AD-A281 628

Software Cache Coherence for Large Scale Multiprocessors

Leonidas I. Kontothanassis and Michael L. Scott

Technical Report 513
March 1994

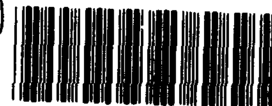
DTIC ELECTE JUL 13 1994 S B D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

94 7 12 056

9421300



REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
March 1994

3. REPORT TYPE AND DATES COVERED
technical report

4. TITLE AND SUBTITLE
Software Cache Coherence in Large Scale Multiprocessors

5. FUNDING NUMBERS
N00014-92-J-1801
ARPA Order No. 8930

6. AUTHOR(S)
Leonidas I. Kontothanassis and Michael L. Scott

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester, NY 14627-0226

8. PERFORMING ORGANIZATION REPORT NUMBER
TR 513 .

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Information Systems
Arlington, VA 22217

ARPA
3701 N Fairfax Drive
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
(see title page)

14. SUBJECT TERMS
software coherence; cache; adaptive protocol; program modifications

15. NUMBER OF PAGES
17

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

Software Cache Coherence for Large Scale Multiprocessors

Leonidas I. Kontothanassis and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{kthanasi,scott}@cs.rochester.edu

March 1994

Abstract

Shared memory provides an attractive and intuitive programming model that makes good use of programmer time and effort. Shared memory however requires a coherence mechanism to allow caching for performance and to ensure that processors do not use stale data in their caches. We evaluate several algorithmic and architectural alternatives in the design space of NCC-NUMA[1] machines with a globally-accessible physical address space. We present a new adaptive algorithm for software cache coherence that reduces interprocessor communication and scales to large numbers of processors; we compare it to existing software and hardware coherence schemes. We also evaluate (1) the tradeoffs among various write policies (write-through, write-back, write-through with a write-collect buffer) and (2) the effect on performance of using remote memory access. Finally, we observe that certain simple program changes can greatly improve performance. For example, we find that the use of reader-writer locks, synchronization variable relocation, and data structure padding and alignment can allow a protocol to avoid significant amounts of coherence overhead.

This work was supported in part by NSF Institutional Infrastructure grant no. CDA-8822724 and ONR research grant no. N00014-92-J-1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

[1]NCC-NUMA stands for non cache coherent, non uniform memory access.

1 Introduction

Large scale multiprocessors can provide the computational power needed for some of the larger problems of science and engineering today. Shared memory provides a comfortable programming model that makes good use of programmer time and can yield good performance for programs with high locality of reference. Unfortunately the introduction of caching to reduce memory latencies also introduces the coherence problem—the need to ensure that processors do not use stale data in their caches. For large scale multiprocessors with point to point networks, directory based coherence seems to be the hardware alternative of choice [1, 19], but can be expensive both in terms of hardware cost and in terms of the design time and intellectual effort required to produce a correct, efficient implementation. Software coherence protocols provide an attractive alternative, especially with the advent of relaxed models of memory consistency. Such models help mitigate the false sharing introduced by larger coherence blocks (pages instead of lines).

In this paper we present an adaptive algorithm for software cache coherence based on the notion of lazy release consistency [15], but targeted for architectures with non-coherent caches and a globally accessible physical address space. Machines in this class include the Cray T3D and the BBN TC2000. We believe such machines to be crucial for the future of high-performance computing: they can be built from off-the-shelf parts, and can follow improvements in microprocessors and other hardware technologies closely. Our work focuses primarily on *behavior-driven coherence*—policies that move and replicate data in response to observed patterns of program behavior—as opposed to compiler-based techniques.

There are at least two reasons to hope that software coherence protocols may be competitive with hardware coherence. First, trap-handling overhead is not very large in comparison to remote communication latencies, and will become even smaller as processor improvements continue to outstrip network improvements. Second, software may be able to embody protocols that are too complicated to implement reliably in hardware at acceptable cost. (Our algorithm, for example, delays forwarding write notices until the next release of a synchronization variable, and we are considering future enhancements that will perform work in the background during synchronization waits.)

We present our algorithm in section 2. After describing experimental methodology in section 3, we turn to performance results. We compare our adaptive algorithm to several known alternatives, including sequentially-consistent hardware, release-consistent hardware, sequentially-consistent software, and a relaxed-consistency software scheme due to Karin Petersen [20, 21]. We find substantial improvements with respect to the other software schemes, enough in most cases to bring software cache coherence within sight of the hardware alternatives. We also report on the impact of several architectural alternatives on the effectiveness of software coherence. These alternatives include the choice of write policy (write-through, write-back, write-through with write-collect buffer) and the availability of a remote reference facility, which allows a processor to choose to access data directly in a remote location, without creating a local copy. Finally, to obtain the full benefit of software coherence, we observe that minor program changes can be crucial. In particular, we identify the need to employ reader-writer locks, avoid certain interactions between program synchronization and the coherence protocol, and align data structures with page boundaries whenever possible.

2 An Adaptive Protocol for Relaxed Consistency Software Coherence

In this section we present a scalable algorithm for software cache coherence. The algorithm was inspired by Karin Petersen's thesis work with Kai Li [20, 21]. Like most behavior-driven software coherence protocols, Petersen's algorithm relies on address translation hardware, and therefore uses pages as its unit of coherence. The key idea is to maintain a centralized *weak list* (in memory) that identifies all pages for which there are multiple writers, or a writer and more than one reader—in other words, for which there are likely to be inconsistent copies. On an acquire operation, a processor uses non-cached references to scan the weak list, and then purges all lines of all weak pages from its cache. The algorithm was designed for use on small-scale bus-based multiprocessors without hardware coherence, and has been shown to work well for several applications on such machines.

Unfortunately the mechanism used to propagate write notices (a centralized weak list) poses serious obstacles to scalability: the size of the weak list and consequently the amount of work that needs to be done at synchronization points increases with the size of the machine. Moreover the frequency of references to each element of the weak list also increases with the size of the machine, implying the potential for serious memory contention. Our goal has been to achieve scalability by designing an algorithm whose overhead is a function of the degree of sharing and not of the size of the machine. Since previous studies have shown that the degree of sharing for coherence blocks remains relatively constant when the size of the machine increases [9], an algorithm with the above property should scale nicely to larger numbers of processors.

It would be tempting to maintain a list on each processor of the weak pages cached by that processor. When releasing a synchronization variable, a processor would post notices to each list separately. This approach would eliminate both the problem of centralization (i.e. memory contention) and the need for processors to do unnecessary work at acquire points (checking entries in which they have no interest). However it would make releases considerably more expensive since a potentially large number of remote memory operations might have to be performed. Our goal is to maintain the low acquire overhead of per-processor weak lists while causing only a constant amount of work per shared page on a release.

Our solution assumes a distributed, non-replicated page table data structure that maintains cacheability and sharing information, similar to the coherent map data structure of PLATINUM [12]. Each processor has its own local page table and must make sure that it is consistent with the coherent map whenever the processor performs an acquire operation. Pages in the coherent map can be in one of four states:

Uncached – No processor has a mapping to this page. This is the initial state for all pages.

Shared – One or more processors have read-only mappings to this page.

Dirty – A single processor has both read and write mappings to the page.

Weak – Two or more processors have mappings to the page and at least one has both read and write mappings to it.

When a processor takes a read fault on an already dirty page or a write fault on a shared page, it marks the coherent map entry to indicate that the page has entered the weak state. In the

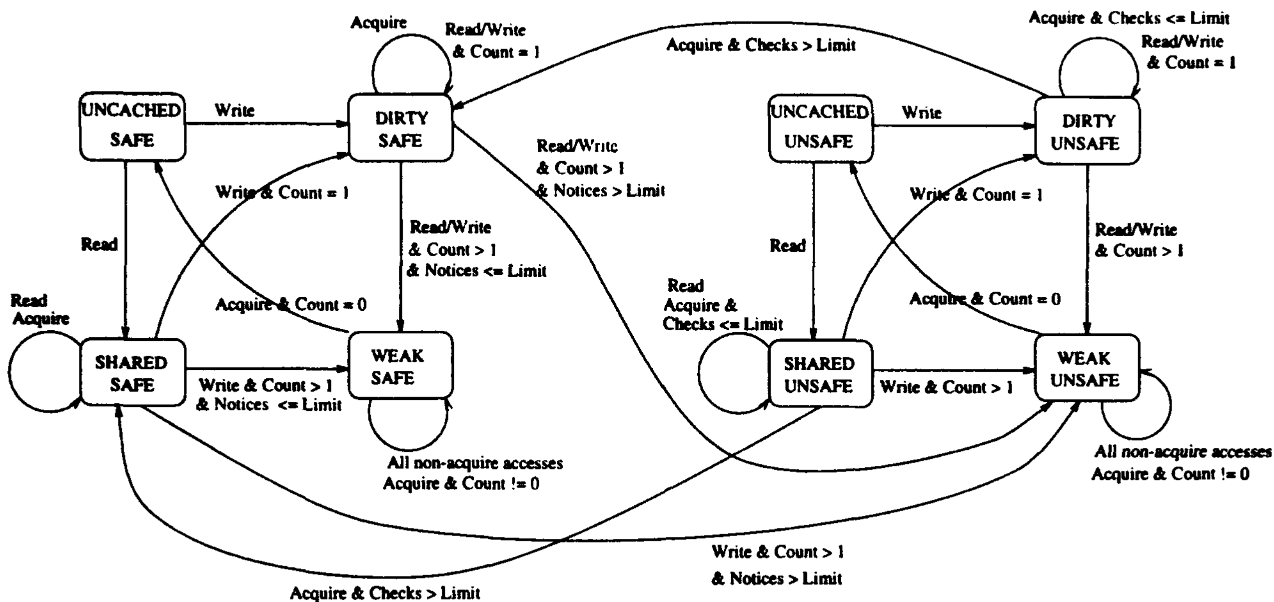


Figure 1: Adaptive software cache coherence state diagram

simplest case, it also sends write notices to all processors sharing the page (i.e. those with an active translation). When a processor reaches an acquire point it must invalidate all weak pages for which it currently has mappings. (Before mapping that page in the future, it must also purge the page's lines from its cache.) When a processor reaches a release point, it must ensure that all dirty lines have been flushed from its cache, either by writing back (with a write-back cache), or by waiting for all previous write-throughs to reach their destinations.

We apply two important optimizations to this basic protocol. First, we consider delaying the point in time at which the transition to the weak state occurs. A processor that introduces the potential for inconsistent copies can change the coherent map and post write notices to all sharing processors immediately, or it can wait until the subsequent release. Waiting for the release has the potential to slow things down by lengthening the critical path of the computation (especially for barriers, in which many processors may want to post notices for the same page at roughly the same time). Our experiments, however (see section 4), indicate that delayed transitions are generally a win. They reduce the number of invalidations done at acquire operations, especially for applications with false sharing.

Our second optimization takes advantage of the fact that page behavior is likely to remain constant for the execution of the program, or at least a large portion of it. We introduce an additional pair of page states, called **Safe** and **Unsafe**. These new states, which are orthogonal to the others (for a total of 8 disjoint states), reflect the past behavior of the page. A page that has made the transition to weak repeatedly is marked as `Unsafe` and no longer requires that its transition to the weak state be accompanied by the sending of write notices. Instead the processor that causes the transition to the weak state changes the coherent map entry to reflect the change and continues. The acquire part of the protocol now requires that the acquiring processor check the coherent map entry for all its pages that are `Unsafe` and invalidate the ones that are also marked as weak. The

Availability Codes
Dist | Avail and/or Special
A-1

state diagram for our protocol without the delayed notice option appears in figure 1. The diagram represents the state of a page in the system as a whole (as opposed to its state on a single processor), together with the transitions on read, write and acquire accesses on behalf of the various processors. `Count` is the number of processors having mappings to the page; `notices` is the number of notices that have been sent on behalf of a `Safe` page; and `checks` is the number of times the coherent map has been incorrectly checked on behalf of an `Unsafe` page.

In the case where an `Unsafe` page is found to be weak, the processor performing the acquire operation must access the coherent map entry on the home node anyway, to indicate that it is no longer sharing the page. Reading the entry first, to verify that the page is weak, increases overhead by only a small constant factor. In the case where an `Unsafe` page is found not to be weak, the access to the directory is truly wasted work, since the processor performing the acquire does not need to invalidate its mapping to the page. To guard against this waste, our policy switches a page back to `Safe` after a small number of unnecessary checks of the coherent map.

Our protocol resembles the implementation of release consistency in Munin [8], in that processors issue write notices at the time of a synchronization release operation. It resembles the implementation of lazy release consistency in TreadMarks [16] in that the actual coherence operations are performed by processors at the time of an acquire, by perusing received write notices. It differs from these proposals in its exploitation of the global physical address space, its use of a weak list, and its notion of safe and unsafe pages. The global address space eliminates the need to broadcast data to processors. Writing back to home nodes suffices; processors retrieve data from there via the normal cache fill mechanism. The weak list, as introduced by Petersen and Li, moves most of the responsibility for write notices from the releasing to the acquiring processors. It is a form of lazy evaluation, and saves work when a modified block is not used by all potential readers before being modified again. The notion of safe and unsafe pages, new to our work, allows processors to skip coherence actions altogether in common cases: acquiring processors must query home nodes only for blocks that have recently been actively shared.

3 Experimental Methodology

We use execution driven simulation to simulate a mesh-connected multiprocessor with up to 64 nodes. Our simulator consists of two parts: a front end, Mint [24, 25], which simulates the execution of the processors, and a back end that simulates the memory system. The front end calls the back end on every data reference (instruction fetches are assumed to always be cache hits). The back end decides which processors block waiting for memory and which continue execution. Since the decision is made on-line, the back end affects the timing of the front end, so that the interleaving of instructions across processors depends on the behavior of the memory system. This is more accurate than trace-driven simulation, in which the order of events is predetermined (recorded in the trace).

The front end is the same in all our experiments. It implements the MIPS II instruction set. Interchangeable modules in the back end allow us to explore the design space of software and hardware coherence. Our hardware-coherent modules are quite detailed, with finite-size caches, full protocol emulation, distance-dependent network delays, and memory access costs (including memory contention). Our simulator is capable of capturing contention within the network, but only at a substantial cost in execution time; the results reported here model network contention at the sending

System Constant Name	Default Value
TLB size	128 entries
TLB fill time	24 cycles
Interrupt cost	140 cycles
Page table modification	160 cycles
Memory response time	20 cycles/cache line
Page size	4K bytes
Total cache per processor	128K bytes
Cache line size	32 bytes
Network path width	16 bits (bidirectional)
Link latency	2 cycles
Wire latency	1 cycle
Directory lookup cost	10 cycles
Cache purge time	1 cycle/line

Table 1: Default values for system parameters

and receiving nodes of a message, but not at the nodes in-between. Our software-coherent modules add a detailed simulation of TLB behavior, since it is the protection mechanism used for coherence and can be crucial to performance. To avoid the complexities of instruction-level simulation of interrupt handlers, we assume a constant overhead for page faults. Table 1 summarizes the default parameters used in our simulations, which are in agreement with those published in [3] and in several hardware manuals.

3.1 Workload

We report results for six parallel programs. Three are best described as computational kernels: **Gauss**, **sor**, and **fft**. Three are complete applications: **mp3d**, **water**, and **appbt**. The kernels are local creations. **Gauss** performs Gaussian elimination without pivoting on a 448×448 matrix. **Sor** computes the steady state temperature of a metal sheet using a banded parallelization of red-black successive overrelaxation on a 640×640 grid. **Fft** computes an one-dimensional FFT on a 65536-element array of complex numbers, using the algorithm described in [2].

Mp3d and **water** are part of the SPLASH suite [23]. **Mp3d** is a wind-tunnel airflow simulation. We simulated 40000 particles for 10 steps in our studies. **Water** is a molecular dynamics simulation computing inter- and intra-molecule forces for a set of water molecules. We used 256 molecules and 3 times steps. Finally **appbt** is from the NASA parallel benchmarks suite [4]. It computes an approximation to Navier-Stokes equations. It was translated to shared memory from the original message-based form by Doug Burger and Sanjay Mehta at the University of Wisconsin. Due to simulation constraints our input data sizes for all programs are smaller than what would be run on a real machine, a fact that may cause us to see unnaturally high degrees of sharing. Since we still observe reasonable scalability for all our applications we believe that this is not too much of a problem.

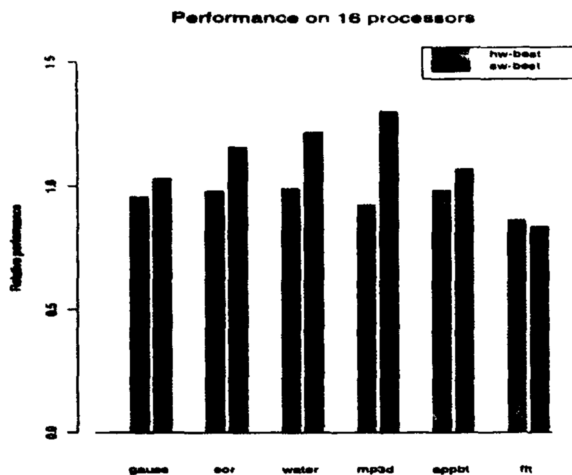


Figure 2: Comparative software and hardware system performance on 16 processors

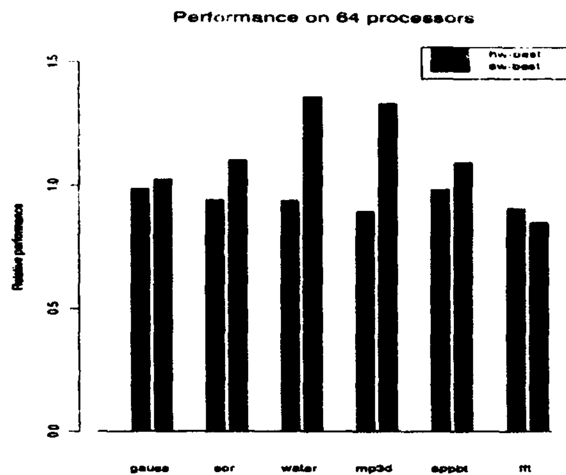


Figure 3: Comparative software and hardware system performance on 64 processors

4 Results

Scalable protocols for software cache coherence are little more than an academic exercise if the end result is not comparable in performance to a hardware coherent system. Figures 2 and 3 show the comparative performance of our best software protocol when compared to the best hardware protocol on 16 and 64 processors respectively. The unit line in the graphs represents the performance of a sequentially consistent hardware coherence protocol. In all cases the performance of the software protocol is within 45% of the performance of the hardware protocol. In most cases it is much closer. For `fft`, the software protocol is actually faster.

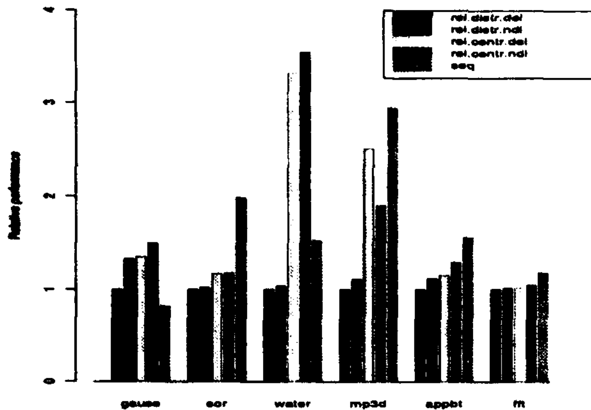
In several cases we have made minor changes to the applications (as described in section 4.3) to improve their performance under software coherence. We believe that the results for mp3d could be further improved, with more major restructuring of access to the space cell data structure.

In the following three subsections, we consider (1) variations on the software coherence protocol, (2) the choice among write-back caches, write-through caches, and write-through caches with a write-collect buffer, and (3) the types of program changes that improve the performance of software cache coherence, including the use of remote reference.

4.1 Comparison of different software coherence protocols

This section compares the different software protocols discussed in section 2. The architecture on which the comparison is made assumes a write-back cache which is flushed at the time of a release. Coherence messages (if needed) can be overlapped with the flush operations, once the writes have entered the network. The five protocols we compare are:

rel.distr.del: The delayed version of our distributed protocol, with `Safe` and `Unsafe` pages. Write notices are propagated at the time of a release and invalidations are done at the time of an



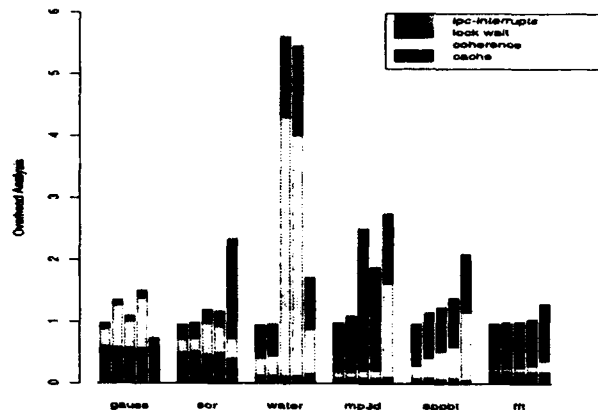


Figure 4: Comparative performance of different software protocols on 64 processors

Figure 5: Overhead analysis of different software protocols on 64 processors

acquire. At release time, the protocol scans the TLB dirty bits to determine which pages have been written. Pages can therefore be mapped read/write on the first miss, eliminating the need for a second trap if a read to an unmapped page is followed by a write. This protocol has slightly higher bookkeeping than `rel.distr.nodel` below, but reduces trap costs and possible coherence overhead by postponing sending write notices as long as possible. It provides the unit of comparison (normalized running time of 1) for the other protocols.

rel.distr.nodel: Same as `rel.distr.del`, except that write notices are propagated as soon as an inconsistency occurs. (Invalidations are done at the time of an acquire, as before.) While this protocol has slightly less bookkeeping (no need to remember pages for an upcoming release operation), it may cause higher coherence overhead and higher trap costs. The TLB dirty bits are not sufficient here, since we want to take action the moment an inconsistency occurs. We must use the write-protect bits to generate page faults.

rel.centr.del: Same as `rel.distr.del`, except that propagation of write notices is done by inserting weak pages in a global list which is traversed on acquires.

rel.centr.nodel: Same as `rel.distr.nodel`, except that propagation of write notices is done by inserting *weak* pages in a global list which is traversed on acquires.

seq: A sequentially consistent software protocol that allows only a single writer for every coherence block at any given point in time. Interprocessor interrupts are used to enforce coherence when an access fault occurs. Interrupts present several problems for our simulation environment (fortunately this is the only protocol that needs them) and the level of detail at which they are simulated is significantly lower than that of other system aspects. Results for this protocol may underestimate the cost of coherence management (especially in cases of high network traffic) but since it is the worst protocol in most cases, the inaccuracy has no effect on our conclusions.

Figure 4 presents the running time of the different software protocols on our set of partially modified applications. We have used the best version of the applications that does not require protocol modifications (i.e. no reader/writer locks or use of remote reference). The distributed protocols outperform the centralized implementations, often by a significant margin. The distributed protocols show the largest improvement (almost three-fold) on **water** and **mp3d**, the two applications in which software coherence lagged the most behind hardware coherence. This is predictable behavior: applications in which the impact of coherence is important are expected to show the greatest variance with different coherence algorithms.

The one application in which the sequential protocol outperforms the relaxed alternatives is Gaussian elimination. While the actual difference in performance may be smaller than shown in the graph, due in part to the reduced detail in the implementation of the sequential protocol, there is one source of overhead that the relaxed protocols have to pay that the sequential version does not. Since the releaser of a lock does not know who the subsequent acquirer of the lock will be, it has to flush changes to shared data at the time of a release in the relaxed protocols, so those changes will be visible. **Gauss** uses locks as flags to indicate that a particular pivot row is available to processors to eliminate their rows. In section 4.3 we note that use of the flags results in many unnecessary flushes, and we present a refinement to the relaxed consistency protocols that avoids them.

Sor and **water** have a very disciplined form of sharing, **sor** among neighbors and **water** within a well-defined subset of the processors partaking in the computation. The distributed protocol makes a processor pay a coherence penalty only for the pages it cares about, while the centralized one forces processors to examine all weak pages, which is all the shared pages in the case of **water**, resulting in very high overheads.

Appbt and **fft** have limited sharing. **Fft** exhibits limited pairwise sharing among different processors for every phase (the distance between paired elements decreases for each phase). We were unable to establish the access pattern of **appbt** from the source code; it uses linear arrays to represent higher dimensional data structures and the computation of offsets often uses several levels of indirection.

Mp3d has very undisciplined and wide-spread sharing. We have modified the program slightly (prior to the current studies) to ensure that colliding molecules belong with high probability to either the same processor or neighboring processors. Therefore the molecule data structures exhibit limited pairwise sharing. The main problem is the space cell data structures. Space cells form a three dimensional array. Unfortunately molecule movement is fastest in the outermost dimension resulting in long stride access to the space cell array. That coupled with the large coherence block results in having all the pages of the space cell data structure shared across all processors. Since the processors modify the data structure for every particle they process, the end behavior is a long weak list and serialization on the centralized protocols. The distributed protocols improve the coherence management of the molecule data structures but can do little to improve on the cell data structure, since sharing is wide-spread.

While runtime is the most important metric of application performance it does not capture the full impact of a coherence algorithm. Figure 5 shows the breakdown of overhead into its major components for the five software protocols on our six applications. These components are: IPC interrupt handling overhead (sequentially consistent protocol only), time spent waiting for application locks, coherence protocol overhead (including waiting for system locks and flushing and purging cache lines), and time spent waiting for cache misses. Coherence protocol overhead has an impact on the

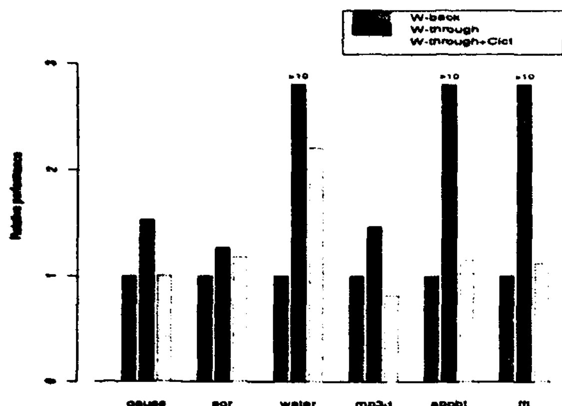


Figure 6: Comparative performance of different cache architectures on 64 processors

Application	Write Back	Write Through	Write Collect
Gauss	38%	74%	47%
Sor	8.5%	52.5%	40.5%
Water	5.1%	25%	8.3%
Mp3d	24.2%	40.7%	39%
Appbt	11.1%	28.4%	18.6%
Fft	29.3%	N/A [2]	38.8%

Figure 7: Delayed cache misses for different cache types

time spent waiting for application locks—the two are not easily separable. The relative heights of the bars do not agree in figures 4 and 5, because the former pertains to the critical path of the computation, while the latter provides totals over all processors for the duration of execution. The coherence part of the overhead is significantly reduced by the distributed delayed protocol for all applications. For mp3d the main benefit comes from the reduction of lock waiting time. The program is tightly synchronized; a reduction in coherence overhead implies less time holding synchronization variables and therefore a reduction in synchronization waiting time.

4.2 Choice of cache type

In this section we consider the choice of write policy for the cache. Specifically, we compare the performance obtained with a write-through cache, a write-back cache, and a write-through cache with a buffer for merging writes [10]. We assume that a single policy is used for all cached data: we cannot use a different policy based on whether data is private or shared.

Write-back caches impose the minimum load on the memory and network, since they write blocks back only on eviction, or when explicitly flushed. In a software coherent system, however, write-back caches have two undesirable qualities. The first of these is that they delay the execution of synchronization operations, since dirty lines must be flushed at the time of a release. Write-through caches have the potential to overlap memory accesses with useful computation.

The second problem is more serious, because it affects program correctness in addition to performance. Because a software coherent system allows multiple writers for the same page, it is possible for different portions of a cache line to be written by different processors. When those lines are

[2] Our write-through simulation for `fft` required too much memory so we had to modify it slightly. The number of delayed misses that we have is not directly comparable with that of the other two protocols, although it is larger than either of them

flushed back to memory we must make sure that changes are correctly merged so no data modifications are lost. The obvious way to do this is to have the hardware maintain per-word dirty bits, and then to write back only those words in the cache that have actually been modified. We assume there is no sub-word sharing: words modified by more than one processor imply that the program is not correctly synchronized.

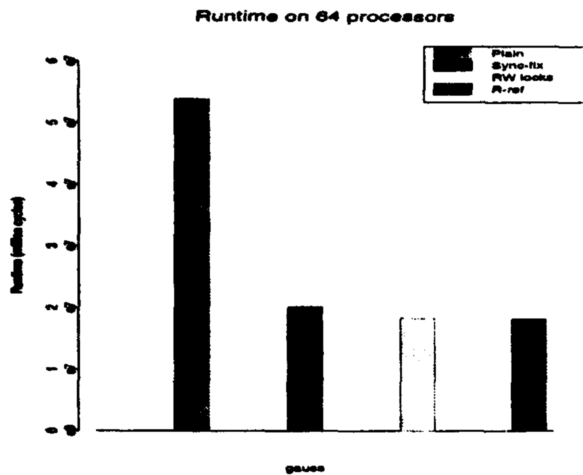
Write-through caches can potentially benefit relaxed consistency protocols by reducing the amount of time spent at release points. They also eliminate the need for per-word dirty bits. Unfortunately, they may cause a large amount of traffic, delaying the service of cache misses and in general degrading performance. In fact, if the memory subsystem is not able to keep up with all the traffic, write-through caches are unlikely to actually speed up releases, because at a release point we have to make sure that all writes have been globally performed before allowing the processor to continue. A write completes when it is acknowledged by the memory system. With a large amount of write traffic we may have simply replaced waiting for the write-back with waiting for missing acknowledgments.

Write-through caches with a write-collect buffer [10] employ a small (16 entries in our case) fully associative buffer between the cache and the interconnection network. The buffer merges writes to the same cache line, and allocates a new entry for a write to a non-resident cache line. When it runs out of entries the buffer randomly chooses a line for eviction and writes it back to memory. The write-collect buffer is an attempt to combine the desirable features of both the write-through and the write-back cache. It reduces memory and network traffic when compared to a plain write-through cache and has a shorter latency at release points when compared to a write-back cache. Unfortunately per-word dirty bits are required at the buffer to allow successful merging of cache lines into memory.

Figure 6 presents the relative performance of the different cache architectures when using the best relaxed protocol on our best version of the applications. For all programs with the exception of `mp3d` the write-back cache outperforms the others. The main reason is the reduced amount of memory traffic. Figure 7 presents the number of delayed cache misses under different cache policies. A miss is defined as delayed when it is forced to wait in a queue at the memory while contending accesses are serviced. The difference between the different cache types is more pronounced on programs that have little sharing or a lot of private data. `Water`, `appbt` and `fft` fall in this category. For `water`, which has a very large number of private writes, the write-through cache ends up degrading performance by a factor of more than 50.

For programs that have mostly shared data with active sharing, the write-through policies fare better. The best example is `mp3d`, in which the write-collect cache outperforms the write-back cache by about 20%. The reason for this is that frequent synchronization in `mp3d` requires frequent write-backs and thus generates approximately the same amount of traffic as it would with a write-through cache. Furthermore a flush operation on a page costs 128 cycles (1 cycle per line) regardless of the number of lines actually present in the cache. So if only a small portion of a page is touched, the write-back policy still pays a high penalty at releases.

Our results are in partial agreement with those reported in [10]. We both find that write-through caches suffer significant performance degradation due to increased network and memory traffic. However, while their results favor a write-collect buffer in most cases, we discover that write-back caches are preferable under our software scheme. We believe the difference stems from the fact that we overlap cache flush costs with coherence management (in their case cache flushes constitute the coherence management cost) and that we use a different set of applications.



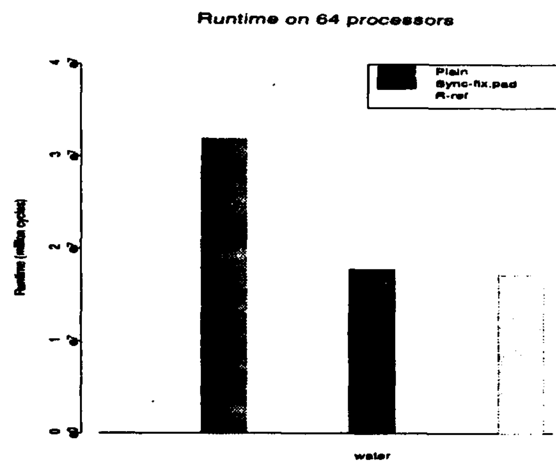


Figure 8: Runtime of Gauss with different levels of restructuring

Figure 9: Runtime of water with different levels of restructuring

4.3 Program modifications to support software cache coherence

As mentioned at the beginning of this section, the performance observed under software coherence is very sensitive to the locality properties of the application. In this sub-section we describe the modifications we had to make to our applications in order to get them to run efficiently on a software coherent system. We then present performance comparisons for the modified and unmodified applications.[3]

We have used four different techniques to improve the performance of our applications. Two are simple program modifications and require no additions to the coherence protocol. Two take advantage of program semantics to give hints to the coherence protocol on how to reduce coherence management costs. Our four techniques are:

- Separation of synchronization variables from other writable program data.
- Data structure alignment and padding at page or subpage boundaries.
- Identification of reader-writer locks and avoidance of coherence overhead at the release point.
- Identification of fine grained shared data structures and use of remote reference for their access to avoid coherence management.

All our changes produced dramatic improvements on the runtime of one or more applications, with some showing improvement of well over 100%.

Separation of busy-wait synchronization variables from the data they protect is the opposite of a common optimization for hardware-coherent systems, and is therefore counter-intuitive. Under

[3]Time constraints have prevented us from collecting all of the results in the previous two sub-sections for the modified versions of the programs. We would expect to put those results in the final version of the paper.

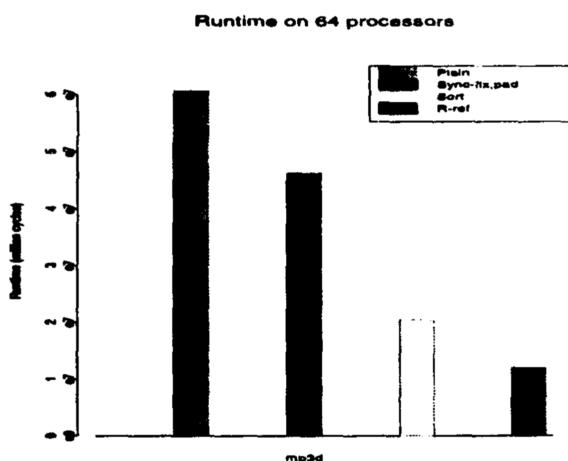


Figure 10: Runtime of mp3d with different levels of restructuring

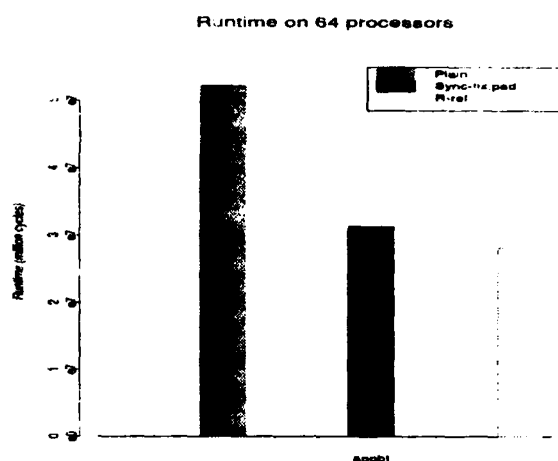


Figure 11: Runtime of appbt with different levels of restructuring

hardware coherence, co-location is desirable because lock acquisition results in prefetching the data (or part of it) that is protected by the lock. Under software coherence however such a tactic can prove disastrous. The problem stems from an adverse interaction between application and operating system synchronization variables. Processors trying to access the lock take a page fault trying to map the page containing the lock variable. When the processor that is in the critical section attempts to write a variable in the same page with the lock, it must perform a coherence transaction notifying other users of the page that it has been written. Unfortunately, the OS lock protecting the page table entry may not available, because the other users are serialized on it, trying to map the page. We have observed this effect in Gaussian elimination, where it interacts with access to the pivot row locks (see below), degrading performance almost three fold. It is also present in the implementation of barriers under the Argonne P4 macros (used by the SPLASH applications), since they employ a shared counter protected by a lock. We have changed the barrier implementation to avoid the problem in all our applications and have relocated the synchronization data structures in `Gauss` to eliminate the adverse interaction. `Gauss` enjoys the greatest improvement due to this change, though noticeable improvements occur in `water`, `appbt` and `mp3d` as well.[4]

Data structure alignment and padding is a well-known means of reducing false sharing [14]. Since coherence blocks in software coherent systems are large (4K bytes in our case), it is unreasonable to require padding of data structures to that size. However we can often pad data structures to subpage boundaries so that a collection of them will fit exactly in a page. This approach coupled with a careful distribution of work, ensuring that processor data is contiguous in memory, can greatly improve the locality properties of the application. `Water` and `appbt` already had the contiguity property, so padding was sufficient to achieve good performance. `Mp3d` however starts by assigning molecules to random coordinates in the three-dimensional space. As a result, interacting particles are seldom contiguous in memory, and generate large amounts of sharing. We fixed this problem by sorting the

[4]The hardware-coherent results in figures 2 and 3 are for the best-performing version of each program. This may be either the one with or the one without the segregated locks, though the differences are slight in all our applications.

particles according to their slow-moving X coordinate and assigned each processor a contiguous set of particles. Interacting particles are now likely to belong to the same page and processor, reducing the amount of sharing.

We were motivated to give special treatment to reader-writer locks after studying the Gaussian elimination program. **Gauss** uses locks to test for the readiness of pivot rows. In the process of eliminating a given row, a processor acquires (and immediately releases) the locks on the previous rows one by one. With regular exclusive locks, the processor is forced on each release to notify other processors of its most recent (single-element) change to its own row, even though no other processor will attempt to use that element until the entire row is finished. Our change is to observe that the critical section protected by the pivot row lock does not modify any data (it is in fact empty!), so no coherence operations are needed at the time of the release. We communicate this information to the coherence protocol by identifying the critical section as being protected by a reader's lock.[5]

In general, changing to the use of reader's locks means changing application semantics, since concurrent entry to a readers' critical section is allowed. Alternatively, one can think of the change as a program annotation that retains exclusive entry to the critical section, but permits the coherence protocol to skip the usual coherence operations at the time of the release. (In **Gauss** the difference does not matter, because the critical section is empty.) A "skip coherence operations on release" annotation could be applied even to critical sections that modify data, if the programmer or compiler is sure that the data will not be used by any other processor until after some *subsequent* release. This style of annotation is reminiscent of *entry consistency* [5], but with a critical difference. Entry consistency requires the programmer to identify the data protected by particular locks—in effect, to identify all situations in which the protocol must *not* skip coherence operations. Errors of *omission* affect the correctness of the program. In our case correctness is affected only by an error of *commission* (i.e. marking a critical section as protected by a reader's lock when this is not the case).

Even with the changes just described, there are program data structures that are shared at a very fine grain, and can therefore cause performance degradations. It can be beneficial to disallow caching for such data structures, and to access the memory module in which they reside directly. We term this kind of access remote reference, although the memory module may sometimes be local to the processor making the reference. We have identified the data structures in our programs that could benefit from remote reference and have annotated them appropriately by hand (our annotations range from one line of code in **water** to about ten lines in **mp3d**.) **Mp3d** sees the largest benefit: it improves by almost two fold when told to use remote reference on the space cell data structure. **Appbt** improves by about 12% when told to use remote reference on a certain array of condition variables. **Water** and **Gauss** improve only minimally; they have a bit of fine-grain shared data, but they don't use it very much.

The performance improvements for our four modified applications can be seen in figures 8 through 11. **Gauss** improves markedly when fixing the synchronization problem and also benefits from the identification of reader-writer locks. Remote reference helps only a little. **Water** gains most of its performance improvement by padding the molecule data structures to sub-page boundaries and relocating synchronization variables. **Mp3d** benefits from relocating synchronization variables and

[5] An alternative fix for **Gauss** would be to associate with each pivot row a simple flag variable on which the processors for later rows could spin. Reads of the flag would be acquire operations without corresponding releases. This fix was not available to us because our programming model provides no means of identifying acquire and release operations except through a pre-defined set of synchronization operations.

padding the molecule data structure to subpage boundaries. It benefits even more from improving the locality of particle interactions via sorting, and remote reference shaves off another 50%. Finally **appbt** sees dramatic improvements after relocating one of its data structures to achieve good page alignment and benefits nicely from the use of remote reference as well.

Our program changes were simple: none consumed more than a few hours of programmer effort once the problem was identified. We believe that such modest forms of tuning represent a reasonable demand on the programmer. We are also hopeful that smarter compilers will be able to make many of the changes automatically.

5 Related work

Our work is most closely related to that of Petersen and Li [21, 20]: we both use the notio[illegible]k pages, and purge caches on acquire operations. The major difference is our introduction of [illegible]e and **Unsafe** page states, with a resulting increase in scalability. We have also examined architectural alternatives that were not addressed by Petersen and Li. Our work resembles Munin [8] and lazy release consistency [15] in its use of delayed write notices, but we take advantage of the globally accessible physical address space for both coherence maintenance and data propagation.

Our use of remote reference to reduce the overhead of coherence management can also be found in work on NUMA memory management [6, 7, 12, 18, 17]. However relaxed consistency greatly reduces the opportunities for profitable remote data reference. In fact, early experiments we have conducted with on-line NUMA policies and relaxed consistency have failed badly in their attempt to determine when to use remote reference.

On the hardware side our work bears resemblance to the Stanford Dash project [19] in the use of a relaxed consistency model, and to the Georgia Tech Beehive project [22] in the use of relaxed consistency and per-word dirty bits for successful merging of inconsistent cache lines. Both these systems use their extra hardware to allow coherence messages to propagate in the background of computation (possibly at the expense of extra coherence traffic) in order to avoid a higher waiting penalty at synchronization operations.

Coherence for distributed memory with per-processor caches can also be maintained entirely by a compiler [11, 13]. Under this approach the compiler inserts the appropriate cache flush and invalidation instructions in the code, to enforce data consistency. The static nature of the approach, however, and the difficulty of determining access patterns for arbitrary programs, often dictates conservative decisions that result in higher miss rates and reduced performance.

6 Conclusions

We have shown that supporting a shared memory programming model while maintaining high performance does not necessarily require expensive hardware. Similar results can be achieved by maintaining coherence in software using the operating system and address translation hardware. We have introduced a new adaptive protocol for software cache coherence and have shown that it out-performs existing approaches (both relaxed and sequentially consistent). We have also studied the tradeoffs between different cache write policies, showing that in most cases a write-back cache is preferable

but that a write-collect buffer can help make a write-through cache feasible. Both write-back (with per-word dirty bits) and write-collect require special hardware, but neither approaches the complexity of full-scale hardware coherence. Finally we have shown how some simple program modifications can significantly improve performance on a software coherent system.

We are currently studying the sensitivity of software coherence schemes to architectural parameters (e.g. network latency and page and cache line sizes). We are also pursuing protocol optimizations that will improve performance for important classes of programs. For example, we are considering policies in which flushes of modified lines and purges of invalidated pages are allowed to take place "in the background"—during synchronization waits or idle time, or on a communication co-processor. We are developing on-line policies that use past page behavior to identify situations in which remote access is likely to out-perform remote cache fills. We are considering several issues in the use of remote reference, such as whether to adopt it globally for a given page, or to let each processor make its own decision (and deal with the coherence issues that then arise). Finally, we believe strongly that software coherence can benefit greatly from compiler support. We are actively pursuing the design of annotations that a compiler can use to reduce the overhead of OS-based coherence management.

Acknowledgements

We would like to thank Ricardo Bianchini and Jack Veenstra for the long nights of discussions, idea exchanges and suggestions that helped make this paper possible.

References

[1] A. Agarwal and others. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1992.

[2] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1989.

[3] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108-120, Santa Clara, CA, April 1991.

[4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Report RNR-91-002, NASA Ames Research Center, January 1991.

[5] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[6] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19-31, Litchfield Park, AZ, December 1989.

[7] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212-221, Santa Clara, CA, April 1991.

[8] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152-164, Pacific Grove, CA, October 1991.

[9] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224-234, Santa Clara, CA, April 1991.

[10] Y.-C. Chen and A. Veidenbaum. An Effective Write Policy for Software Coherence Schemes. In *Proceedings Supercomputing '92*, Minneapolis, MN, November 1992.

[11] H. Cheong and A. V. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *Computer*, 23(6):39-47, June 1990.

[12] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32-44, Litchfield Park, AZ, December 1989.

[13] E. Darnell, J. M. Mellor-Crummey, and K. Kennedy. Automatic Software Cache Coherence Through Vectorization. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[14] M. D. Hill and J. R. Larus. Cache Considerations for Multiprocessor Programmers. *Communications of the ACM*, 33(8):97-102, August 1990.

[15] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *ACM SIGARCH Computer Architecture News*, 20(2), May 1992.

[16] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. ParaNet: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter '94 Technical Conference*, San Francisco, CA, January 1994.

[17] R. P. LaRowe Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319-363, November 1991.

[18] R. P. LaRowe Jr., C. S. Ellis, and L. S. Kaplan. The Robustness of NUMA Memory Management. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 137-151, Pacific Grove, CA, October 1991.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash Multiprocessor. *Computer*, 25(3):63-79, March 1992.

[20] K. Petersen and K. Li. Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support. In *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 1993.

[21] K. Petersen. Operating System Support for Modern Memory Hierarchies. Ph. D. dissertation, CS-TR-431-93, Department of Computer Science, Princeton University, October 1993.

[22] G. Shah and U. Ramachandran. Towards Exploiting the Architectural Features of Beehive. GIT-CC-91/51, College of Computing, Georgia Institute of Technology, November 1991.

[23] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5-44, March 1992.

[24] J. E. Veenstra. Mint Tutorial and User Manual. TR 452, Computer Science Department, University of Rochester, May 1993.

[25] J. E. Veenstra and R. J. Fowler. Mint: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, pages 201-207, Durham, NC, January - February 1994.