An Evaluation of Multiprocessor Cache Coherence Based on Virtual Memory Support

Karin Petersen Kai Li
Department of Computer Science
Princeton University

Abstract

This paper presents an evaluation of the impact of several architectural parameters on the performance of Virtual Memory (VM) based cache coherence schemes for shared-memory multiprocessors. The VM-based cache coherence schemes use the traditional VM translation hardware on each processor to detect memory access attempts that might leave caches incoherent, and maintain coherence through VM-level system software. The implementation of this class of coherence schemes is flexible and economical: It allows different consistency models, requires no special hardware for multiprocessor cache coherence, and supports arbitrary interconnection networks.

We used trace-driven simulations to evaluate the effect of different architectural parameters on the performance of the VM-based schemes. These parameters include VM page sizes, write-back and write-through caches, memory access latencies, bus and crossbar interconnections, and different cache sizes. Our results show that for appropriate parameters VM-based cache coherence is an economical and practical approach for building shared-memory multiprocessors.

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

1 Introduction

The performance gap between processors and memory has widened in the past. In the last decade, microprocessor speeds have increased at a rate of 50% to 100% each year whereas DRAM speeds have increased at a rate of 10% or less each year [HJ91]. As the performance gap becomes wider, bus-based multiprocessors become more sensitive to available memory bandwidth on the memory bus. This is particularly true for scientific programs that do not exhibit a high degree of locality in data memory references.

Many interconnects can provide higher aggregate memory bandwidth than memory buses. Unfortunately, known interconnects such as crossbars, rings and routing networks, cannot take advantage of multiprocessor support implemented in many high-performance processors because snoopy cache coherence protocols require atomic broadcast capabilities to invalidate or update replicated cache blocks. Without atomic broadcast, the current alternatives are to employ directory-based cache coherence protocols or to use compiler support to insert cache flush and invalidation instructions into application-code to keep caches coherent.

The class of software coherence schemes we investigate can be implemented on very simple multiprocessor architectures based on arbitrary interconnection networks, without requiring hardware directories or compiler support. The virtual memory translation hardware on each processor detects shared accesses that could lead to memory incoherencies, and the page access fault handlers execute the appropriate actions to maintain cache coherence.

Since the VM-based protocols maintain cache coherence at the virtual memory address space level, different consistency models can be used for different pages in the address space. This flexibility allows programmers or compilers to choose among consistency models such as no coherence, release consistency, and sequential consistency for separate data structures.

We used trace-driven simulations to evaluate the performance of the VM-based cache coherence schemes. Our traces were gathered from programs written for snoopy cache multiprocessors without modifications to improve data access patterns.

This paper reports the results for VM-based protocols that implement sequential consistency and lazy release consistency. We investigated several architectural parameters and their effects on the performance of the VM-based schemes. These parameters include VM page sizes, write-back and write-through caches, memory access latency, crossbar networks and cache sizes. We compare the VM-based schemes with the Illinois invalidation-based snoopy-cache protocol [PP84] and a scheme with no support for shared data caching. Our results show that VM-based cache coherence is an economical and practical approach for building shared-memory multiprocessors.

2 VM-based Cache Coherence

The idea of the VM-based cache coherence schemes stems from previous work on shared virtual memory [LH89, CBZ91]. This section describes the basic mechanism required for VM-based cache coherence schemes and the algorithms for implementing two memory consistency models.

2.1 Architectural Framework

VM-based cache coherence schemes can be implemented on very simple multiprocessor architectures using arbitrary interconnection networks, as shown in Figure 1. No snoopy cache or directory controllers are required in the system. The only additional hardware support needed is an interprocessor interrupt mechanism over the network.

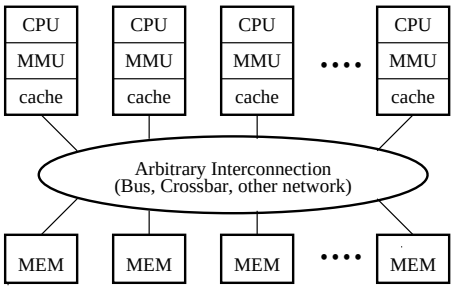

Figure 1: A simple multiprocessor using uniprocessor CPUs, caches, and memory modules.

Memory Management Units (MMUs) are standard components of most uniprocessor architectures. The implementation of the VM-based schemes requires an MMU to support the following, commonly available features:

- At least three different protection states associated with a page: *read-only* (READ), *read/write* (WRITE), and *no access* (NIL).
- Physical caches which support page-based selective *write-through* or *write-back*, and selective invalidation of cached lines from software, as supported by architectures like the the Intel i860, the MC68040 and the MC88020 [Int89, ELM90, Asl90].
- Each processor has its own page table and TLBs can be reloaded from software.

The VM-based software maintains cache coherence at the virtual memory address space level. When a shared address space is created, the system initializes a page table for each processor. In these tables, the entries (PTE's) for a shared page will always have the same virtual-to-physical address translation. The protection bits, on the other hand, may be different for each processor, and will reflect the coherence state of the page. VM software will maintain the page tables at run time according to a chosen cache consistency model.

In all VM-based algorithms we assume private data is written back to memory on cache line replacement. For shared data, we first present each algorithm assuming write-through to keep memory updated. Then we enumerate the changes required to use write-back on both private and shared data.

2.2 Sequential Consistency

Sequential consistency (SC) [Lam79] 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, or multiple readers in the absence of a writer.

Using write-through for shared data, the VM-based implementation of SC is as follows:

Initial state: For each processor set the protection bits of all page table entries (PTEs) for shared pages 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 illustrates this algorithm with a transition diagram for the state of a page. In the diagram, a *remote page fault* means a page fault on a different processor.

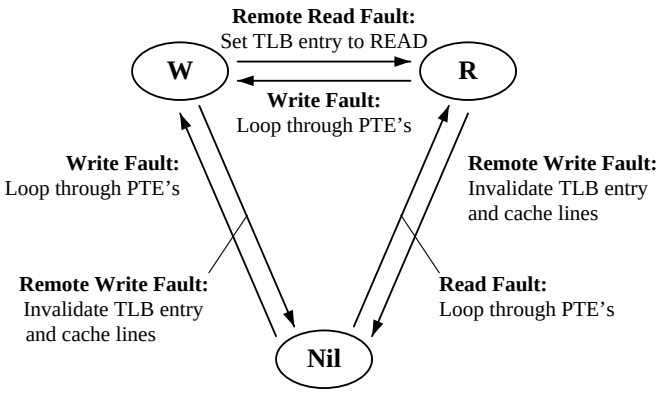

Figure 2: Transition diagram for sequential consistency.

Only few changes to the algorithm are required when write-back is used for shared data structures. With write-back caches, the memory may not have up-to-date data when another processor requests access to a page; our algorithm must therefore 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 may be reduced.

One of the expensive operations of the algorithm described above is the invalidation of cache lines when the protection of a page changes from READ or WRITE to NIL. To loop through all cache lines belonging to a page and invalidating them individually is inefficient. Some cache architectures, such as the MC88200 [Asl90], allow all cached lines of a page to be invalidated with one instruction.

In [Aut] we present a fast software mechanism for page based cache invalidation that can be used in the absence of an explicit instruction. The software mechanism uses aliases for physical addresses and page table management to implement "one-time-identifiers" [Smi85]. In our simulations we assume that either hardware support is provided or the software fast selective invalidation is implemented, and therefore cache lines can be invalidated quickly on a per page basis.

2.3 Lazy Release Consistency

Relaxed consistency models [DSB86, AH90, GLL+90] are based on the following observation: most shared memory multiprocessors guarantee that all memory accesses complete only when all other 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 other variables. Under relaxed memory consistency models, ordinary

accesses can be issued and performed in any order, and synchronization points force memory accesses to be globally performed and therefore return the system to a coherent state. These methods are used to hide the latency of each memory access. In addition to hiding latency, for the VM-based cache coherence schemes, relaxed consistency models also reduce the number of page faults caused by false-sharing.

A hardware implementation of RC is described in [GGH91]. Notice that in software it is possible to take further advantage of the semantics associated with `Acquire` and `Release` accesses. An `Acquire` is used to obtain a lock, and a `Release` frees the lock. Stores to memory that are required by RC to perform with respect to all other processors on a `Release` operation, can be delayed until the next `Acquire` of that lock. This is the main idea behind lazy release consistency (LRC) [KCZ91]. Intuitively, the advantage of LRC over RC is that memory updates need to be performed only when locks change owners.

For our VM-based implementation of LRC, we chose to make synchronization variables non cacheable to avoid the page fault overhead on locks with high contention. Shared pages will therefore either contain ordinary data, and be cacheable, or synchronization variables, and then be non-cacheable. Many commercial processors allow data to be marked cacheable on a per page basis [Kan87, Int89, Inc92].

Synchronization variables are kept sequentially consistent, by ensuring that memory accesses block until completed at the memory modules. For this, in architectures that include a write-buffer, a processor will have to stall following a store to a synchronization variable, until its write-buffer drains. Therefore using a write-through cache, before a `Release` performs, all stores issued previous to the `Release` are guaranteed to have been performed at the memory modules as well.

In LRC, page faults on ordinary data accesses will be used to keep track of how many processors have access (READ or WRITE) to a shared page, and when it might potentially become incoherent, that is, at least one of the processors that has access to the page has also written to it.

In order for a processor to know which pages are being simultaneously shared and updated we define a globally shared list of pages called the `WeakList`. The `WeakList` is used when a processor needs to invalidate or flush the cache lines of shared pages to which it and other processors have access. We also use a counter, $weak_i$, to keep track of the number of processors that have access to page i at any given time.

The algorithm for the VM-based implementation of LRC, using write-through for shared pages, is as follows:

Initial State: For each processor set the protection bits of all page table entries for shared pages to NIL. In addition: `WeakList` = NULL, and $weak_i = 0$ for all i.

Read Fault on page i: 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: 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.

On `Acquire`: 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`.

On `Release`: the write-buffer, if used, has to drain before the `Release` is issued.

In the implementation of the VM-based LRC with write-back for shared data, we use an additional word for each lock, $lastowner_x$, to identify on which processor the lock was last held. The remaining changes in the algorithm occur on `Acquires` and `Releases`:

On `Acquire`: Interrupt $lastowner_x$, requesting it to flush all its updates to pages in the `WeakList`. Then, simultaneously, for all pages $i \in$ `WeakList`, if the processor issuing the Acquire has access to i 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`. Wait for an acknowledgement from $lastowner_x$, indicating that it has completed the flushing procedure. Set $lastowner_x$ to be this processor.

On `Release`: do nothing other than issuing the Release.

3 Performance Evaluation

We use trace-driven simulations to evaluate the proposed VM-based cache coherence methods. There are two main goals in our performance evaluation: compare the VM-based cache coherence approach with the Illinois invalidation-based snoopy-cache protocol [PP84] using a bus-based architecture, and evaluate the impact and implications of different architectural parameters such as VM page sizes, write-back and write-through caches, memory access latency, crossbar networks and cache block sizes.

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. The applications were only modified to allocate synchronization variables on pages separate from other shared data. Other than that, the application's algorithms, memory allocation strategies, and data placement remain the same.

3.1 Applications

The application suite includes a Gaussian elimination program, matrix multiply, and three programs from the SPLASH parallel applications [SWG91]. 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.

GE is a Gaussian elimination with partial pivoting on an 80×80 double floating point matrix. Rows are assigned to processors 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 variation in the benchmark suite.

LocusRoute, from SPLASH, is a parallel VLSI standard cell router. The applications is parallelized by using multiple threads that cycle in a loop of removing a wiring task from a central queue and finding a minimum cost placement for the wire on the layout. WATER, also from SPLASH, is an N-body molecular dynamics application that evaluates forces and potentials in a liquid system of water molecules. Work is parallelized by statically partitioning molecules over all processors. 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. For MP3D the work is also parallelized by partitioning the set of molecules and statically scheduling each set on a processor.

3.2 The Simulator

We implemented a processor cycle level simulator. It consists of a group of light-weight processes, each representing one of the processors being simulated. These processes cycle in a loop reading from the trace files and executing the operations required to satisfy a trace-entry. Accesses to synchronization variables only have one entry in the trace. The simulator keeps track of the values of these variables and inserts additional accesses into the trace in order to handle each synchronization event appropriately.

The VM-based cache coherence algorithms are also simulated by dynamically adding instructions and memory accesses to the traces of each processor. We used the instruction sequence for the page access fault handler in the Ultrix OS Kernel version 4.2 from the DEC-5000/200 workstations to estimate the cost for the instructions in the fault handlers of our algorithms.

Each node in the multiprocessor architecture we simulate resembles a MIPS R3000 processor, with a $64K$ direct mapped data cache. Writes are buffered in an 8 entry write-buffer. Depending on the cache coherence protocol, loads may bypass stores queued in the write-buffer. A load operation to an address in the write-buffer will not generate a bus transaction, but instead return the value stored in the buffer. The memory unit consists of four interleaved memory modules and takes advantage of page mode. Every machine instruction takes one cycle.

The simulator is highly parameterized. For example, the values for page size, cache size, cache block size, memory latency, number of memory modules, and number of processors can be changed easily. The modules that

implement different cache organizations, synchronization, coherency schemes, and interconnection networks can also be interchanged with no effort. This allows for a very flexible evaluation of the impact of architectural issues.

In our first set of experiments the multiprocessor architecture consists of 4 processors connected to a memory unit through a split transaction bus (see Fig. 3). The main reason for choosing a bus initially, rather than another kind of network, is to compare the VM-based methods with a snoopy cache scheme.

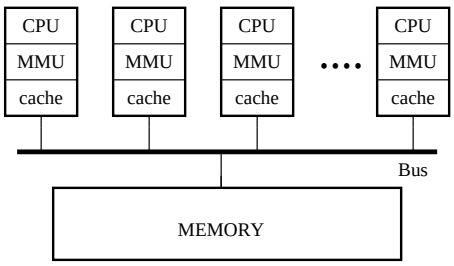

Figure 3: Simulated Architecture

The bus has separate address and data lines. The costs for the phases in a bus transaction are: 1 cycle for arbitration, 2 cycles for address latching, 2 cycles to send 4 bytes of data, 15 cycles latency at the memory modules for the first access to a page, and 2 cycles for subsequent accesses to the same page. The virtual memory page size is 1K bytes. The cache line size is 32 bytes. Therefore, the round-trip latency for a cache line fill in the absence of bus contention is 34 cycles. The scheme used for bus arbitration is very simple, load requests have priority over write-buffer requests, and requests of the same type are served on a first-come-first-served basis. For the VM-based schemes cache lines are invalidated selectively on a per page basis, and write-through is used for updates on shared pages.

3.3 VM-based vs. Snoop and No

Figure 4 shows the relative performance of the following cache coherence protocols:

- **Snoop** — The Illinois hardware invalidation-based coherence mechanism for snoopy caches.
- **NO** — No caching of shared data.
- **SC** — Sequential consistency model for the VM-based coherence scheme as described in Section 2.
- **LRC** — Lazy release consistency model for the VM-based coherence scheme as described in Section 2.

For the purpose of this paper we use normalized execution time to characterize the performance of the VM-based coherence schemes with respect to the hardware snoopy cache approach. In each graph, the execution time for the snoopy scheme represents 100%, and the execution time of all other schemes is normalized to it. In our experiments, execution time is measured as the number of cycles required for all processors to simulate the whole trace for a given coherence mechanism.

We also show a breakdown of the execution time for all applications under each of the evaluated coherence methods. The breakdown is based on the average time one processor spends in each section. Four major regions can be identified in the breakdown, from top to bottom: cycles a processor is busy with the application, cycles due to synchronization or coherence overhead, time spent waiting for memory accesses to complete, and last, cycles a processor is idle due to bus contention.

The most economical and the simplest method of cache coherence is implemented by not caching shared variables (NO). However, it has poor performance for all applications. Comparing NO with the VM-based SC, we can see dramatic performance gains in all applications except for MP3D. The performance gains are considerable, ranging from 20% to 55%. 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 MUL with an extremely large amount shared accesses, the performance gain is 55%.

There are two reasons for MP3D's poor 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

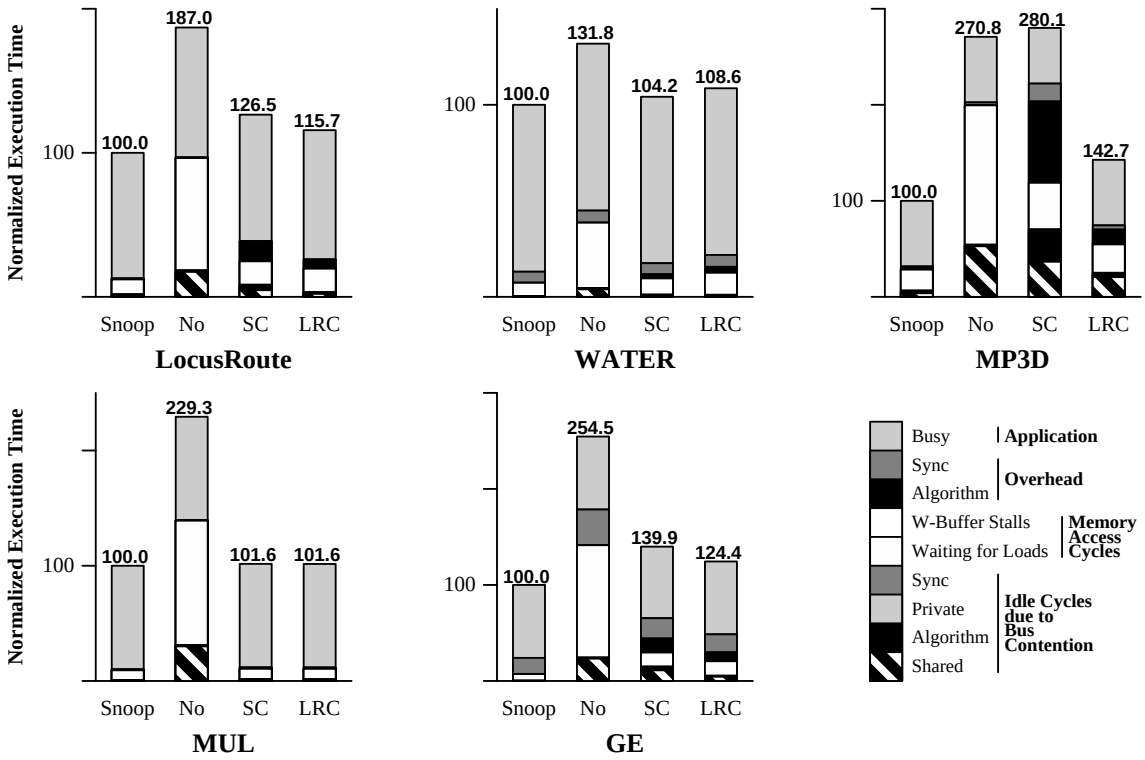

Figure 4: Performance of the coherence schemes for four processors.

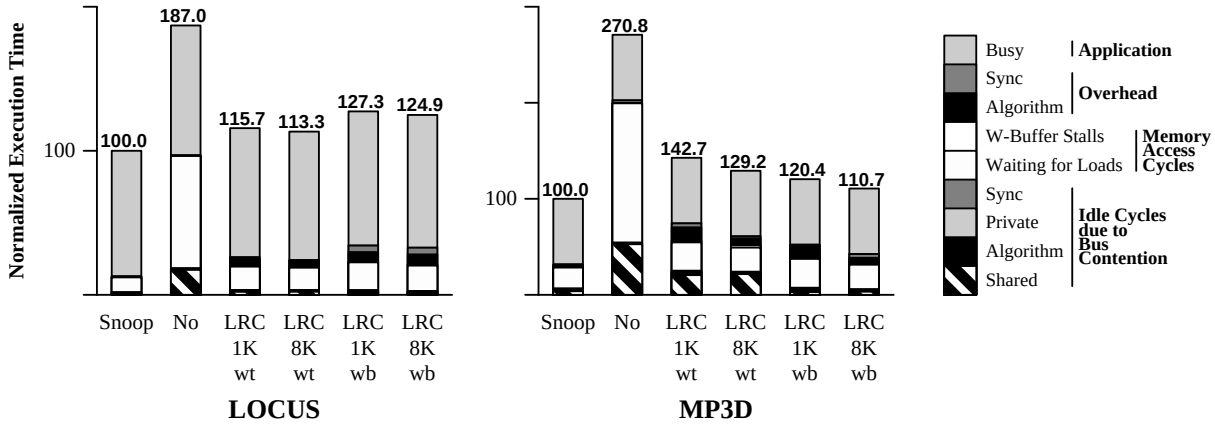

Figure 5: LRC performance with larger page size and write-back caches.

a set of molecules to each processor. In addition, the granularity of our coherence unit, 1024 bytes, adds to the overhead by introducing high degrees of false sharing. Page faults occur at an average rate of 1 in every 15 accesses for the sequential consistency mechanism, of which 62% are false misses.

LRC performs best for almost all applications. The performance gain over SC depends on the degree of contention for shared pages. For applications with very little contention for shared pages, such as MUL and LocusRoute, the difference is less than 10%, but for applications with more contention, the improvement can be substantial, for example, 57% for MP3D. WATER actually performs a little better under SC, because in LRC the number of page faults is only 2% less than in SC, and the increased cost of LRC at synchronization events outweighs the small page fault reduction.

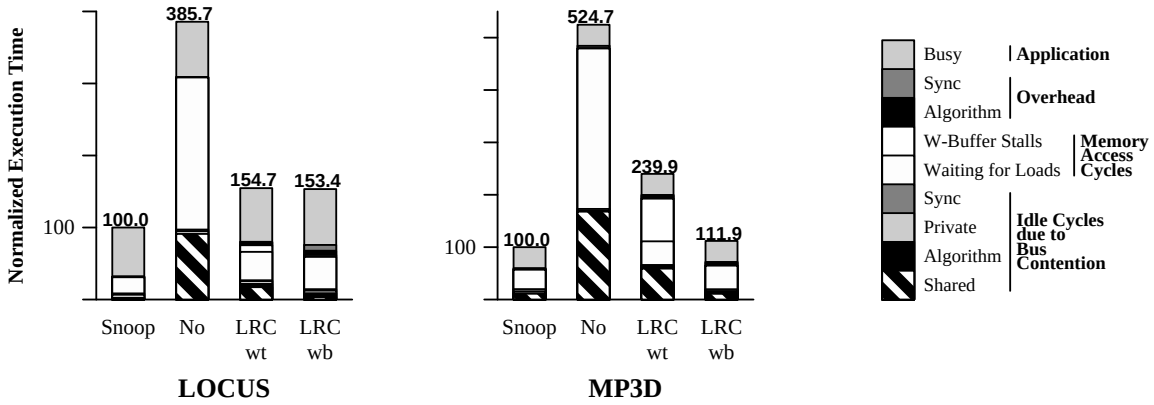

Figure 6: LRC performance with long latency bus interconnections (8 Kbyte pages).

The performance of LRC as compared to Snoop varies depending on the application. MUL performs best, within 2% of the snoopy cache approach. MP3D, however, requires 42% more time to execute under the LRC approach than under Snoop. The execution time breakdown for each application shows two reasons for the difference in performance: the overhead due to the VM-based implementation of LRC, and the time a processor spends waiting for memory loads to complete.

The overhead of the VM-based implementation of LRC is a function of the number of pages simultaneously shared and updated by the processors and the frequency of synchronization events that require coherence actions. For MP3D, which has the worst performance compared to the snoopy approach, the LRC overhead accounts for almost 15% of its execution time. For MUL, on the other hand, it is only .3%.

The other factor that contributes to the difference between LRC and Snoop, is the load on the memory bus. Although the number of bus transactions generated by accesses to private variables is roughly the same in LRC and Snoop, there is a substantial increase in bus activity due to shared accesses in LRC. This increase is only a little over a factor of one for MUL, about eight for MP3D, LocusRoute, and WATER, but as large as 28 for GE. For all applications but MUL, around 90% of all bus accesses to shared variables are due to cache write-throughs, only the remaining 10% is due to cache loads. Some additional bus traffic is also generated by memory accesses due to the LRC algorithm.

A more detailed analysis of the results presented in this section can be found in [PL93].

3.4 Page Size

We selected two applications to study the impact of several architectural parameters on the performance of the VM-based LRC: MP3D and LOCUS. We did not use MUL or WATER, because they behave well under the VM-based schemes, and expect this to continue even with changes in the architecture. The behavior of GE is similar to MP3D, and would therefore not provide additional insight to the analysis.

As mentioned earlier, the overhead of the VM-based implementation of LRC is a function of the number of pages simultaneously shared and updated, or in other words, it depends directly on the length of the `WeakList` when processing `Acquires` or `Releases`. With larger pages, the length of the `WeakList` could potentially decrease and reduce the overhead on synchronization events. On the other hand, the number of page faults can either decrease, because data is accessed from small pages that are part of a larger page, or increase due to more false sharing.

Figure 5 exhibits the effect of changing the page size of the system from 1 Kbyte to 8 Kbytes. The results labeled LRC(1K wt) and LRC(8K wt) show that for both applications larger pages improve performance. Not surprisingly, MP3D shows the larger performance gain, 9%, since most of its pages become shared and updated during one of the phases of its execution. The average number of pages on the `WeakList` is reduced from 30.2 with 1K pages to 9.9 with 8K pages. Because almost all pages become shared and updated the number of page faults is also reduced with larger pages. The average number of page faults is down from 1 fault in every 87 shared data accesses, to 1 fault in every 244 accesses with 8K pages.

For LOCUS the performance difference is only 2%. The average number of pages on the `WeakList` is reduced from 18.7 to 11.1, and the number of page faults is reduced by 15%.

Large page sizes would have increased false sharing prohibitively for the VM-based implementation of SC. However, for LRC the results show that using large pages can reduce the overhead of its VM-based implementation.

3.5 Write-Back Caches

In Section 3.3 we mentioned that for all applications but MUL, around 90% of all bus accesses to shared variables are due to cache write-throughs, only the remaining 10% is due to cache loads. This represents a significant load on the bus.

Figure 5 shows that using write-back for shared data improves the performance of MP3D, but deteriorates the performance of LOCUS. The advantage of using write-back to update physical memory is that it reduces the load on the interconnection network. For LOCUS, however, writing shared data through did not saturate the bus. Only 2.8% of LOCUS' execution time is spent idle waiting for the bus. By using write-back for shared data, the time idle due to bus contention is reduced to 2.5%. On the other hand, the additional overhead of requesting cache flushes when a lock changes owners increases the algorithm overhead from 5.5% to 8.9%, therefore worsening the overall performance of LOCUS by 10%.

For MP3D the bus contention generated by using write-through is substantial. MP3D spends 17.5% of its execution time idle, waiting for the bus. This allows write-back for shared data to improve performance by 15%. With write-back only 6% of the execution time is spent waiting for the bus, and the algorithm overhead does not increase at all. The overhead remains the same because caches can be flushed during the time a processor was originally idle waiting for a barrier to complete.

Using write-back for shared-data is advisable for applications that update shared variables frequently, but synchronize relatively few times.

The combined effect of using both write-back for shared data and larger pages further improves the performance or MP3D, but still compares poorly with the write-through approach for LOCUS.

3.6 Memory Latency

The performance gain of moving from NO to the VM-based schemes is 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 gain from changing to the VM-based schemes, instead of NO, to increase as well.

To evaluate the effect of memory latency, we ran the simulator using the following costs in a bus transaction: 2 cycles for arbitration, 4 cycles for address latching, 4 cycles to send 4 bytes of data, 60 cycles latency at the memory modules for the first access to a page, and 4 cycles for subsequent accesses to the same page. A cache line fill in the absence of contention takes 98 processor cycles, almost three times longer than with the original parameters. Figure 6 shows that that under the VM-based LRC schemes both LOCUS and MP3D perform in less than half the time taken by the non-caching scheme (NO).

LOCUS performs within 55% of Snoop, and about half of its execution time is spent waiting for memory accesses to complete. In turn half of the cycles spent on memory accesses are idle due to bus contention. For MP3D with write-through on shared data the time idle due to bus contention is much larger, corresponding to 61% of total execution time. The idle time is reduced to 17% by decreasing the load on the bus with write-back for shared data structures. Using write-back MP3D performs within 11% of Snoop. Notice that for MP3D, Snoop also spends about 20% of its cycles idle due to bus contention.

Although the performance of LRC with respect to Snoop becomes slightly worse with longer memory latencies, it is much better than using no caching for shared data.

3.7 Crossbar Interconnections

As we saw above and in Section 3.3 the load on the memory bus greatly affects the performance of the VM-based LRC scheme. One of the main advantages of the VM-based schemes it that they are not bound to a specific interconnection

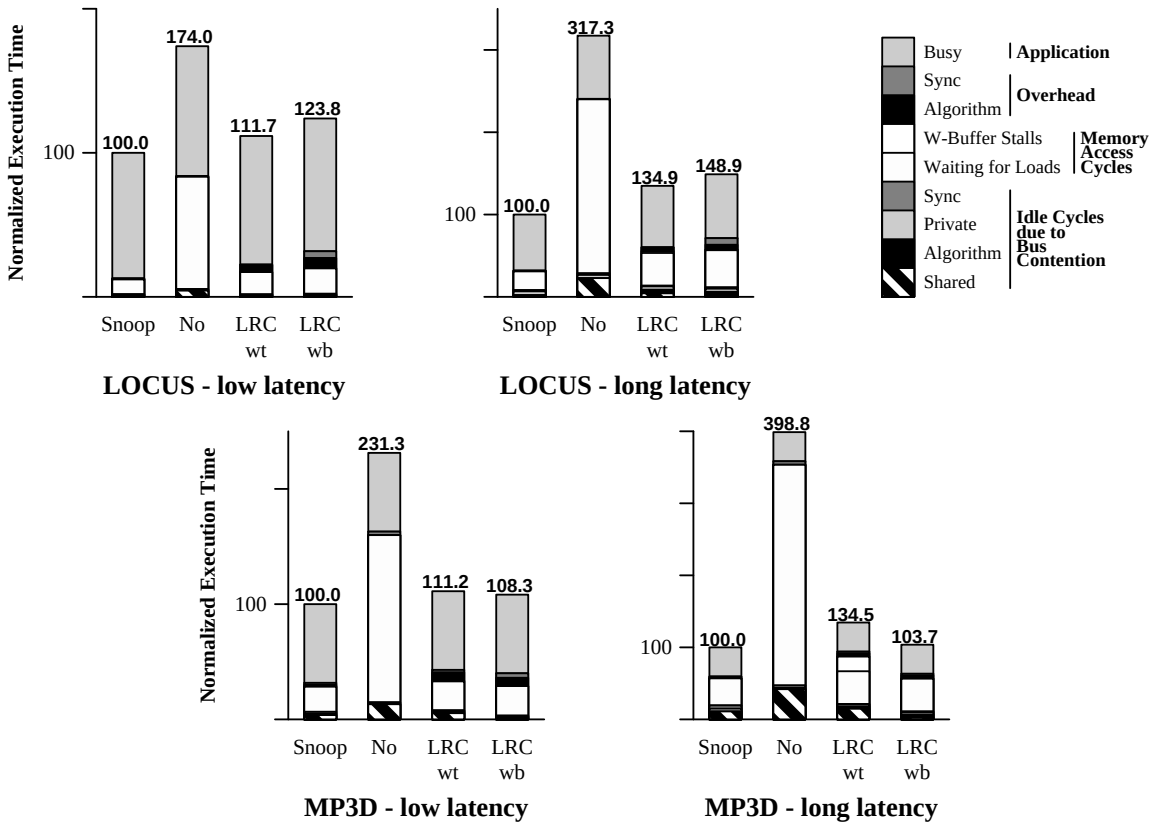

Figure 7: LRC performance with low contention interconnections (8 Kbyte pages).

topology. Any interconnection network with intrinsically lower contention, like a crossbar, could be used to connect processors to memory modules.

Figure 7 shows a performance estimate of using a lower contention network for all models but Snoop. This estimate was based on the assumption that bus load due to shared accesses can be evenly distributed on four disjoint connections to each of the four memory modules. Contention on the interconnection due to private accesses or accesses to data structures used in the VM-based algorithm is not changed. Results are shown for NO, both the write-through and write-back based LRC implementations, and for both previously evaluated memory latencies.

The graphs show that both applications perform better on low contention interconnects. With low memory latencies, the write-through implementation improves the performance of MP3D by 13%. With write-back there is also some improvement, but only by 2%. The effect of low contention interconnects for large memory latencies is, as expected, even more noticeable, 43% for write-through and 7% for write-back.

For LOCUS the improvements with write-through are 4% for short memory latencies, and 12% for long memory latencies. The impact on the write-back scheme is negligible, since the original contention for the bus was fairly low.

Notice that the no-caching approach also benefits from low contention interconnects. The execution time for LOCUS and MP3D is reduced by up to 17% and 24% respectively. However, the VM-based LRC schemes perform at least 1.5 and 2.1 times better than NO for these applications.

3.8 Cache Size

Since trace driven simulation is considerably slower than the real architectures being simulated, only small data sets can be used when generating the address traces. Small data sets cause few capacity cache misses and therefore don't accurately portray the behavior of applications with realistic data sets. In [GHG+91] caches are scaled down to obtain a more accurate ratio between problem size and cache size. Figure 8 shows the results of the simulations using 4 Kbyte data caches and a low contention interconnection.

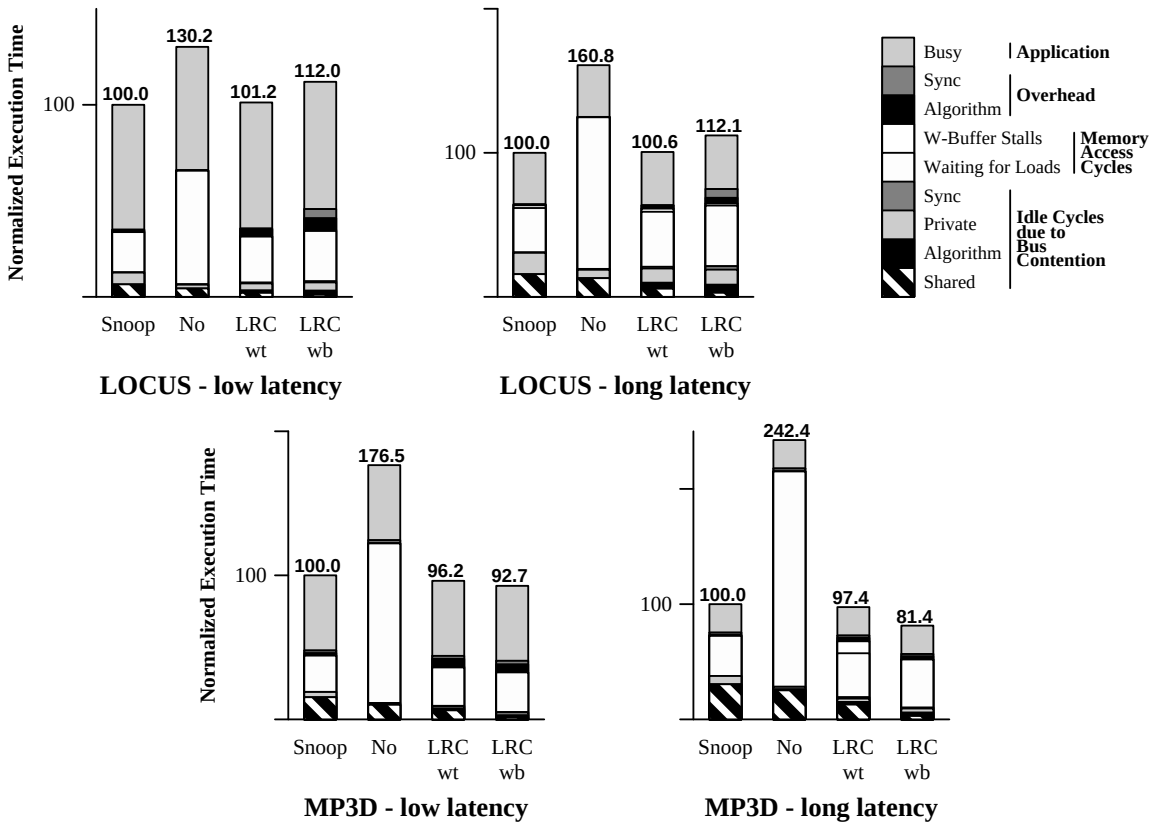

Figure 8: LRC performance with 4K data caches (8 Kbyte pages and low contention interconnect).

In the experiments with 4 Kbyte caches, the miss ratios increase an average factor of 4.5 for LOCUS and 1.6 for MP3D. For the snoopy-cache architecture, the cache miss increment is caused mostly by accesses to shared data. For example the miss ratio on shared data for LOCUS increases by a factor of 4.13, while the misses on private accesses only increase by a factor of 3. Under the VM-based LRC the cache misses on private and shared data increase by about the same factor.

Although MP3D's cache miss ratios are very similar for the VM-based implementation of LRC and Snoop, it performs better under the VM-based LRC than under Snoop because the time spent idle due to contention on the network is higher for the bus required in architectures with snoopy caches.

For LOCUS the cache miss ratios are also very similar, but the bus contention for Snoop is much smaller than with MP3D, and therefore LOCUS performs about the same under Snoop and write-through based LRC.

When most of an application's working set fits in the cache, misses occur mainly due to coherence constraints. In this case, if the cost of a coherence miss is high, such as in the VM-based schemes, the relative performance of the system is lower as compared to an optimized hardware approach. When, on the other hand, cache misses also occur regularly due to capacity limitations, which are less expensive on architectures with low contention interconnection networks, the performance gap between the hardware and VM-based approaches closes significantly.

3.9 Scalability of LRC

Figure 9 shows the performance of VM-based LRC on an architecture with 4 Kbyte data caches, a low latency, low contention interconnection, and eight rather than four processors.

In the graph the four processor execution time for the snoopy scheme represents 100%, and the execution time of all other schemes is normalized to it. The results show that the VM-based implementations of LRC slightly loose competitiveness against Snoop. With eight processors the write-back LRC version of MP3D performs within 6% of

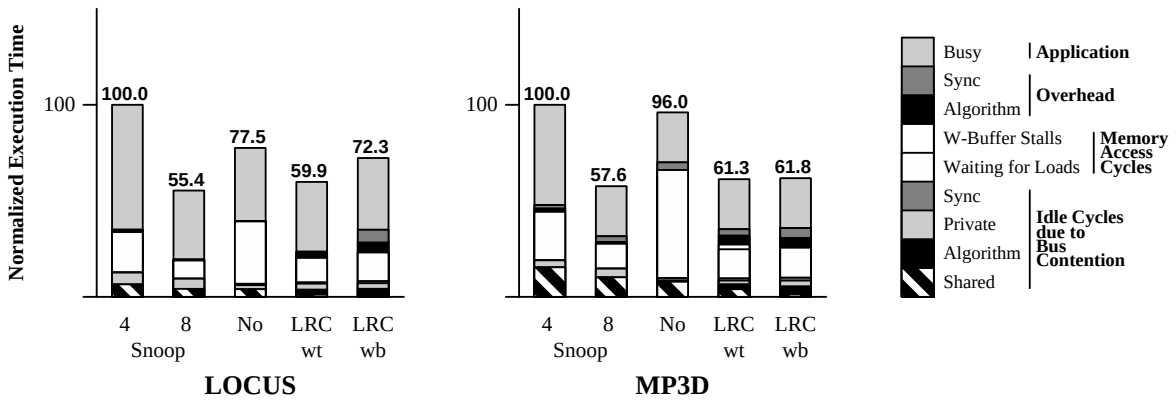

Figure 9: LRC for an eight processor system (8K pages, 4K cache, low latency and low contention interconnect).

Snoop, instead of −3.8% in the four processor case (see Figure 8). The performance of LOCUS for the write-through LRC stays within 8% of Snoop in an eight processor system, having been within 1% on four processors.

The performance of LOCUS scales by a factor of 1.8 on the snoopy-cache architecture, and 1.68 for the VM-based write-through LRC system. For MP3D, Snoop scales by 1.73 and the write-back LRC by 1.56.

4 Related Work

The most common hardware-based solutions for the cache coherence problem use either snoopy caches [PP84, RS85, TS87, LT88] or directories [Tan76, ASHH88, LLJ+92]. Snoopy-cache approaches require bus-based interconnections and sophisticated cache controllers that listen to all transaction on the bus and execute the actions required by the protocol to keep caches coherent.

Directory-based schemes use directory entries for each shared memory block to keep track of all copies in processor caches. A hardware directory controller implements a state machine that responds to access requests for a memory block by adding, invalidating or updating copies as needed. Since directory-based schemes allow arbitrary interconnection networks, they scale better than the bus-based snoopy cache schemes. In general, hardware-based protocols require special purpose hardware that is difficult to design, requires additional chip real estate, and might be restricted to bus-based interconnections.

Several cache-coherence schemes based on compiler support have been proposed [Vei86, LYL87, CV88, Che92, DMCK92]. An important advantage of these software-based methods is that they need little hardware support: only a few instructions used by the compilers to invalidate and flush the cache. Previous research [OA89, AAHV91] 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. However, compiler-based solutions have to be conservative when determining the interference of shared memory accesses, and often require special hardware support for performance.

The class of software coherence schemes we analyzed in this paper differs from all the above in that it requires no special hardware or compiler support in its implementation. The algorithms used for the VM-based schemes are similar to implementations of shared virtual address spaces on distributed memory multicomputers [LH89, CBZ91, KCZ91].

5 Conclusions

VM-based software cache coherence schemes are effective and economical approaches for building simple shared-memory multiprocessors. Our results show that the VM-based schemes can perform very well.

The VM-based cache coherence schemes can be implemented easily using write-through on cache lines containing shared data. However, the performance of the VM-based schemes is sensitive to the contention on the interconnection network. Using write-through for shared data can overload the interconnect if an application updates its shared data frequently. Write-back for shared data allows for better performance in these cases, even though it increases the synchronization overhead of the VM-based approach.

Although large pages are prohibitively expensive for the VM-based implementation of sequential consistency, they improve the performance of VM-based LRC.

The simplicity of the architecture for the VM-based schemes has several advantages over hardware snoopy-cache and directory schemes. It allows shared memory multiprocessors to be built with off-the-shelf uniprocessors and caches, keeping the development cycle to a minimum and allowing for more efficient use of chip real estate. However, the more significant advantage is that arbitrary interconnection networks such as crossbars can be used to connect processors to memory modules. This advantage becomes critical when the latency for memory accesses is high. Bus interconnections saturate very quickly when transactions take a long time to complete. Interconnections such as crossbars alleviate the contention problem and our results show that the VM-based coherence schemes perform very well in this case.

Our results also show that applications with large working sets also perform well under the VM-based LRC. The performance relative to snoopy-cache architectures decreases, when most of the data fits in the cache and synchronization events are rare.

Another advantage of the VM-based schemes is that different consistency models can be tested and realized without altering the hardware, since they are completely implemented in software. The shared memory address space of parallel applications can be partitioned with respect to any criterion, and different coherence models can be used to manage each partition. This flexibility allows compilers and programmers to perform optimizations with data placement and coherence strategies easily. Furthermore, the parallel execution of multiple unrelated programs can take advantage of full uniprocessor performance without being encumbered by cache coherence hardware.

Although we have investigated the performance impact of many architectural parameters such as VM page sizes, memory latencies, crossbar interconnects, and cache sizes, our study is based on a simple architecture that has a single-level cache. The VM-based cache coherence approach can be easily applied to two-level cache architectures. We are currently looking into the implications of two-level cache memory hierarchies.

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.

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.

[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.

[Asl90] Mich Aslup. Motorola's 88000 Family Architecture. *IEEE Micro*, pages 48–66, June 1990.

[Aut] Same Authors. Omitted for anonymity.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 153–164, 1991.

[Che92] Hoichi Cheong. Life Span Strategy - A Compiler-Based Approach to Cache Coherence. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 139–148, July 1992.

[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.

[DMCK92] Ervan Darnell, John M. Mellor-Crummey, and Ken Kennedy. Automatic Software Cache Coherence through Vectorization. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 129–138, July 1992.

[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.

[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.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the 3rd Internatinal Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[GHG+91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, 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.

[HJ91] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[Inc92] SPARC International Inc. *The SPARC Architecture Manual.* Prentice Hall, 1992.

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

[Kan87] Gerry Kane. *mips R2000 RISC Architecture.* Prentice Hall, 1987.

[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] Leslie 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+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.

[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.

[PL93] Karin Petersen and Kai Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. To appear in Proceedings of the 7th International Parallel Processing Symposium, April 1993.

[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.

[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.

[Tan76] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings National Computer Conference*, pages 749–753, June 1976.

[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.