diagram, a *remote page fault* means a page fault on a different processor.

Figure 3 shows an example of SC for a two processor system. Processors $P1$ and $P2$ are reading data from page $i$, hence the protection bits for page $i$ on both processors are READ. After issuing read memory references the caches of both processors contain certain data from page $i$.

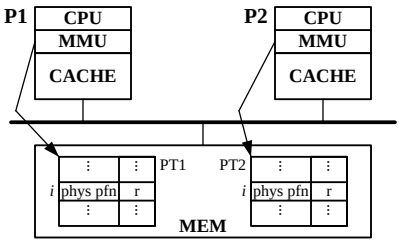


Figure 3: P1 and P2 read sharing page $i$

Suppose that processor $P2$ performs a write to a memory location in page $i$. An access violation occurs and the page fault handler on $P2$ fields the trap. The fault handler checks processor $P1$'s page table entry for page $i$, changes it from READ access to NIL, has $P1$ invalidate the cache lines of page $i$, changes its own page table entry to WRITE, and then returns to the faulting instruction. The final state is shown in Figure 4.

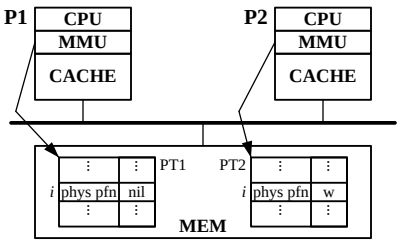


Figure 4: After $P2$ writes into page $i$

The algorithm can be modified to use write-back caches instead of write-through caches. With write-back caches, the memory may not have up-to-date data; our algorithm must flush cache entries associated with a page when undergoing a transition from a WRITE state. These transitions now take longer, but

overall traffic along the interconnection could be reduced.

The cost of looping over all page table entries for page $i$ is proportional to the number of processors in the system. Although the cost of finding a page table entry is only a few instructions, the cost of finding all page table entries for a page can be noticeable. Looping can be avoided by maintaining a pointer $W_i$ for every page $i$. $W_i$ points to either NULL or to the processor which holds WRITE access to page $i$. The faulting processor needs to loop through PTEs only when a write fault occurs and $W_i$ is NULL. One can avoid all looping by using a copyset bit-vector is used to keep track of which processors have read access, similar to the existing directory schemes.

### Fast Cache Invalidation

The bottleneck of the algorithm just described is the invalidation of cache lines when the protection of a page changes from READ or WRITE to NIL. Loop through all cache lines belonging to a page and invalidating them individually is inefficient.

An alternative is to postpone the invalidation of cache lines until the reverse protection transition, from NIL to READ or WRITE. We call such a method *lazy invalidation* because it only invalidates cache lines from a page before they become accessible. Lazy invalidation is correct because the PTE with no access protection setting will prevent the CPU from accessing any data from that page. Although lazy invalidation avoids invalidations to pages that will never be accessible, the cost for those that are going to be reaccessed is still high.

Our fast cache invalidation mechanism reduces the cost of an invalidation to that of executing several instructions. It is based on the notion of aliases for physical memory addresses. Aliases reference the same physical memory location, but have different physical cache tags. Therefore, a cache hit with one alias address will be a miss with another alias address, although referring to the same physical memory location. This notion was implemented with special hardware in the TLB's and caches for the one-time-identifier scheme discussed in [Smi85]. We use the unused high order bits in physical memory addresses to achieve physical address aliasing without special hardware support. Using unused physical address bits for aliasing was first proposed in [GL91] for cache consistency in the presence of dual-ported memories.

We use the unused high order bits in a page table entry as a page version number (equivalent to one-time-identifier). Suppose that there are $k$ unused bits and that $version_i$ is the version number for page $i$. Initially, $version_i$ is set to 0. When the protection of page $i$ is changed from NIL to READ or WRITE, instead of invalidating the cache lines, the algorithm changes its PTE entry and lets $version_i = (version_i + 1) \bmod 2^k$. By incrementing the version number, the virtual page translation starts using a different alias physical page number. The new physical address guarantees that the CPU sees no stale data in the cache for virtual page $i$, since no address translation will provide old physical addresses. When the value of $version_i$ is reset to 0, the system performs the expensive invalidations for all cache lines in the page.

The amortized cost of the fast invalidation method depends on how many unused physical address bits are available. One way to get more unused bits is to have a limit on the total shared physical pages and to place all shared physical pages in one reserved section of the physical address space. For instance, if we set such a limit to 8M bytes, and use half of the physical address space for shared addresses, a 32-bit machine will have 8 unused bits, which means that one can perform 256 fast invalidations for each slow invalidation.

### Relaxed Consistency Models

Relaxed consistency models [AH90, DSB86, GLL+90] are based on the following realization: most shared memory multiprocessors guarantee that all memory accesses are globally performed, i.e., when a memory

access completes all processors see its results. However, most programmers only use this guarantee for synchronization variables (e.g. semaphores), and then use the ordering of these variables to ensure the correct use of the rest of their programs' variables. Under relaxed memory consistency models, ordinary accesses can be issued and performed in any order, and synchronization points are viewed as fences that force memory accesses to be globally performed and therefore return the system to a coherent state. These methods often significantly reduce the access time of shared data structures by hiding the latency of each memory access.

In addition to hiding latency, relaxed consistency models also reduce some page faults caused by false-sharing. Figure 5 shows data access sequences on two processors. Assume that the variables $a$ and $b$ reside on the same page $i$. Right before time $t_1$, both processors have read access to the page. Under sequential consistency using the lazy invalidation method, when $P2$ writes $b$, it generates a write fault and initiates the action to change $P1$'s PTE to NIL and change $P2$'s PTE to WRITE. At time $t_2$, $P1$ writes $a$. $P1$ has a write fault and changes $P2$'s PTE to NIL, invalidates it's own (i.e., $P1$'s) cache lines for page $i$ and changes it's own PTE to WRITE. At time $t_3$, $P2$ reads $a$. $P2$ has a read fault, changes $P1$'s PTE to READ, invalidates it's own cache lines from the page, and changes $P2$'s PTE to READ. There are three page faults and two cache invalidations.

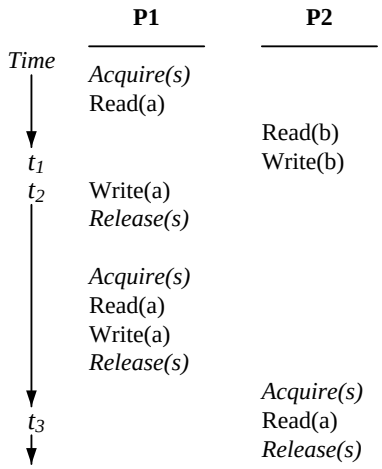


Figure 5: Example for relaxed consistency schemes.

Under a relaxed consistency model where reads and writes are ordered through `Release` and `Acquire` accesses, $P1$ and $P2$ can both have access to page $i$ at time $t_1$ and $t_2$. In this case, there will be no page faults until $P2$ performs the `Acquire` right before time $t_3$. In this example, a relaxed consistency model can eliminate two page faults due to false sharing.

## Release Consistency

The release consistency (RC) model used in the VM-based cache coherence mechanism adheres to conditions proposed in [GLL+90].

There are two kinds of synchronization accesses in the RC model: `Acquire`, and `Release`. They are processor consistent[1] with respect to one another and ordinary accesses are fenced by the synchronization accesses:

[1] The conditions for processor consistency are that load accesses stall for other load accesses to complete, but may bypass stores, and stores can perform only if all previous loads and stores are performed.

- Before an ordinary READ or WRITE access is allowed to perform with respect to any other processor, all previous `Acquire` accesses must be performed.
- Before a `Release` access is allowed to perform with respect to any other processor, all previous ordinary READ and WRITE accesses much be performed.
- Synchronization accesses are processor consistent with respect to one another.

Our model is stronger than the RC model defined in [GLL+90], because we enforce SC on all synchronization accesses, while RC only requires processor consistency on their ordering.

We partition an application's shared pages into two sets: $S$ contains only synchronization variables, and $G$ contains ordinary data structures. Faults are handled differently for pages in the two sets. For faults on pages in $G$, we no longer enforce a policy of exclusive writers or shared readers. Instead, we let processors read and write their own copies of pages simultaneously, and worry about flushing and invalidation when there are faults of pages in $S$.

On `Acquires` and `Releases`, the processor invalidates cache lines of shared pages. In order for a processor to know which pages are currently being shared and updated we define a globally shared list of pages called the `WeakList`. A page is added to the list when it becomes shared and writable, and is removed from the list when all processors have relinquished (READ or WRITE) access to it. On an `Acquire`, a processor checks all pages in the `WeakList` to which it has access, and performs the appropriate invalidations. A counter, $weak_i$, is used to keep track of the number of processors that have access to page $i$ at any given time.

The algorithm for our RC is:

**Initial State:** Same as before. `WeakList` = NULL, and $weak_i = 0$ for all $i$.

**Read Fault on page $i \in G$:** Increment $weak_i$. If there is a processor writing page $i$ then add $i$ to the `WeakList`. Set the PTE to READ and restart.

**Write Fault on page $i \in G$:** Increment $weak_i$ if faulting from NIL access. If $weak_i > 1$ then add $i$ to the `WeakList`. Set the PTE to WRITE and restart.

**Page Fault on page $i \in S$:** Treat these as in the sequentially consistent case. In addition, before restarting do the following:

> **If faulted on `Acquire`:** For all pages $i \in$ `WeakList`, if the processor has READ (or WRITE) access to $i$: invalidate the cache lines in page $i$, and set the access for $i$ to NIL. Decrement $weak_i$, if $weak_i = 0$, remove page $i$ from the `WeakList`.
>
> **If faulted on `Release`:** Make sure the write buffer is empty.

The implementation of this algorithm in our simulation allocates a page for each synchronization variable and its data structure.

If we use write-back caches instead of write-through caches, the algorithm becomes slightly more complicated. On a `Release`, before flushing the write buffer, a processor must flush all its dirty cache lines from pages in the `WeakList`, in order for ordinary read and write operations to get performed before the `Release` is performed. Additionally, a page must be added to the `WeakList` as soon as it becomes writable, regardless of whether it is shared. Otherwise, assuming variables $a$ and $b$ are in different pages, then the scenario in Figure 5 would cause $P2$ to receive a stale value of $a$, since it is not shared when $P1$ performs its `Release`.

### Lazy Release Consistency

The main idea behind lazy release consistency (LRC) [KCZ91] is that ordinary updates need not be visible to all processors on release operations, as long as there is a guarantee that when a processor acquires a lock, it will see all updates to shared variables preceding the last release of that lock. Intuitively, the advantage of LRC over RC is that it reduces the number of global updates.

LRC must adhere to the following constraints, as defined in [KCZ91]:

- Before an ordinary (non-synchronization) READ or WRITE access is allowed to perform with respect to any other processor, all previous (in $\rightarrow$ order) `Acquire` accesses must be performed with respect to that other processor.
- Before a `Release` access is allowed to perform with respect to any other processor, all previous (in $\rightarrow$ order) ordinary READ and WRITE accesses much be performed with respect to that other processor.
- Synchronization accesses must be sequentially consistent with respect to one another.

The $\rightarrow$ ordering of shared memory accesses is defined as follows:

- If $a_1$ and $a_2$ are consecutive accesses in the same process, and $a_1$ occurs before $a_2$ in program order, then $a_1 \rightarrow a_2$.
- If $a_1$ is a `Release` access in process $P_1$, and $a_2$ is an `Acquire` access in process $P_2$ to the same synchronization variable, such that $a_2$ is the first `Acquire` on that variable to perform at $P_1$ after the `Release` of $a_1$, then $a_1 \rightarrow a_2$.
- if $a_1 \rightarrow a_2$ and $a_2 \rightarrow a_3$ then $a_1 \rightarrow a_3$.

The $\rightarrow$ ordering of accesses between distinct processes is only a consequence of a `Release-Acquire` pair between these processes. Our algorithm for LRC is:

**Initial State:** Same as before. `WeakList` = NULL, and $weak_i = 0$ for all $i$.

**Read Fault on page $i \in G$:** Increment $weak_i$. If there is a processor writing page $i$ then make sure $i$ is in the `WeakList`. Set the PTE to READ and restart.

**Write Fault on page $i \in G$:** Increment $weak_i$ if faulting from NIL access. If $weak_i > 1$ then make sure $i$ is in the `WeakList`. Set the PTE to WRITE and restart.

**Page Fault on page $i \in S$:** Treat these as in the sequentially consistent case. In addition, before restarting do the following:

**If faulted on `Acquire`:** Interrupt processor $w_i$, which currently has WRITE access to the page that contains the lock, and wait for $w_i$ to flush all cache entries for pages in the `WeakList` and if necessary to release the lock. For all pages $i \in$ `WeakList`, if the processor has READ or WRITE access to $i$, then invalidate the cache lines in page $i$, and set the access of $i$ to NIL. Decrement $weak_i$ and if $weak_i = 0$, remove page $i$ from the `WeakList`.

**If faulted on `Release`:** and another processor had previously interrupted to indicate an acquire, drain write-buffer before the release is restarted.

In our implementation of LRC stores get performed for all nodes when the main memory gets updated and the processors PTE's for pages in the `WeakList` get invalidated in order to trigger a reload of the data item. The `WeakList` implements the idead of the **context graph** explained in [KCZ91].

To implement LRC, we allocate each synchronization variable on a separate page, in order to properly detect the previous owner of a lock. We expect the number of locks per application not to be too large, and therefore believe not to be wasting to many physical memory pages by taking this approach.

Consider again the execution sequences in Figure 5. At the first `Release` after $t_2$, page $i$ is weak since both $P1$ and $P2$ have read access and updated variables from this page. Under RC, when $P1$ releases $s$ for the first time, it will make sure all updates to page $i$ are performed, i.e., the new value of $a$ will have to be written to the shared memory. In reality there is no need for $P_1$ to make sure that $a$ has been performed at the shared memory level on the first release. This is what LRC tries to exploit: $P2$ will interrupt $P1$ when acquiring $s$, for $P1$ to make sure all updates previous to its second release of $s$ are performed with respect to $P2$, i.e, are written to the shared memory, thus avoiding one update of shared memory. Notice, that the performance difference between RC and LRC should be greater for systems with write-back caches, since updates to memory require flushing all entries from the cache that correspond to pages in the `WeakList`.

# 3 Trace-Driven Simulation

We use trace-driven simulations to evaluate the proposed VM-based cache coherence methods. In order to compare the VM-based methods with a hardware-based cache coherence mechanism, we selected application programs implemented for snooping bus-based multiprocessors with per-processor caches, and uniformly accessible shared memory, without doing any modification to the algorithms, memory allocation strategies, or data placement.

### Applications

The application suite includes a Gaussian elimination program, matrix multiply, and three programs from the SPLASH parallel applications [SWG91, WG89]. All applications are coded in C, and use the *parmacs* macros from Argonne National Laboratory [LO87] to implement parallel operations such as: creation of cooperating processes, allocation of shared memory, and usage of lock and barrier synchronization variables.

LocusRoute, from SPLASH, is a parallel VLSI standard cell router. The basic strategy of the router is to minimize layout area, and to do so the program evaluates the cost of routing a wire through cells that have the smallest amount of other wires running through them, and finally chooses the route of minimal cost. The work is parallelized by having all wires routed on a task queue, and processors cycle in a loop of removing a wire from the queue and placing it. The task queue is not a bottleneck because each routing task takes a while to complete.

WATER, also from SPLASH, is an $N$-body molecular dynamics application that evaluates forces and potentials in a liquid system of water molecules. These forces and potentials are calculated over a series of time steps separated by barrier synchronization. Work is parallelized by statically partitioning molecules over all processors. These molecules will be close in the data structure, but need not be close in physical space. Since molecules have more interaction with other molecules that are close in space, locality of reference could be compromised if the initial placement of the molecules in the data structures is not appropiate. For the input used in this experiment, molecules are ordered by their $3D$-coordinates, which are uniformly distributed. Locking is used to atomically access to the molecule data structures.

MP3D, the third program from SPLASH, is a 3-dimensional particle simulator for rarefied fluid flow. It simulates the trajectories of fluid molecules as an object moves through the fluid and collides with the

molecules. Over a series of time steps, MP3D evaluates the velocities and positions within a bounded physical space of all molecules. The work is parallelized by partitioning the set of molecules and statically scheduling each set on a processor. The locality of references to the data structures for the physical space cells is low, because of the interference caused by several processors accessing the same space cell during the same time step. The processes use barriers to synchronize between the different phases that make up one time step, and no locking is used on the data structures for the physical space because the probability of data races affecting the result is small.

GE is a Gaussian elimination with partial pivoting on an 80×80 double floating point matrix. Processors are assigned rows to compute statically. Synchronization occurs twice during each pivoting phase: once after the pivot has been determined, and once at the end.

MUL is a program for matrix multiplication of two 80 × 80 double floating point matrices. Each processor computes the results for a statically determined set of rows. The reason for using such a simple program is to have variations in the benchmark suite.

| **Application** | **Shared Addr.** (K) | **Trace Size** (MB) | **Instructions** (x1000) | **Data Refs.** (x1000) | **Distribution of Data References** (%) | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | **Total Reads** | **Shared Refs.** | **Shared Reads** | **Sync. Refs.** |
| LocusRoute | 228 | 101.95 | 43944 | 13370 | 75 | 14 | 11 | .08 |
| WATER | 146 | 391.77 | 244593 | 56250 | 81 | 8 | 7 | .06 |
| MP3D | 205 | 11.59 | 5143 | 1528 | 71 | 36 | 22 | .03 |
| MUL | 150 | 11.8 | 19172 | 1573 | 97 | 65 | 65 | 0 |
| GE | 50 | 5.39 | 7912 | 825 | 64 | 63 | 42 | .07 |

Table 1: Address Trace Characteristics.

## Multiprocessor Traces

For the trace collection we implemented a tool that instruments load and store instructions in MIPS assembly code. We use this tool to annotate not only the application programs, but also the library code. All programs are compiled by the standard `cc` compiler using the `O1` optimization switch.

We did not use *pixie* [Smi91] to generate our traces, because it does not provide a facility to distinguish which Unix process, i.e., virtual processor, generated each traced reference.

To gather the traces, the applications were run on a DEC-5000/200 workstation [Nie90]. Each "virtual processor" corresponds to a Unix process. We used Ultrix primitives `shmget`, `shmctl`, and `shmat` to share memory among processes.

In order to emulate the execution of a shared memory multiprocessor, we used synchronization points to schedule the virtual processors in a round-robin fashion. The synchronization primitives (`lock`, `unlock`, and `barrier`) are all implemented as requests sent to an additional process, called *the lock manager*. Each virtual processor communicates with the lock manager via two pipes, a central request pipe, to which all processors write their requests, and a per virtual processor acknowledgment pipe, to which the lock manager writes to indicate that the virtual processor can continue executing. The lock manager waits for requests and handles them in FIFO order.

Trace collection for each application starts when the collaborating processes are created, and finishes when they all rejoin. For the uniprocessor case, the applications do not contain any synchronization operations.

Table 1 shows the characteristics of the address traces for these applications. The shared memory requirements for the SPLASH applications in the table are based on the data from [SWG91].

### Simulated Architecture

The multiprocessor architecture for our simulation experiments consists of 4 uniprocessor nodes connected to a memory unit via a split transaction bus (see Fig. 6). The main reason to choose a bus instead of another kind of network is to allow us the comparison of the VM-based methods with a snoopy cache scheme.

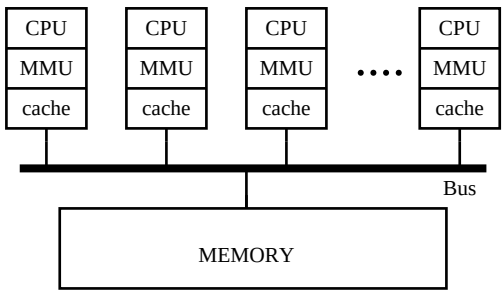


Figure 6: Simulated Architecture

Each node in the system resembles a MIPS R3000 processor, with a 64K direct mapped data cache. Writes are buffered in a 8 entry write buffer. Depending on the cache coherence protocol reads may bypass writes queued in the write buffer. A read operation to an address in the write buffer will not generate a bus transaction. It will return the value stored in the buffer. The memory unit consists of two interleaved memory modules and takes advantage of page mode.

The split transaction bus has separate address and data lines with the following latencies:

- 1 cycle for arbitration,
- 1 cycle for address latching, and
- 1 cycle to send each set of 4 bytes of data.

The virtual memory page size is 1K bytes. The cache line size is 32 bytes. The latency at the memory modules is 6 cycles. Therefore, the round-trip latency for a cache line fill in the absence of bus contention is 16 cycles.

We assume that every machine instruction takes one cycle. Misses on instruction fetches are simulated by introducing accesses to random addresses with probability of one in every 500 instructions. Cache lines are selectively invalidated using the fast invalidation method.

We have implemented a simulator to evaluate each consistency model. The simulator works at the processor cycle level, and consists of a group of light-weight processes, each representing one of the processors being simulated. These processes cycle in a main loop reading from the trace files and executing the operations required to satisfy a trace-entry. Accesses to shared addresses are executed by calling a common set of routines provided by all modules that implement the different coherence schemes.

The synchronization primitives, `lock`, `unlock` and `barrier`, are simulated in the main loop by issuing the corresponding set of accesses to the lock or barrier variables. Synchronization is based on a variant of spinning locks proposed in [And89].

Page fault handling is simulated by the modules that implement the different consistency models. The overhead of each step in the page fault handler was calculated by adding the cost of similar instructions used in the access fault handler of the Ultrix kernel for the DEC-5000/200 workstations.

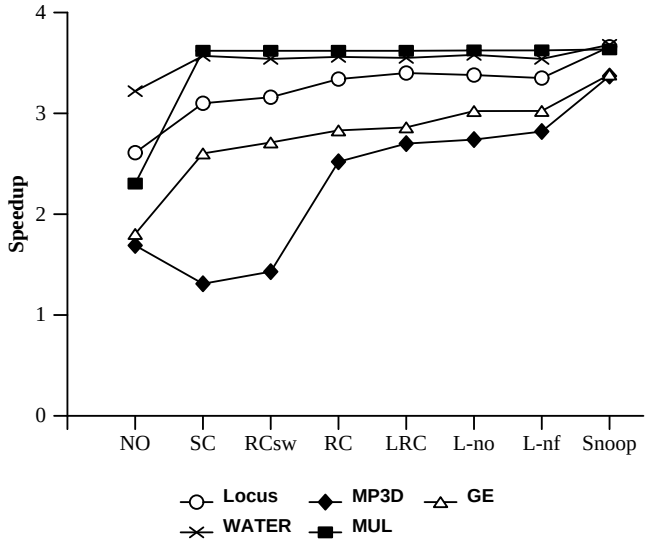


Figure 7: Performance of various consistency models.

# 4 Results

In this section, we present a comparative analysis of the performance achieved by the following cache consistency models:

- **NO** — No caching of shared data.
- **SC** — Sequential consistency model for the VM-based coherence scheme as described in Section 2.
- **RC** — Release consistency model for the VM-based coherence scheme as described in Section 2.
- **RCsw** — Release consistency with a single writer. It is the same as **RC** except that ordinary reads can be in any order with respect to other reads and writes, while writes are sequentially consistent.
- **LRC** — Lazy release consistency model for the VM-based coherence scheme as described in Section 2.
- **Snoop** — The Illinois hardware invalidation-based coherence mechanism for snoopy caches.

All models were simulated using the same architecture parameters.

Two additional versions of LRC are included in the results: **L-no** and **L-nf**. In both these implementations of LRC, synchronization variables are non cacheable, and for each lock variable an additional variable is used to store the previous holder of a lock to identify the `Realease-Acquire` transitions between different processors. L-nf extends L-no with a complete simulation of the `WeakList` management. L-nf is used to validate the cost values used to estimate the overhead of the `WeakList` for all other schemes.

### Speedups

Figure 7 shows the relative performance of all coherence mechanisms for each application. For the purpose of this paper, we use speedups to characterize the performance of the parallel execution of each application with respect to all the evaluated coherence schemes. Speedups were calculated using the results of each applications' uniprocessor simulation. Table 2 shows the exact speedup numbers we generated with our experiments.

| Application | NO | SC | RCsw | RC | LRC | L-no | L-nf | Snoop |
|---|---|---|---|---|---|---|---|---|
| LocusRoute | 2.61 | 3.10 | 3.16 | 3.34 | 3.40 | 3.38 | 3.35 | 3.66 |
| | (71.3%) | (84.7%) | (86.3%) | (91.2%) | (92.8%) | (92.3%) | (91.5%) | (100%) |
| WATER | 3.22 | 3.57 | 3.54 | 3.56 | 3.55 | 3.58 | 3.54 | 3.68 |
| | (87.3%) | (96.8%) | (96.1%) | (96.6%) | (96.3%) | (97.1%) | (96.1%) | (100%) |
| MP3D | 1.69 | 1.31 | 1.43 | 2.52 | 2.70 | 2.74 | 2.82 | 3.37 |
| | (50.1%) | (38.8%) | (42.4%) | (74.7%) | (80.1%) | (81.3%) | (83.6%) | (100%) |
| MUL | 2.30 | 3.62 | 3.62 | 3.62 | 3.62 | 3.62 | 3.62 | 3.63 |
| | (63.4%) | (99.6%) | (99.6%) | (99.6%) | (99.6%) | (99.7%) | (99.7%) | (100%) |
| GE | 1.80 | 2.60 | 2.71 | 2.83 | 2.86 | 3.02 | 3.02 | 3.38 |
| | (53.3%) | (76.9%) | (80.2%) | (83.7%) | (84.6%) | (89.3%) | (89.3%) | (100%) |

Table 2: Speedups.

The percentages shown in Table 2 indicate the relative performance of the VM-based coherence schemes with respect to the snoopy-cache scheme Snoop. Clearly LRC outperforms all other VM-based approaches. Even in the worst cases, i.e., for MP3D and GE, the VM-based LRC remains within 20% of the Illinois hardware snoopy-cache scheme.

A main factor in the performance differences among the VM-based schemes is the variance in their page fault ratios. Table 3 shows these page fault ratios for each application. It is easy to see the correlation between the page faults ratios and the speedups in Figure 7.

| Application | SC | | RCsw | | RC | | LRC | | L-no | L-nf |
|---|---|---|---|---|---|---|---|---|---|---|
| | ord | sync | ord | sync | ord | sync | ord | sync | ord | ord |
| LocusRoute | 2.04 | .04 | 1.05 | .04 | .94 | .03 | .94 | .03 | .91 | .91 |
| WATER | .66 | .30 | .65 | .30 | .68 | .30 | .92 | .30 | .91 | .92 |
| MP3D | 7.08 | .07 | 4.26 | .12 | 1.13 | .11 | 1.14 | .11 | 1.14 | 1.14 |
| MUL | .027 | .003 | .027 | .003 | .027 | .003 | .027 | .003 | .03 | .03 |
| GE | 1.23 | .40 | .56 | .59 | .55 | .46 | .52 | .51 | .50 | 0.50 |

Table 3: Page Fault Rates as % of shared accesses.

### NO versus Others

The most economical and the simplest method of cache coherence is implemented by not caching shared variables, i.e., the no-caching scheme (NO). However, it has poor speedups for all applications.

Moving to other schemes, we can see dramatic performance gains in all applications except for MP3D. There are two reasons for MP3D's behavior: it has low locality of reference, and a high degree of interference between processors accessing the same shared memory locations. The low locality is a result of statically assigning each processor a set of molecules, which over time move into any space cells. The high degree of interference is caused by molecules managed by different processors moving into the same space cells. In addition the granularity of our coherence unit, 1024 Bytes, adds to this factor by introducing high degrees of false sharing. Page faults occur at a rate of up to 1 in every 15 accesses for the sequential consistency mechanism, of which 86% are false misses.

Moving from NO to the VM-based schemes, the performance gains are considerable in all other cases, ranging from 10% to 57%. The application with the lowest performance gain is WATER, because it has the lowest degree of sharing. With an increasing degree of sharing, caching shared data structures becomes more critical. For example, with 14% of the memory references of LocusRoute being shared accesses, the

performance gain is 18%. For applications with an extremely high degree of sharing such as MUL and GE, the performance gains are 44% and 57% respectively.

The performance gains by moving from NO to the VM-based schemes are very sensitive to the memory latency cycle time in the simulation. As the gap between microprocessor speed and DRAM speed continues to increase, we expect the performance gains by moving from NO to the VM-based schemes to increase as well.

### RCsw versus SC

The main difference between RCsw and SC is that RCsw allows a single writer to coexist with multiple readers whereas SC only allows one exclusive writer or multiple readers. This relaxation of consistency has the potential to reduce page faults due to the false sharing.

The reduction in page fault rates is a good measure of whether it is possible to obtain performance gains by moving from SC to RCsw. For GE, for example, 54% of ordinary page faults are eliminated by moving from SC to RCsw, which results in a 4% performance gain over SC.

As shown in Figure 7, moving from SC to RCsw can also result in a small performance loss for some applications. The main reason is that in our implementation of the VM-based schemes we manipulate only one `WeakList`. Hence, our simulation treats data associated with all synchronization variables the same; any `Acquire` will invalidate the access to all pages in the `WeakList`. The performance loss for WATER is a little under 1%. In WATER, locks are used to ensure mutually exclusive access to the data structures associated with 200 different molecules, and an `Acquire` on any of these locks will invalidate the access to all weak pages, even if they have no relation with the lock that is being accessed.

For applications such as MP3D, ordering writes allows for an 40% reduction of the number of ordinary page faults, but with the increased overhead on synchronization accesses, the resulting performance gain obtained by moving from SC to RCsw is only 9%.

### RC versus RCsw and SC

The performance of applications generally improves when moving from RCsw to RC, because RC allows multiple writers and readers to coexist for ordinary pages. Relaxing the ordering of writes can reduce page faults significantly. In MP3D, for example, the page fault reduction is 73%.

For WATER allowing multiple writers to coexist does not completely overcomes the penalty of having only one `WeakList`, but the difference in performance between SC and RC is negligible.

For MP3D, RC performs much better than SC and RCsw; the page fault reduction is significant while the overhead due to page faults on synchronization faults remains low. Table 3 shows that most page faults in SC are due to interference on ordinary accesses, and this is what RC takes advantage of when relaxing the coherence constraints.

GE and MUL do not gain in performance by relaxing the consistency models, because there are few conflicting updates to shared data structures in these applications. GE does not achieve the speedups of MUL, because during each phase when the pivoting row is being established only one processor is doing useful work, and synchronization is required between all accesses.

### LRC versus RC and RCsw

LRC has the best overall performance. But as expected the performance of LRC is similar to RC, the main difference being that RC waits for the write-buffer to drain before a release can perform, while in LRC the write-buffer is drained only if the lock "moves" to another processor.

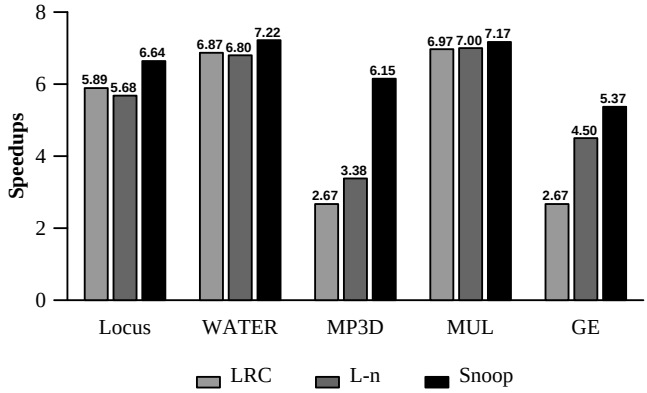


Figure 8: Speedups of LRC versus Snoop. (8 processors)

For LocusRoute this capability allows for a 3% increase in the speedup of the application.

The effect of not caching synchronization variables (L-no) varies. For applications with high contention on locks or barriers, such as GE, not caching these variables avoids ping-ponging the access rights for the page holding the variables. On the other hand, for applications with little contention on locks, like LocusRoute, not caching the variables slightly decreseases performance. But in general in a 4 processor system, the performance of L-no is very similar to LRC.

### Scalability of LRC

Figure 8 shows the speedups of LRC versus Snoop for each program on 8 processors. The speedup gaps between LRC and Snoop are small for WATER and MUL, staying within 5%. Moving from 4 to 8 processors, the speedup gaps for these applications increase at most by an additional 2%. For LocusRoute the gap increases from 8% to 15%, due to more contention on synchronization variable acceses. For MP3D, however, the speedup gap is increased dramatically from 20% to 63%. For GE, the gap is increased from 16% to 50%.

Not caching synchronization variables (L-nf) allows for better speedups for applications like GE, where 77% of all page faults were due to synchronization variable accesses.

Both MP3D and GE are designed for snoopy-cache multiprocessors where the cost of a synchronization is small. [LLJ+91] shows that MP3D structured as used for our simulations has poor speedups on the DASH architecture as well, and that by using a restructured version of MP3D (PSIM4) it is possible to obtain better speedups on the same hardware. Since we are using the original unmodified MP3D, it is not surprising that it has does not scale well for the VM-based LRC model.

Overhead for the VM-based LRC approach comes from ordinary page faults and synchronization. Hence the difference in LRC's performance when moving to an 8-processor system is due to larger costs for page fault handling and synchronization. For MP3D, the total cost of ordinary page faults and synchronization increases by roughly 70% when moving from an 4-processor to an 8-processor system. For GE, although the cost of ordinary page faults remains low in an 8-processor system, the cost of synchronization triples when caching of synchronization variables is allowed. For MUL and WATER, on the other hand, the overhead of ordinary page faults and synchronization for the 4-processor system is low, and although it increases by 100% and 50% respectively when moving to an 8-processor system, it still remains low and has very little effect on the performance of the application. The overhead increase for LocusRoute is only 10%.

Some of the overhead increase for the VM-based coherence schemes, when scaling to a larger system is due to checking more state information, but we believe that most of the overhead comes from bus

| Application | 4 proc | 8 proc |
|---|---|---|
| LocusRoute | .02 | .4 |
| WATER | .01 | .12 |
| MP3D | .2 | 2.38 |
| MUL | .03 | .6 |
| GE | .26 | 1.82 |

Table 4: Arbitration waiting cycles.

contention. For the VM-based coherence schemes, handling page faults requires additional memory bus bandwidth, and for MP3D and GE the bus-based interconnection becomes a bottleneck on an 8-processor system. Table 4 shows the average number of bus arbitration waiting cycles for a memory bus transaction. For GE and MP3D, the average waiting cycles for an arbitration increase by an order of magnitude.

Our simulation results are limited to a particular bus-based architecture. To a certain extent, it is not fair to compare the VM-based schemes with the snoopy-cache scheme because VM-based schemes allow arbitrary interconnection networks. The question of whether using other interconnection networks would affect the performance of the VM-based schemes significantly is still open.

# 5 Conclusions

VM-based software cache coherence schemes are effective and economical approaches to build shared-memory multiprocessors. Our results show that when contention for shared data structures is low, VM-based schemes can perform very well. Under low contention the best approach can perform as well as a snooping scheme. With increasing contention on shared memory accesses a 4 processor system stays within 20% of the snooping scheme.

The simplicity of the architecture for the VM-based schemes has a great advantage over hardware snoopy-cache and directory schemes. It allows one to build shared memory multiprocessors using off-the-shelf uniprocessors and caches, keeping the development cycle to a minimum. Since VM-based schemes are implemented in software, different consistency models can be tested and realized without altering the hardware. In addition, it allows for more efficient use of chip real estate, and for the parallel execution of multiple unrelated programs to take advantage of uniprocessor performance without being encumbered by cache coherence hardware.

Another advantage of the VM-based coherence methods is their flexibility. The shared memory address space of an application can be partitioned with respect to any criterion, and different coherence models can be used to manage each partition. The flexibility allows compilers and programmers to perform optimizations with data placement and coherence strategies easily.

The lazy release consistency model, LRC, overall performs better than the other VM-based schemes: SC, RCsw and RC. This model is easy to implement with write-through caches.

The performance of all relaxed consistency models, RCsw, RC, and LRC can be further optimized. For simplicity, the current implementation maintains a central list of weak pages which may cause unnecessary invalidations. We are in the progress of evaluating further optimized approaches for release consistency where the system maintains a list of weak pages for each synchronization variable.

Our simulations are limited to a particular set of architecture parameters. Many questions remain open. For example, the simulated architecture uses a split-transaction bus as the interconnection for caches and memory. It is not clear to what degree other types of interconnection networks such as meshes or crossbars will improve the performance of the VM-based schemes.

### Acknowledgements

This research was supported in part by NSF grant CCR-9020893, by DARPA and ONR under contracts N00014-91-J-4039, and by Intel Supercomputer Systems Division. Early work on the trace collection was developed during Karin Petersens summer internship at Matsushita Information Technology Laboratory.

## References

[AAHV91] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. Comparison of Hardware and Software Cache Coherence Schemes. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 298–307, May 1991.

[AH90] Sarita V. Adve and Mark D. Hill. Weak Ordering - A new definition. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2–14, May 1990.

[And89] T. E. Anderson. The Performance Implications of Spinwaiting Alternatives for Shared Memory Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II, pages 170–174, August 1989.

[ASHH88] Anant Argarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherency. In *Proccedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, May 1988.

[BF90] Andreas V. Bechtolsheim and Edward H. Frank. Sun's SPARCstation 1: A Workstation for the 1990's. In *Proceedings of the Spring '90 IEEE COMPCON*, pages 184–188, February 1990.

[BMW85] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 782–789, August 1985.

[CV88] Hoichi Cheong and Alexander V. Veidenbaum. A Cache Coherency Scheme With Fast Selective Invalidation. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 299–307, May 1988.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[EGK+85] Jan Edler, Allan Gottieb, Clyde P. Kruskal, Kevin McAuliffe, Larry Rudolph, Marc Snir, Patricia Teller, and James Wilson. Issues Related to MIMD Shared-Memory Computers: The NYU Ultracomputer Approach. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 126–135, June 1985.

[ELM90] Robin Edenfield, Bill Ledbetter, and Ralph McGarity. The 68040 On-Chip Memory Subsystem. In *Proceedings of the IEEE Spring '90 COMPCON*, pages 264–269, February 1990.

[GL91] Mark Greenstreet and Kai Li. Cache Consistency with Dual-Port Memory: A Software Approach. *Presented at the 2nd Workshop on Scalable Shared Memory Multiprocessors*, May 1991.

[GLL+90] Kouroush Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Int89] Intel. *i860 64-Bit Microprocessor - Advance Information.* Intel Corporation, 1989.

[KCZ91] Peter Keleher, Alan L. Cox, and Willie Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. Technical Report TR91-168, Computer Science Department, Rice University, November 1991.

[Lam79] Lesile Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *TOCS*, 7(4):321–359, Nov 1989.

[LLJ+91] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The Stanford DASH Prototype: Logic Overhead and Performance. Technical report, Computer Science Laboratory, Stanford University, December 1991.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The Stanford DASH Prototype: Logic Overhead and Performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[LO87] E. L. Lusk and R. A. Overbeek. Use of Monitors in FORTRAN: A Tutorial on the Barrier, Self-scheduling DO-Loop, and Askfor Monitors. Technical Report ANL-84-51, Rev. 1, Argonne National Laboratory, June 1987.

[LT88] Tom Lovett and Shreekant Thakkar. The Symmetry Multiprocessor System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.

[LYL87] R. L. Lee, P. C. Yew, and D. H. Lawrie. Multiprocessor Cache Design Considerations. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 253–262, June 1987.

[MB89] Sang Lyul Min and Jean-Loup Baer. A Timestamp-based Cache Coherency Scheme. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume I, pages 23–32, August 1989.

[McC84] E. McCreight. The Dragon Computer System: An Early Overview. Technical report, Xerox Corporation, September 1984.

[Nie90] Michael J. K. Nielsen. DECstation 5000 Model 200. In *Proceedings of the IEEE Spring '91 COMPCON*, pages 220–225, February 1990.

[OA89] Susan Owicki and Anant Agarwal. Evaluating the Performance of Software Cache Coherence. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230–242, April 1989.

[PP84] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 348–354, 1984.

[RS85] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Consistency Schemes for MIMD Parallel Processors. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 340–347, June 1985.

[Smi85] Alain J. Smith. CPU Cache Consistency with Software Support and Using 'One Time Identifiers'. In *Proceedings of the Pacific Computer Communications '85*, pages 153–161, 1985.

[Smi91] Michael D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, November 1991.

[SWG91] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical report, Computer Systems Laboratory, Stanford University, April 1991.

[TS87] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of ASPLOS II*, pages 164–172, October 1987.

[Vei86] Alexander V. Veidenbaum. A Compiler-Assisted Cache Coherence Solution for Multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 1029–1036, August 1986.

[WG89] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.