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

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 for message-passing machines have not provided a performance-competitive solution. We claim that an intermediate hardware option—memory-mapped network interfaces that support a global physical address space—can provide most of the performance benefits of hardware cache coherence. We present a software coherence protocol that runs on this class of machines and greatly narrows the performance gap between hardware and software coherence. We compare the performance of the protocol to that of existing software and hardware alternatives and evaluate the tradeoffs among various cache-write policies. We also observe that simple program changes can greatly improve performance. For the programs in our test suite and with the changes in place, software coherence is often faster and never more than 55% slower than hardware coherence.

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 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, 11], but is substantially harder to achieve on large-scale multiprocessors [1, 15, 19]. 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 Given the speed of advances in microprocessor technology, long development times generally lead to machines with out-of-date processors.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.[1]

*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]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 [9].

Unfortunately, the current state of the art in software coherence for message-passing machines provides performance nowhere close to that of hardware cache coherence. Our contribution is to demonstrate that most of the benefits of hardware cache coherence can be obtained on large machines simply by providing a global physical address space, with per-processor caches but without hardware cache coherence. Machines in this non-cache-coherent, non-uniform memory access (NCC-NUMA) class include the Cray Research T3D and the Princeton Shrimp [6]. In comparison to hardware-coherent machines, NCC-NUMAs can more easily be built from commodity parts, and can follow improvements in microprocessors and other hardware technologies closely.

We present a software coherence protocol for NCC-NUMA machines that scales well to large numbers of processors. To achieve the best possible performance, we exploit the global address space in three specific ways. First, we maintain directory information for the coherence protocol in uncached shared locations, and access it with ordinary loads and stores, avoiding the need to interrupt remote processors in almost all circumstances. Second, while using virtual memory to maintain coherence at the granularity of pages, we never copy pages. Instead, we map them remotely and allow the hardware to fetch cache lines on demand. Third, while allowing multiple writers for concurrency, we avoid the need to keep old copies and compute diffs by using ordinary hardware write-through or write-back to the unique main-memory copy of each page.

Full-scale hardware cache coherence may suffer less from false sharing, due to smaller coherence blocks, and can execute protocol operations in parallel with "real" computation. Current trends, however, are reducing the importance of each of these advantages: relaxed consistency, better compilers, and improvements in programming methodology can also reduce false sharing, and trends in both programming methodology and hardware technology are reducing the fraction of total computation time devoted to protocol execution. At the same time, software coherence enjoys the advantage of being able to employ techniques such as multiple writers, delayed write notices, and delayed invalidations, which may be too complicated to implement reliably in hardware at acceptable cost. Software coherence also offers the possibility of adapting protocols to individual programs or data regions with a level of flexibility that is difficult to duplicate in hardware. We exploit this advantage in part in our work by employing a relatively complicated eight-state protocol, by using uncached remote references for application-level data structures that are accessed at a very fine grain, and by introducing user level annotations that can impact the behavior of the coherence protocol.

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 compare our protocol to a variety of existing alternatives, including sequentially-consistent hardware, release-consistent hardware, straightforward sequentially-consistent software, and a coherence scheme for small-scale NCC-NUMAs due to Petersen and Li [21]. 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-merge 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. We summarize our findings and conclude in section 6.

2 The Software Coherence Protocol

In this section we present a scalable protocol for software cache coherence. As in most software coherence systems, we use virtual memory protection bits to enforce consistency at the granularity of pages. As in Munin [26], Treadmarks [14], and the work of Petersen and Li [20, 21], we allow more than one processor to write a page concurrently, and we use a variant of release consistency to limit coherence operations to synchronization points. (Between these points, processors can continue to use stale data in their caches.) As in the work of Petersen and Li, we exploit the global physical address space to move data at the granularity of cache lines: instead of copying pages we map them remotely, and allow the hardware to fetch cache lines on demand.

The protocol employs a distributed, non-replicated directory data structure that maintains cacheability and sharing information, similar to the coherent map data structure of PLATINUM [10]. A page can be in one of the following four states:

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

Shared – One or more processors have read-only mappings to the 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 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*.

Use of a distributed coherent map and per-processor weak lists enhances scalability by minimizing memory contention and by avoiding the need for processors at acquire points to scan weak list entries in which they have no interest (something that would happen with a centralized weak list [20]. However it may make the transition to the weak state very expensive, since a potentially large number of remote memory operations may 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.

In practice we find that the distinction between safe and unsafe pages makes a modest, though not dramatic, contribution to performance in well-structured programs (up to 5% improvement in our application suite). It is more effective for more poorly-structured programs (up to 35% for earlier versions of our programs), for which it provides a "safety net", allowing their performance to be merely poor, instead of really bad.

The state diagram for a page in our protocol appears in figure 1. The state of a page as represented in the coherent map 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.

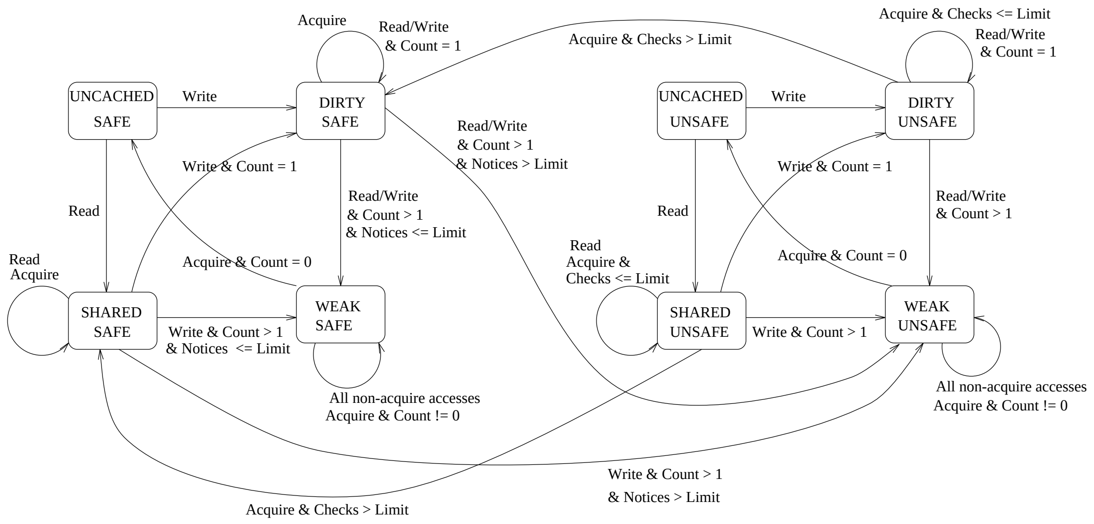
UNCACHED SAFE
DIRTY SAFE
SHARED SAFE
WEAK SAFE
UNCACHED UNSAFE
DIRTY UNSAFE
SHARED UNSAFE
WEAK UNSAFE
Write
Read
Acquire
Read/Write & Count = 1
Acquire & Checks > Limit
Acquire & Checks <= Limit
Read/Write & Count > 1 & Notices > Limit
Read/Write & Count > 1 & Notices <= Limit
Write & Count = 1
Acquire & Count = 0
Read Acquire
Write & Count > 1 & Notices <= Limit
All non-acquire accesses Acquire & Count != 0
Read Acquire & Checks <= Limit
Read/Write & Count > 1
Write & Count > 1
Write & Count > 1 & Notices > Limit

Figure 1: Scalable software cache coherence state diagram

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[2]. 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 [26], 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 mechanism whereby written data makes its way back into main memory. Petersen and Li found a write-through cache to be the best option, but this could lead to 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 4.2) 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-merge buffer [8] 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-merge buffer, or wait for acknowledgments of write-throughs.

[2]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

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 [25], that simulates the execution of the processors, and a back end that simulates the memory system. 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 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 page fault on a read to an unmapped page consists of the following: a) a TLB fault service, 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.

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

Lock acquisition itself requires traversing the network and accessing the memory module where the lock is located. Assuming that accessing the coherent entry lock requires traversing 10 intermediate nodes and there is no contention in the network and the lock is found to be free the cost for lock acquisition is $(2+1)*10+12+1+(2+1)*10 = 73$ cycles. The total cost for the above transaction would then be $24 + 140 + 73 + 160 = 398$ cycles.

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 [24]. `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 the applications (with the exception of mp3d) 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 4.1 by evaluating the tradeoffs between different software protocols. Then, in sections 4.2 and 4.3, 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 4.4, we compare the best of the software results to the corresponding results on sequentially-consistent and release-consistent hardware.

4.1 Software coherence protocol alternatives

This section compares our software protocol (presented in section 2) to the protocol devised by Petersen and Li [20] (modified to distribute the centralized weak list among the memories of the machine), and to a sequentially consistent page-based cache coherence protocol. For each of the first two protocols we present two variants: one that delays write-driven state transitions until the subsequent release operation, and one that performs them immediately. The comparisons assume a write-back cache. Coherence messages (if needed) can be overlapped with the flush operations, once the writes have entered the network. The protocols are named as follows:

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 protocol is a straightforward extension of the one proposed by Petersen and Li [20] to large scale distributed memory machines. The previous protocol (`rel.centr.del`) is also similar to that of Petersen and Li with the addition of the delayed write notices.

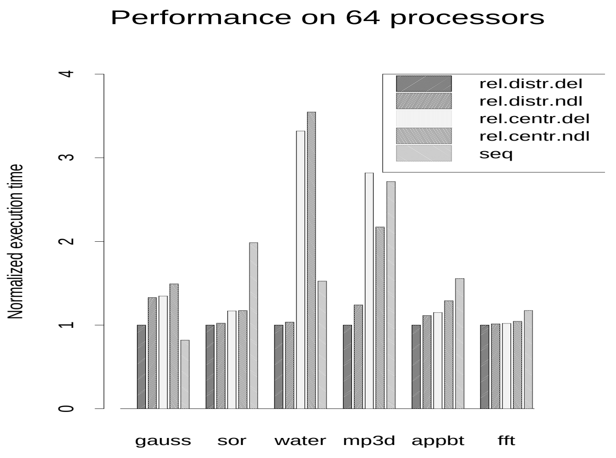

Figure 2: Comparative performance of different software protocols on 64 processors

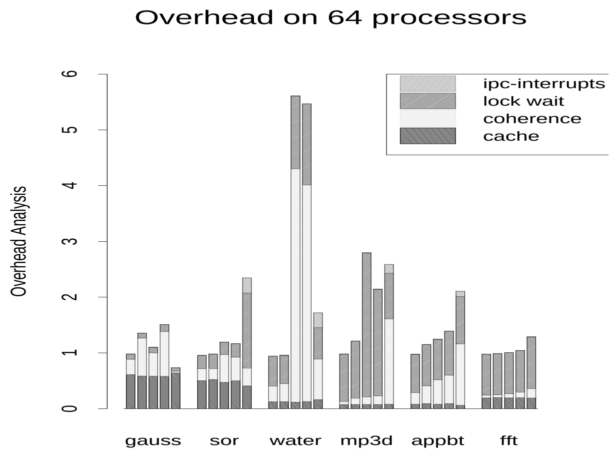

Figure 3: Overhead analysis of different software protocols on 64 processors

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 4.3). The distributed protocols outperform the centralized implementations, often by a significant margin. The largest improvement (almost three-fold) is realized on `water` and `mp3d`. These applications exhibit higher degrees of sharing and thus accentuate the importance of the coherence algorithm. In the remaining programs where coherence is less important, our protocols still provide reasonable performance improvements over the remaining 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 has become available to other processors. In section 4.3 we note that the use of locks as 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. 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 [20], 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, 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 shared 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` [24] 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 structure exhibits limited pairwise sharing. The main problem is the space cell data structure. 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 large coherence blocks results in all the pages of the space cell data structure being shared across all processors. Since the processors modify the data structure for every particle they process, the end result is a long weak list and serialization in the centralized protocols. The distributed protocols improve the coherence management of

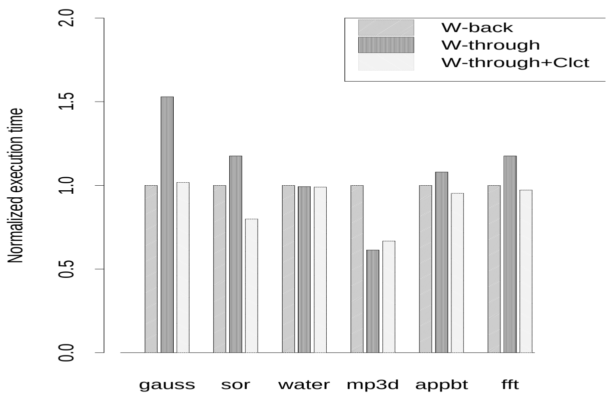

Figure 4: Comparative performance of different cache architectures on 64 processors

the molecule data structure but can do little to improve on the space cell data structure, since sharing is widespread.

While run time 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: interprocessor 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 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 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 [8]. The policy is applied on only shared data. Private data uses a write-back policy by default.

Write-back caches impose the minimum load on the memory and network, since they write blocks back only on evictions, 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-merge buffer [8] 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-merge 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 almost all programs the write-through cache with the write-merge buffer outperforms the others. The exceptions are `mp3d`, in which a simple write-through cache is better, and `Gauss`, in which a write-back cache is better. In both cases the performance of the write-through cache with the write-merge buffer is within 5% of the better alternative.

We also ran experiments using a single policy for both private and shared data. These experiments capture the behavior of an architecture in which write policies cannot be varied among pages. If a single policy has to be used for both shared and private data, a write-back cache provides the best performance. As a matter of fact the write-through policies degrade performance significantly, with plain write-through being as much as 50 times worse in `water`.

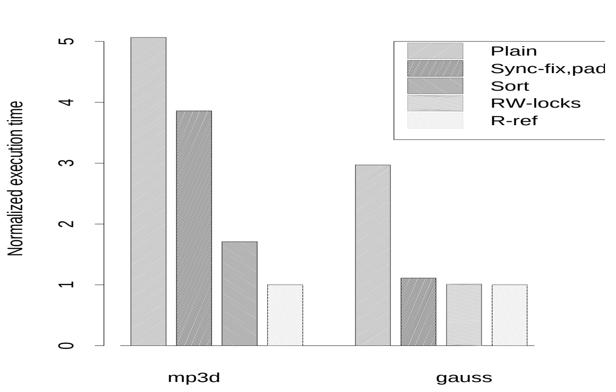

Figure 5: Normalized runtime of `Mp3d` and `Gauss` with different levels of restructuring

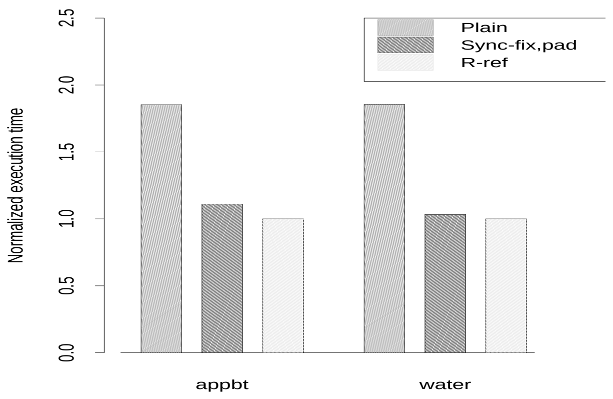

Figure 6: Normalized runtime of `appbt` and `water` with different levels of restructuring

4.3 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. They require no additions to the coherence protocol, and can be used in the context of hardware coherent systems as well. 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 50%.

Separation of busy-wait synchronization variables from the data they protect is also important on hardware coherent systems, where it avoids invalidating the data protected by locks in response 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 colocation is aggravated by an adverse interaction between 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 adverse interaction. `Gauss` enjoys the greatest improvement due to these changes, 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 [12]. 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.

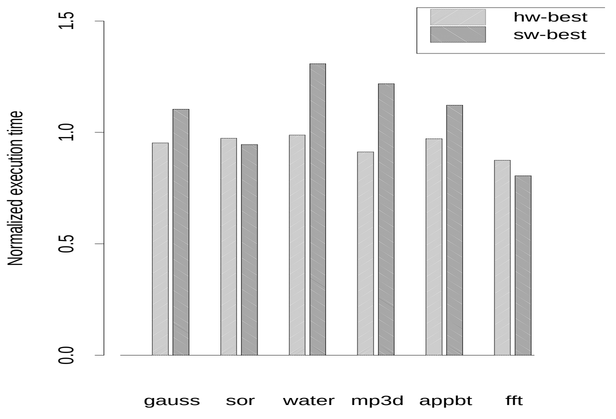

Figure 7: Comparative software and hardware system performance on 16 processors

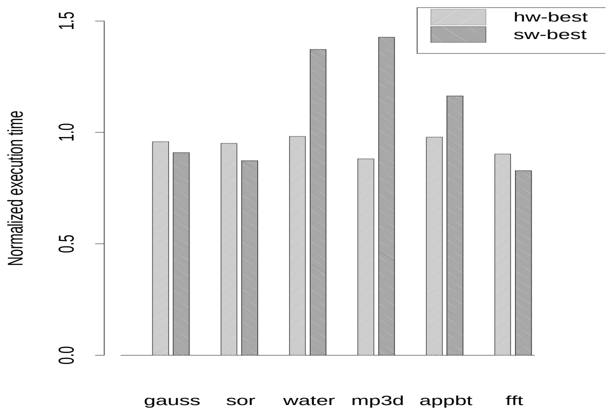

Figure 8: Comparative software and hardware system performance on 64 processors

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.[3]

Even with the changes just described, there are program data structures that are shared at a very fine grain, and that can therefore cause performance degradation. It can be beneficial to disallow caching for such data structures, and to access the memory modules 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 5 and6. `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 took 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.

4.4 Hardware v. software coherence

Figures 7 and 8 compare the performance of our best software protocol to that of a relaxed-consistency DASH-like hardware protocol [18] 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 55%[4] of the relaxed consistency hardware protocol. In most cases it is much closer. For the three kernels, the software protocol is faster than the hardware.

[3]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.

[4]The only example of a disparity as large as 55% occurs in mp3d on 64 processors. However mp3d does not really scale to a machine of that size. The 16-processor case provides a more meaningful comparison: here the performance difference is closer to 30%.

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 write-through caches with a write-merge buffer. The applications include all the program modifications described in section 4.3, 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. We have also run experiments (not shown here) varying several of the architectural constants in out simulations. In all cases software cache coherence maintained performance comparable to that of the hardware alternatives.

5 Related Work

Our work is most closely related to that of Petersen and Li [20]: we both use the notion of weak pages, and purge caches on acquire operations. The difference is scalability: we distribute the coherent map and use per processor weak lists, distinguish between safe and unsafe pages, check the coherent map 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 [26] and lazy release consistency [13] in its use of delayed write notices, but we take advantage of the globally accessible physical address space for cache fills and write-through, 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, 10, 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 [23] 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 [9]. 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-through cache with a write-merge buffer is preferable. Finally we have shown certain 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 should 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. In fact, we believe that many of the performance-enhancing features of our software coherence protocol could be of use in programmable cache controllers, such as those being developed by the Flash [16] and Typhoon [22] projects. It is not yet clear whether such controllers will be cost effective, but they offer the opportunity to combine many of the advantages of both hardware and software coherence: small coherence blocks and concurrent execution of very complicated protocols. Finally, we believe strongly that behavior-driven coherence, whether implemented in hardware or software, can benefit greatly from compiler support. We are actively pursuing the design of annotations that a compiler can use to provide hints to the coherence system, allowing it to customize its actions to the sharing patterns of individual data structures.

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*, pages 239–261. 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] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 142–153, Chicago, IL, April 1994.

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

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

[10] 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.

[11] 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.

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

[13] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 1992.

[14] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. COMP TR93-214, Department of Computer Science, Rice University, November 1993.

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

[16] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The FLASH Multiprocessor. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 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] 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.

[19] 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.

[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 and K. Li. An Evaluation of Multiprocessor Cache Coherence Based on Virtual Memory Support. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 158–164, Cancun, Mexico, April 1994.

[22] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level Shared-Memory. In *Proceedings of the Twenty-First International Symposium on Computer Architecture*, pages 325–336, Chicago, IL, April 1994.

[23] 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.

[24] 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.

[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.

[26] W. Zwaenepoel, J. Bennett, J. Carter, and P. Keleher. Munin: Distributed Shared Memory Using Multi-Protocol Release Consistency. *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, 5(4):11, Winter 1991.