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

July 1994

Abstract

Shared memory is an appealing abstraction for parallel programming. It must be implemented with caches in order to perform well, however, and caches require a coherence mechanism to ensure that processors reference current data. Hardware coherence mechanisms for large-scale machines are complex and costly, but existing software mechanisms have not been fast enough to provide a serious alternative.

We present a new software coherence protocol that narrows the performance gap between hardware and software coherence. This protocol runs on NCC-NUMA[1] machines, in which a global physical address space allows processors to fill cache lines from remote memory. We compare the performance of the protocol to that of existing software and hardware alternatives. We also evaluate the tradeoffs among various write policies (write-through, write-back, write-through with a write-collect buffer). Finally, we observe that certain simple program changes can greatly improve performance. For the programs in our test suite, the performance advantage of hardware cache coherence is small enough to suggest that software coherence may be more cost effective.

Keywords: cache coherence, scalability, cost-effectiveness, lazy release consistency, NCC-NUMA machines

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 an appealing programming model for such machines. To perform well, however, shared memory requires the use of caches, which in turn require a coherence mechanism to ensure that copies of data are sufficiently up-to-date. Coherence is easy to achieve on small, bus-based machines, where every processor can see the memory traffic of the others [4, 16]. Coherence is substantially harder to achieve on large-scale

*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.

multiprocessors [1, 21, 25]; it increases both the cost of the machine and the time and intellectual effort required to bring it to market. Given the speed of advances in microprocessor technology, long development times generally lead to machines with out-of-date processors. There is thus a strong motivation to find coherence mechanisms that will produce acceptable performance with little or no special hardware.[2]

There are at least three reasons to hope that a software coherence mechanism might 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. Third, programmers and compiler developers are becoming aware of the importance of locality of reference and are attempting to write programs that communicate as little as possible, thereby reducing the impact of coherence operations. In this paper we present a software coherence mechanism that exploits these opportunities to deliver performance approaching that of the best hardware alternatives—within 45% worst case in our experiments, and usually much closer.

As in most software coherence systems, we use address translation hardware to control access to shared pages. To minimize the impact of the false sharing that comes with such large coherence blocks [9, 17], we employ a relaxed consistency protocol that combines aspects of both *eager release consistency* [10] and *lazy release consistency* [19, 20]. We target our work, however, at NCC-NUMA machines, rather than message-based multicomputers or networks of workstations. Machines in the NCC-NUMA class include the Cray T3D, the BBN TC2000, and the Princeton Shrimp [**?**]. None of these has hardware cache coherence, but each provides a globally-accessible physical address space, with hardware support for cache fills and uncached references that access remote locations. In comparison to multicomputers, NCC-NUMA machines are only slightly harder to build, but they provide two important advantages for implementing software coherence: they permit very fast access to remote directory information, and they allow data to be moved in cache-line size chunks.

We also build on the work of Petersen and Li [26, 27], who developed an efficient software implementation of release consistency for small-scale multiprocessors. The key observation of their work was that NCC-NUMA machines allow the coherence block and the data transfer block to be of different sizes. Rather than copy an entire page in response to an access fault, a software coherence mechanism for an NCC-NUMA machine can create a mapping to remote memory, allowing the hardware to fetch individual caches lines as needed, on demand.

Our principal contribution is to extend the work of Petersen and Li to large machines. We distribute and reorganize the directory data structures, inspect those structures only with regard to pages for which the current processor has a mapping, postpone coherence operations for as long as possible, and introduce a new dimension to the protocol state space that allows us to reduce the cost of coherence maintenance on "well-behaved" pages.

We compare our mechanism to a variety of existing alternatives, including sequentially-consistent hardware, release-consistent hardware, sequentially-consistent software, and the software coherence scheme of Petersen and Li. 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.

[2] We are speaking here of *behavior-driven coherence*—mechanisms that move and replicate data at run time in response to observed patterns of program behavior—as opposed to compiler-based techniques [13, 15].

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 a 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, by disabling caching. 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.

The rest of the paper is organized as follows. Section 2 describes our software coherence protocol and provides intuition for our algorithmic and architectural choices. Section 3 describes our experimental methodology and workload. We present performance results in section 4 and compare our work to other approaches in section 5. We summarize our findings and conclude in section 6.

2 The Software Coherence Protocol

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 [26, 27]. Petersen's algorithm was designed for small-scale multiprocessors with a single physical address space and non-coherent caches, and has been shown to work well for several applications on such machines.

Like most behavior-driven software coherence schemes, Petersen's relies on address translation hardware, and therefore uses pages as its unit of coherence. Unlike most software schemes, however, it does not migrate or replicate whole pages. Instead, it maps pages where they lie in main memory, and relies on the hardware cache-fill mechanism to bring lines into the local cache on demand. To minimize the frequency of coherence operations, the algorithm adopts release consistency for its memory semantics, and performs coherence operations only at synchronization points.[3] Between synchronization points, processes may continue to use stale data in their caches. To keep track of inconsistent copies, the algorithm keeps a count, in uncached main memory, of the number of readers and writers for each page, together with an uncached *weak list* that identifies all pages for which there are multiple writers, or a writer and one or more readers.

Pages that may become inconsistent under Petersen's scheme are inserted in the weak list by the processor that detects the potential for inconsistency. For example, if a processor attempts to read a variable in a currently-unmapped page, the page fault handler creates a read-only mapping, increments the reader count, and adds the page to the weak list if it has any current writers. On an *acquire* operation, a processor scans the (uncached) weak list, and purges all lines of all weak pages from its cache. The processor also removes all mappings it may have for such a page. If all mappings for a page in the weak list have been removed, the page is removed from the weak list as well.

Unfortunately, while a centralized weak list works well on small machines, it poses serious obstacles to scalability: the size of the list, and consequently the amount of work that a processor needs to perform at a synchronization point, increases with the size of the machine. Moreover the frequency of references to each element of the list also increases with the size of the machine, implying the

[3] Under release consistency [24], memory references are classified as *acquires*, *releases*, or ordinary references. A release indicates that the processor is completing an operation on which other processor(s) may depend; all of the processor's previous writes must be made visible to any processor that performs a subsequent acquire. An acquire indicates that the processor is beginning an operation that may depend on someone else; all other processors' writes must be now be made locally visible.

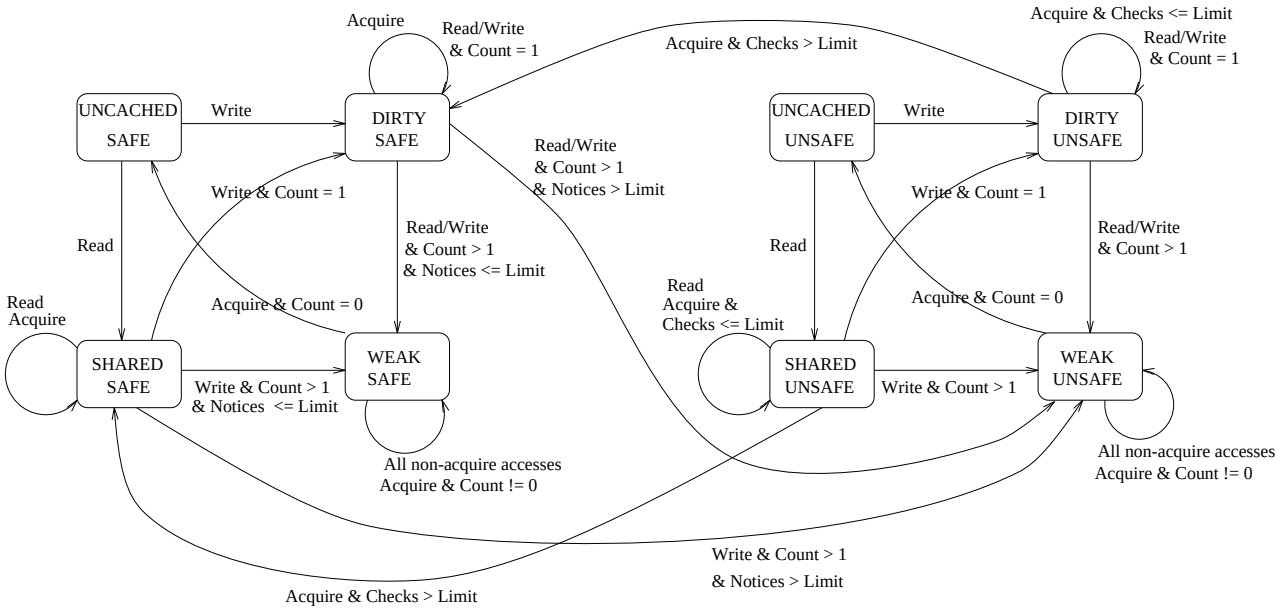

Figure 1: Scalable software cache coherence state diagram

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 [11], an algorithm with the above property should scale nicely to larger numbers of processors.

Our solution assumes a distributed, non-replicated directory data structure that maintains cacheability and sharing information, similar to the coherent map data structure of PLATINUM [14]. Pages can be in one of the following 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.

To facilitate transitions from weak back to the other states, the coherent map includes auxiliary counts of the number of readers and writers of each page.

Each processor holds the portion of the coherent map that describes the pages whose physical memory is local to that processor—the pages for which the processor is the *home node*. In addition, each processor holds a local weak list that indicates which of the pages to which it has mappings are weak. When a processor takes a page fault it locks the coherent map entry representing the page on which the fault was taken. It then changes the coherent map entry to reflect the new state of the page. If necessary (i.e. if the page has made the transition from shared or dirty to weak) the processor updates the weak lists of all processors that have mappings for that page. It then unlocks

the entry in the coherent map. The process of updating a processor's weak list is referred to as posting a *write notice*.

Distribution of the coherent map and weak list eliminates both the problem of centralization (i.e. memory contention) and the need for processors to do unnecessary work at acquire points (scanning weak list entries in which they have no interest). However it makes the transition to the weak state considerably more expensive, since a potentially large number of remote memory operations might have to be performed (serially) in order to notify all sharing processors. Ideally, we would like to maintain the low acquire overhead of per-processor weak lists while requiring only a constant amount of work per shared page on a transition to the weak state.

In order to approach this goal we take advantage of the fact that page behavior tends to be relatively constant over the execution of a program, or at least a large portion of it. Pages that are weak at one acquire point are likely to be weak at another. We therefore introduce an additional pair of states, called **safe** and **unsafe**. These new states, which are orthogonal to the others (for a total of 8 distinct states), reflect the past behavior of the page. A page that has made the transition to weak several times and is about to be marked weak again is also marked as unsafe. Future transitions to the weak state will no longer require the sending of write notices. Instead the processor that causes the transition to the weak state changes only the entry in the coherent map, and then continues. The acquire part of the protocol now requires that the acquiring processor check the coherent map entry for all its unsafe pages, and invalidate the ones that are also marked as weak. A processor knows which of its pages are unsafe because it maintains a local list of them (this list is never modified remotely). A page changes from unsafe back to safe if has been checked at several acquire operations and found not to be weak.

The correctness of the protocol depends on the following observation: unsafe pages are known as such by the processors that create new mappings to them; they will therefore be checked for the possibility of being weak on the next acquire operation. Safe pages have write notices posted on their behalf when they make the transition to the weak state. When a page first makes the transition to unsafe some processors (those that already have mappings to it) will receive write notices; others (those that create subsequent mappings) will know that it is unsafe from the state information saved in the coherent map entry.

Comparing our protocol to the use of a central weak list, we see that rather than iterate over *all* weak pages at each acquire point, a processor iterates over only those pages to which it currently has a mapping, and that on the basis of past behavior have a high probability of really being weak. The comparatively minor downside is that for pages that become weak without a past history of doing so, a processor must pay the cost of posting appropriate write notices.

The state diagram for a page in our protocol appears in figure 1. The state of a page is represented in the coherent map. It is a property of the system as a whole, not (as in most protocols) the viewpoint of a single processor. The transactions represent read, write, and acquire accesses on the part of any processor. `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 that a processor has checked the coherent map regarding an unsafe page and found it not to be weak. The access to the coherent map is then wasted work, since the processor was not required 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.

We apply one additional optimization. When a processor takes a page fault on a write to a shared, non-weak page we could choose to make the transition to weak (and post write notices if the

page was safe) immediately, or we could choose to wait until the processors's next release operation: the semantics of release consistency do not require us to make writes visible before then[4]. The advantage of delayed transitions is that any processor that executes an acquire operation before the writing processor's next release will not have to invalidate the page. This serves to reduce the overall number of invalidations. On the other hand, delayed transitions have the potential to lengthen the critical path of the computation by introducing contention, especially for programs with barriers, in which many processors may want to post notices for the same page at roughly the same time, and will therefore serialize on the lock of the coherent map entry. Delayed write notices were introduced in the Munin distributed shared memory system [10], which runs on networks of workstations and communicates solely via messages. Though the relative values of constants are quite different, experiments indicate (see section 4) that delayed transitions are generally beneficial in our environment as well.

One final question that has to be addressed is the mechanisms whereby written data makes its way back into main memory. Petersen in her work found a write-through cache to be the best option, but these can generate a potentially unacceptable amount of memory traffic in large-scale systems. Assuming a write-back cache either requires that no two processors write to the same cache line of a weak page—an unreasonable assumption—or a mechanism to keep track of which individual words are dirty. We ran our experiments (see section 4.1) under three different assumptions: write-through caches, write-back caches with per-word hardware dirty bits in the cache, and write-through caches with a write-collect buffer [12] that hangs onto recently-written lines (16 in our experiments) and coalesces any writes that are directed to the same line. Depending on the write policy, the coherence protocol at a release operation must force a write-back of all dirty lines, purge the write-collect buffer, or wait for acknowledgments of write-throughs.

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 [30, 31], that 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 and control flow within a processor can change as a result of the timing of memory references. This is more accurate than trace-driven simulation, in which control flow 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 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

[4]Under the same principle a write page-fault on an unmapped page will take the page to the shared state. The writes will be made visible only on the subsequent release operation

System Constant Name	Default Value
TLB size	128 entries
TLB fill time	24 cycles
Interrupt cost	140 cycles
Coherent map 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 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 both in our hardware and software coherence simulations, which are in agreement with those published in [3] and in several hardware manuals.

Some of the transactions required by our coherence protocols require a collection of the operations shown in table 1 and therefore incur the aggregate cost of their constituents. For example a read page-fault on an unmapped page consists of the following: a) a TLB fault and TLB fill, b) a processor interrupt caused by the absence of read rights, c) a coherent map entry lock acquisition, and d) a coherent map entry modification followed by the lock release. Lock acquisition itself requires traversing the network and accessing the memory module where the lock is located. The total cost for the example transaction is well over 300 cycles.

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 [29]. `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 [5]. 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 the applications we believe that the data set sizes do not compromise our results.

4 Results

Our principal goal is to determine whether one can approach the performance of hardware cache coherence without the special hardware. To that end, we begin in section 4 by evaluating the tradeoffs between different software protocols. Then, in sections 4.1 and 4.2, we consider the impact of different write policies and of simple program changes that improve the performance of software cache coherence. These changes include segregation of synchronization variables, data alignment and padding, use of reader-writer locks to avoid coherence overhead, and use of uncached remote references for fine-grain data sharing. Finally, in section 4.3, we compare the best of the software results to the corresponding results on sequentially-consistent and release-consistent hardware.

subsectionSoftware coherence protocol alternatives This section compares the software protocol alternatives 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 posted at the time of a release and invalidations are done at the time of an acquire. At release time, the protocol scans the TLB/page table 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 overhead than `rel.distr.nodel` below, but reduces trap costs and possible coherence overhead by delaying transitions to the dirty or weak state (and posting of associated write notices) for as long as possible. It provides the unit of comparison (normalized running time of 1) in our graphs.

rel.distr.nodel: Same as `rel.distr.del`, except that write notices are posted as soon as an inconsistency occurs. (Invalidations are done at the time of an acquire, as before.) While this protocol has slightly less bookkeeping overhead (no need to remember pages for an upcoming release operation), it may cause higher coherence overhead and higher trap costs. The TLB/page table 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 write notices are propagated by inserting weak pages in a global list which is traversed on acquires. List entries are distributed among the nodes of the machine although the list itself is conceptually centralized.

rel.centr.nodel: Same as `rel.distr.nodel`, except that write notices are propagated by inserting *weak* pages in a global list which is traversed on acquires. This is the protocol proposed by Petersen and Li [26, 27]. The previous protocol (`rel.centr.del`) is also similar to that of Petersen and Li with the addition of the delayed write notices.

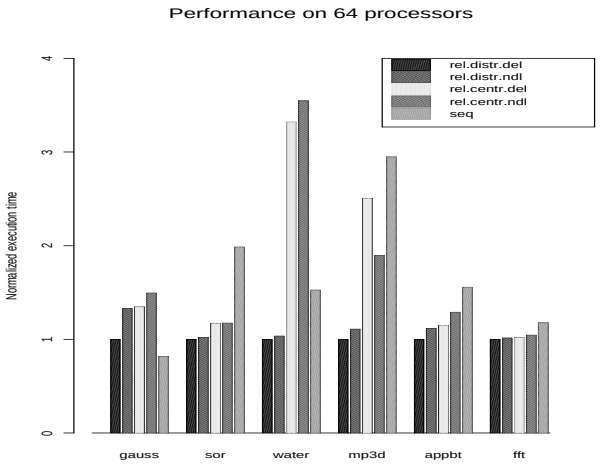

Figure 2: Comparative performance of different software protocols on 64 processors

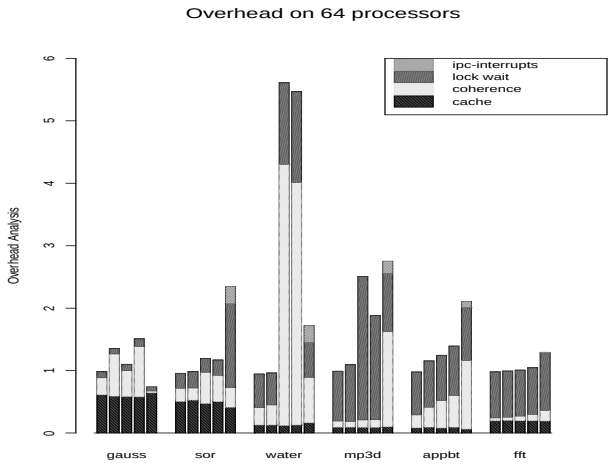

Figure 3: Overhead analysis of different software protocols on 64 processors

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. Interprocessor 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 2 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 identification of reader/writer locks or use of remote reference; see section 4.2). The distributed protocols outperform the centralized implementations, often by a significant margin. The distributed protocols also show the largest improvement (almost three-fold) on `water` and `mp3d`, the two applications in which software coherence lags the most behind hardware coherence (see section 4.3). This is predictable behavior: applications in which the impact of coherence is important are expected to show the greatest variance with different coherence algorithms. However it is important to notice the difference in the scale of figures 2 and 11. While the distributed protocols improve performance over the centralized ones by a factor of three for water and mp3d they are only 30 to 40% worse than their hardware competitors. In programs where coherence is less important, the decentralized protocols still provide reasonable performance improvements over the centralized ones, ranging from 2% to 35%.

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 4.2 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 very regular sharing patterns, `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. It is interesting to notice that in `water` the centralized relaxed consistency protocols are badly beaten by the sequentially consistent software protocol. This agrees to some extent with the results reported by Petersen and Li [26], but the advantage of the sequentially consistent protocol was less pronounced in their work. We believe there are two reasons for our difference in results. First we have restructured the code to greatly reduce false sharing,[5] thus removing one of the advantages that relaxed consistency has over sequential consistency. Second, we have simulated a larger number of processors, aggravating the contention caused by the centralized weak list used in the centralized relaxed consistency protocols.

`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` [29] has very wide-spread sharing. We 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 for application performance it does not capture the full impact of a coherence algorithm. Figure 3 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 time spent waiting for application locks—the two are not easily separable. The relative heights of the bars do not agree in figures 2 and 3, because the former pertains to the critical path of the computation, while the latter provides totals over all processors for the duration of execution. Aggregate costs for the overhead components can be higher but critical path length can be shorter if some of the overhead work is done in parallel. The coherence part of the overhead is significantly reduced by the distributed delayed protocol for all applications. For `mp3d` the main

[5] The sequentially consistent software protocol still outperforms the centralized relaxed consistent software protocols on the unmodified application but to a lesser extent.

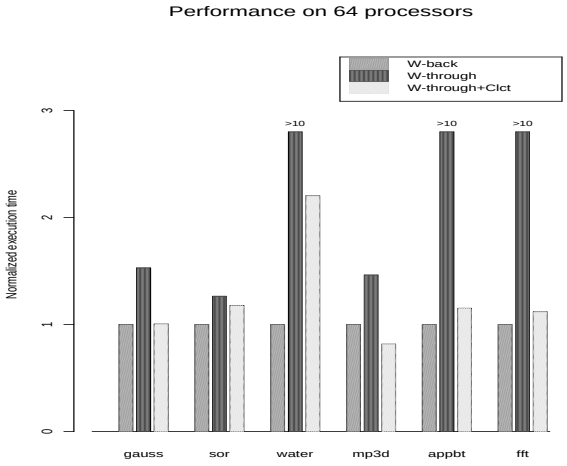

Figure 4: Comparative performance of different cache architectures on 64 processors

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 [6]	38.8%

Figure 5: Delayed cache misses for different cache types

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.

We have also run simulations in order to determine the performance benefits caused by the introduction of the safe and unsafe states. What we have discovered is that for our modified applications the performance impact of these two states is small; they help performance in some cases by up to 5%, and hurt it in other cases (due to unnecessary checks on the unsafe state) by up to 3%. The reason for this behavior is the limited degree of sharing for pages exhibited by our modified applications. We have run simulations on unmodified applications and have found that the existence of these two states can help improve performance by as much as 35%. Unfortunately the performance of software coherence, even with the introduction of our optimization is not competitive to hardware for the unmodified applications. We view our optimization as a safeguard that can help yield reasonable performance for bad sharing patterns, but for well behaving programs that scale nicely under software coherence, its impact is significantly reduced.

4.1 Write policies

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 [12]. We assume that a single policy is used for all cached data both private and shared. We have modified our simulator to allow us to vary policies independently for private and shared data, and expect to have results shortly that will simulate the above options for shared data only, while using a write-back policy for private data.

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 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 [12] 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. Per-word dirty bits are required at the buffer to allow successful merging of cache lines into memory.

Figure 4 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 5 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 most 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 whose data is mostly actively shared, 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, so the program 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.

[6] 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

Our results are in partial agreement with those reported by Chen and Veidenbaum [12]. 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 other coherence management (in their case cache flushes constitute the coherence management cost) and we use a different set of applications.

4.2 Program modifications to support software cache coherence

The performance observed under software coherence is very sensitive to the locality properties of the application. In this 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.

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 also used on hardware coherent systems to avoid invalidating the data protected by locks due to unsuccessful `test_and_set` operations on the locks themselves. Under software coherence however, this optimization becomes significantly more important to performance. The problem caused by the colocation is aggravated by an adverse interaction between the application locks and the locks protecting coherent map entries at the OS level. A processor that attempts to access an application lock for the first time will take a page-fault and will attempt to map the page containing the lock. This requires the acquisition of the OS lock protecting the coherent map entry for that page. The processor that attempts to release the application lock must also acquire the lock for the coherent map entry representing the page that contains the lock and the data it protects, in order to update the page state to reflect the fact that the page has been modified. In cases of contention the lock protecting the coherent map entry is unavailable: it is owned by the processor(s) attempting to map the page for access.

We have observed this lock-interaction effect in Gaussian elimination, in the access to the lock protecting the index to the next available row. 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 separated synchronization variables and data in `Gauss` to eliminate the

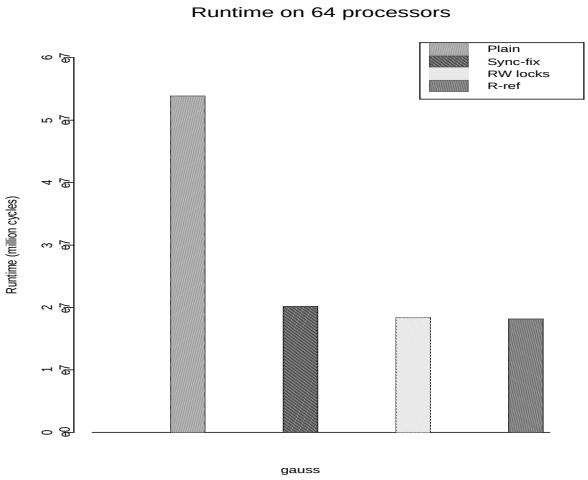

Figure 6: Runtime of `Gauss` with different levels of restructuring

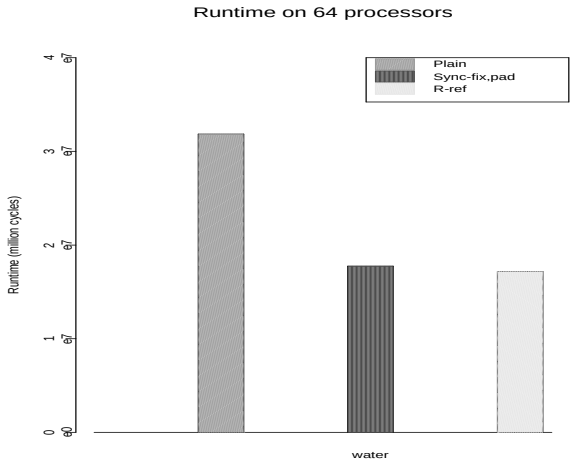

Figure 7: Runtime of `water` with different levels of restructuring

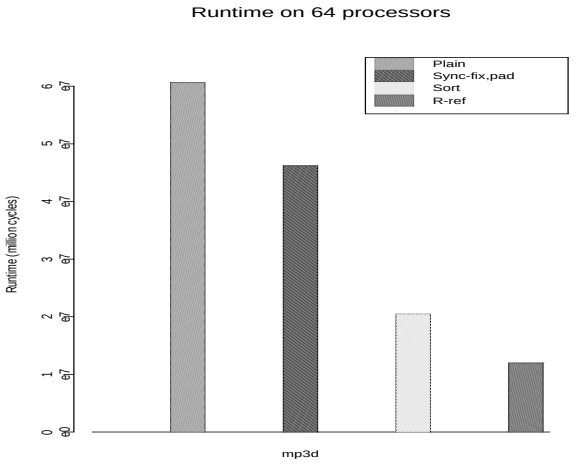

Figure 8: Runtime of `mp3d` with different levels of restructuring

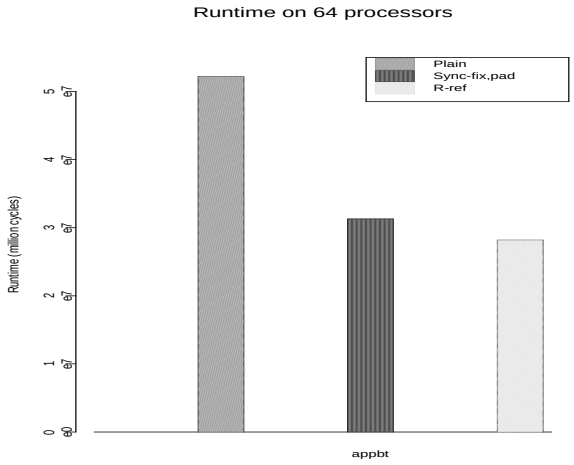

Figure 9: Runtime of `appbt` with different levels of restructuring

adverse interaction. `Gauss` enjoys the greatest improvement due to this change, though noticeable improvements occur in `water`, `appbt` and `mp3d` as well.

Data structure alignment and padding is a well-known means of reducing false sharing [18]. 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 good contiguity, so padding was sufficient to achieve good performance. `Mp3d` on the other hand 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 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.[7]

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* [6], 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 (both spatial and temporal), and that 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

[7] 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.

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 6 through 9. `Gauss` improves markedly when fixing the lock interference 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 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: identifying and fixing the problems was a mechanical process that consumed at most a few hours. The one exception was `mp3d` which, apart from the mechanical changes, required an understanding of program semantics for the sorting of particles. Even in that case identifying the problem was an effort of less than a day; fixing it was even simpler: a call to a sorting routine. 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. The results for `mp3d` could most likely be further improved, with more major restructuring of access to the space cell data structure, but this would require effort out of keeping with the current study.

4.3 Hardware v. software coherence

Figures 10 and 11 compare the performance of our best software protocol to that of a relaxed-consistency DASH-like hardware protocol [24] on 16 and 64 processors respectively. The unit line in the graphs represents the running time of each application under a sequentially consistent hardware coherence protocol. In all cases the performance of the software protocol is within 45% of the better of the hardware protocols. In most cases it is much closer. For `fft`, the software protocol is fastest.

For all programs the best software protocol is the one described in section 2, with a distributed coherence map and weak list, safe/unsafe states, delayed transitions to the weak state, and (except for `mp3d`) write-back caches augmented with per-word dirty bits.[8] The applications include all the program modifications described in section 4.2, though remote reference is used only in the context of software coherence; it does not make sense in the hardware-coherent case. Experiments (not shown) confirm that the program changes improve performance under both hardware and software coherence, though they help more in the software case. They also help the sequentially-consistent hardware more than the release consistent hardware; we believe this accounts for the relatively modest observed advantage of the latter over the former.

[8] The mp3d result uses a write-through cache with the write-collect buffer since this is the configuration that performs best for software coherence on this program.

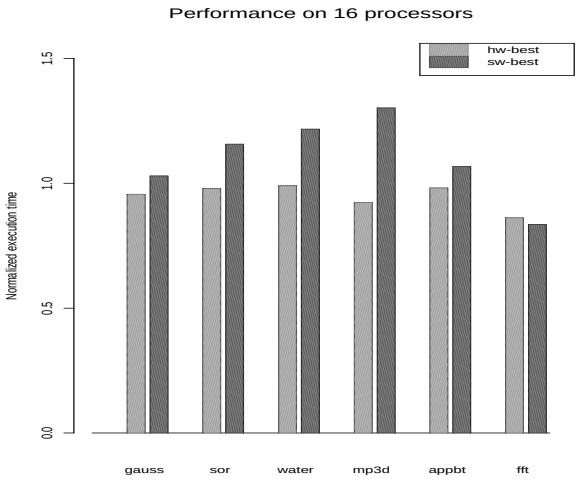

Figure 10: Comparative software and hardware system performance on 16 processors

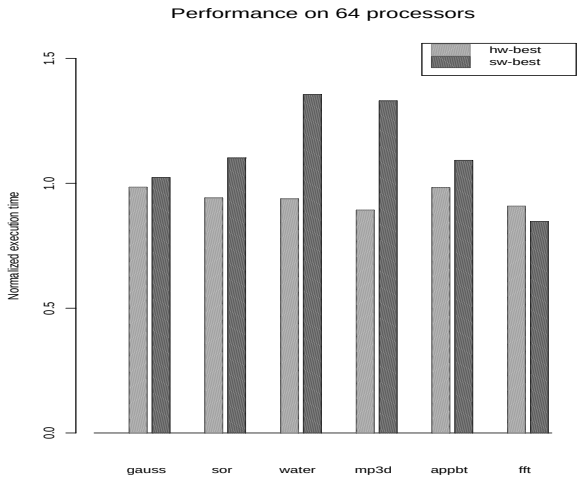

Figure 11: Comparative software and hardware system performance on 64 processors

5 Related Work

Our work is most closely related to that of Petersen and Li [26, 27]: we both use the notion of weak pages, and purge caches on acquire operations. The difference is scalability: we distribute the coherent map and weak list, distinguish between safe and unsafe pages, check the weak list only for unsafe pages mapped by the current processor, and multicast write notices for safe pages that turn out to be weak. We have also examined architectural alternatives and program-structuring issues that were not addressed by Petersen and Li. Our work resembles Munin [10] and lazy release consistency [19] in its use of delayed write notices, but we take advantage of the globally accessible physical address space for cache fills and for access to the coherent map and the local weak lists.

Our use of remote reference to reduce the overhead of coherence management can also be found in work on NUMA memory management [7, 8, 14, 22, 23]. 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 [25] in the use of a relaxed consistency model, and to the Georgia Tech Beehive project [28] 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 [13, 15]. 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 scalable 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 acceptable. 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 provide performance-enhancing hints for OS-based coherence.

Acknowledgements

Our thanks to 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] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

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

[6] 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.

[7] 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.

[8] 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.

[9] W. J. Bolosky and M. L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the Fourth USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–71, September 1993. Also available as MSR-TR-93-1, Microsoft Research Laboratory, September 1993.

[10] 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.

[11] 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.

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

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

[14] 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.

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

[16] S. J. Eggers and R. H. Katz. Evaluation of the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the Sixteenth International Symposium on Computer Architecture*, pages 2–15, May 1989.

[17] S. J. Eggers and R. H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, MA, April 1989.

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

[19] 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.

[20] 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.

[21] Kendall Square Research. KSR1 Principles of Operation. Waltham MA, 1992.

[22] 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.

[23] 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.

[24] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, May 1990.

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

[26] 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.

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

[28] 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.

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

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

[31] 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.