JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 29, 179–195 (1995)

High Performance Software Coherence for Current and Future Architectures[1]

LEONIDAS I. KONTOTHANASSIS AND MICHAEL L. SCOTT[2]

Department of Computer Science, University of Rochester, Rochester, New York 14627-0226

Shared memory provides an attractive and intuitive programming model for large-scale parallel computing, but requires a coherence mechanism to allow caching for performance while ensuring that processors do not use stale data in their computation. Implementation options range from distributed shared memory emulations on networks of workstations to tightly coupled fully cache-coherent distributed shared memory multiprocessors. Previous work indicates that performance varies dramatically from one end of this spectrum to the other. Hardware cache coherence is fast, but also costly and time-consuming to design and implement, while DSM systems provide acceptable performance on only a limit class of applications. We claim that an intermediate hardware option—memory-mapped network interfaces that support a global physical address space, without cache coherence—can provide most of the performance benefits of fully cache-coherent hardware, at a fraction of the cost. To support this claim we present a software coherence protocol that runs on this class of machines, and use simulation to conduct a performance study. We look at both programming and architectural issues in the context of software and hardware coherence protocols. Our results suggest that software coherence on NCC-NUMA machines in a more cost-effective approach to large-scale shared-memory multiprocessing than either pure distributed shared memory or hardware cache coherence.

1. INTRODUCTION

It is widely accepted that the shared memory programming model is easier to use than the message passing model. This belief is supported by the dominance of (small-scale) shared memory multiprocessors in the market and by the efforts of compiler and operating system developers to provide programmers with a shared memory programming model on machines with message-passing hardware. In the high-end market, however, shared memory machines have been scarce, and with few exceptions limited to research projects in academic institutions; implementing shared memory efficiently on a very large machine is an extremely difficult task. By far the most challenging part of the problem is maintaining cache coherence.

Coherence is easy to achieve on small, bus-based machines, where every processor can see the memory traffic of the others [2, 12]. Coherence is substantially harder to achieve on large-scale multiprocessors [1, 15, 19, 23]; 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. If coherence could be maintained efficiently in software, the resulting reduction in design and development times could lead to a highly attractive alternative for high-end parallel computing. Moreover, the combination of efficient software coherence with fast (e.g., ATM) networks would make parallel programming on networks of workstations a practical reality [11].

Unfortunately, the current state of the art in software coherence for message-passing machines provides performance nowhere close to that of hardware cache coherence. To make software coherence efficient, one would need to overcome several fundamental problems with existing distributed shared memory (DSM) emulations [6, 18, 35]. First, because they are based on messages, DSM systems must interrupt the execution of remote processors in order to perform any time-critical interprocessor operations. Second, because they are based on virtual memory, most DSM systems copy entire pages from one processor to another, regardless of the true granularity of sharing. Third, in order to maximize concurrency in the face of false sharing in page-size blocks, the fastest DSM systems permit multiple writable copies of a page, forcing them to compute diffs with older versions in order to merge the changes [6, 18]. Hardware cache coherence avoids these problems by working at the granularity of cache lines and by allowing internode communication without interrupting normal processor execution.

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

[1] This work was supported in part by NSF Institutional Infrastructure Grant CDA-8822724 and ONR Research Grant 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 8930.

[2] E-mail: {kthanasi,scott}@cs.rochester.edu.

0743-7315/95 $12.00

include the Cray Research T3D and the Princeton Shrimp [4]. In comparison to hardware-coherent machines, NCC-NUMAs can more easily be built from commodity parts, with only a small incremental cost per processor for large systems and can follow improvements in microprocessors and other hardware technologies closely. In another paper [20], we show that NCC-NUMAs provide performance advantages over DSM systems ranging from 50% to as much as an order of magnitude. On the downside, our NCC-NUMA protocols require the ability to control a processor's cache explicitly, a capability provided by many but not all current microprocessors.

In this paper, 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 nonreplicated 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.

For the purposes of this paper we define a system to be hardware coherent if coherence transactions are handled by a system component other than the main processor. Under this definition hardware coherence retains two principal advantages over our protocol. It is less susceptible to false sharing because it maintains coherence at the granularity of cache lines instead of pages, and it is faster because it executes protocol operations in a cache controller that operates concurrently with the processor. Current trends, however, are reducing the importance of each of these advantages. Relaxed consistency models mitigate the impact of false sharing by limiting spurious coherence operations to synchronization points, and programmers and compilers are becoming better at avoiding false sharing as well [9, 14].

While any operation implemented in hardware is likely to be faster than equivalent software, particularly complex operations cannot always be implemented in hardware at reasonable cost. Software coherence therefore enjoys the potential advantage of being able to employ techniques that are too complicated to implement reliably in hardware at acceptable cost. It 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.[3] We exploit this advantage in part in our work by employing a relatively complicated eight-state protocol, by using uncached 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. We are exploring additional protocol enhancements (not reported here) that should further improve the performance of software coherence.

The rest of the paper is organized as follows. We present our software protocol in Section 2. We then describe our experimental methodology and application suite in Section 3 and present results in Section 4. We compare our protocol to a variety of existing alternatives, including release-consistent hardware, straightforward sequentially-consistent software, and a coherence scheme for small-scale NCC-NUMAs due to Petersen and Li [26]. We show that certain simple program modifications can improve the performance of software coherence substantially. Specifically, we identify the need to mark reader–writer locks, to avoid certain interations between program synchronization and the coherence protocol, to align data structures with page boundaries whenever possible, and to use uncached references for certain fine grained shared data structures. With these modifications in place, our protocol performs substantially better than the other software schemes, enough in most cases to bring software coherence within sight of the hardware alternatives—for three applications slightly better, usually only slightly worse, and never more than 55% worse.

In Section 5, we examine the impact of several architectural alternatives on the effectiveness of software coherence. We study the choice of write policy (write-through, write-back, write-through with a write-merge buffer) for the cache and examine the impact of architectural parameters on the performance of software and hardware coherence. We look at page and cache line sizes, overall cache size, cache line invalidate and flush costs, TLB management and interrupt handling costs, and network/memory latency and bandwidth. Our experiments document the effectiveness of software coherence for a wide range of hardware parameters. Based on current trends, we predict that software coherence can provide an even more cost-effective alternative to hardware-based cache coherence on future generations of machines. The final two sections of the paper discuss related work and summarize our conclusions.

2. A SCALABLE SOFTWARE CACHE COHERENCE PROTOCOL

In this section, we present a protocol for software cache coherence on large-scale NCC-NUMA machines. As in most software coherence systems, we use virtual memory protection bits to enforce consistency at the granularity of pages. As in Munin [6], Treadmarks [18], and the work of Petersen and Li [26], we allow more than one processor to write a page concurrently, and we use a variant of release consistency [23] to limit coherence operations to synchro-

[3] The advent of protocol processors [21, 28] can make it possible to combine the flexibility of software with the speed and parallelism provided by hardware.

nization 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 novelty of our protocol lies in the mechanism used to maintain and propagate directory information. Most of the information about each page is located at the (unique) processor in whose memory the page can be found (this is the page's *home node*). The information includes a list of the current readers and writers of the page, and an indication of the page's *state*, which may be one of the following: *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 it. A page leaves the weak state and becomes uncached when no processor has a mapping to the page anymore.

The state of a page is a property of the system as a whole, not (as in most procotols) the viewpoint of a single processor. Borrowing terminology from Platinum [10], the distributed data structure consisting of this information stored at home nodes is called the *coherent map*.

In addition to its portion of the coherent map, each processor also holds a local *weak list* that indicates which of the pages for which there are local mappings are currently in the weak state. 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 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 the page. It then unlocks the entry in the coherent map. On an acquire operation, a processor must remove all mappings and purge from its cache all lines of all pages found in its local weak list. It must also update the coherent map entries of the pages it invalidates to reflect the fact that it no longer caches these pages.

At first glance, one might think that modifying the coherent map with uncached memory references would be substantially more expensive than performing a directory operation on a machine with hardware cache coherence. In reality, however, we can fetch the data for a directory entry into a processor's cache and then flush it back before the lock is released. If lock operations are properly designed we can also hide the latency for the data transfers behind the latency for the lock operations themselves. If we employ a distributed queue-based lock [25], a read of the coherent map entry can be initiated immediately after starting the fetch-and-store operation that retrieves the lock's tail pointer. If the fetch-and-store returns nil (indicating that the lock was free), then the data will arrive right away. The write that releases the lock can subsequently be pipelined immediately after the write of the modified data, and the processor can continue execution. If the lock is held when first requested, then the original fetch-and-store will return the address of the previous processor in line. The queue-based lock algorithm will spin on a local flag, after writing that flag's address into a pointer in the predecessor's memory. When the predecessor finishes its update of the coherent map entry, it can write the data directly into the memory of the spinning processor, and can pipeline immediately afterward a write that ends the spin. The end result of these optimizations is that the update of a coherent map entry requires little more than three end-to-end message latencies (two before the processor continues execution) in the case of no contention. When contention occurs, little more than *one* message latency is required to pass both the ownership of the lock and the data the lock protects from one processor to the next. Inexpensive update of remote weak lists is accomplished in the same manner.

Additional optimizations are possible. When a processor takes a page fault on a write to a shared (nonweak) page, we could choose to make the transition to weak and post appropriate write notices immediately or, alternatively, we could wait until the processor's next release operation: the semantics of release consistency do not require us to make writes visible before then. Similarly, a page fault on a write to an unmapped page could take the page to the dirty state immediately, or at the time of the subsequent release. 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. The disadvantage is that delayed transitions may lengthen the critical path of the computation by introducing contention, especially for programs with barriers, in which many processors may attempt 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 shown to improve performance in the Munin distributed shared memory system [6], which runs on networks of workstations and communicates solely via messages. Though the relative costs of operations are quite different, experiments indicate (see section 4) that delayed transitions are generally beneficial in our environment as well.

As described thus far, our protocol incurs at each release point the cost of updating the coherent map and (possibly) posting write notices for each page that has been modified by the processor performing the release. At each acquire point the protocol incurs the cost of invalidating (unmapping and flushing from the cache) any locally accessible pages that have been modified recently by other processors. Whenever an invalidated page is used again, the protocol incurs the cost of fielding a page fault, modifying the coherent map, and reloading any accessed lines. (It also incurs the cost of flushing the write-merge buffer at releases, but this is comparatively minor.) In the aggregate, each processor pays overhead proportional to the number

of pages it is actively sharing. By comparison, a protocol based on a centralized weak list requires a processor to scan the entire list at a lock acquisition point, incurring overhead proportional to the number of pages being shared by *any* processors.

In reality, most pages are shared by a modest number of processors for the applications we have examined, and so local weak lists make sense when the number of processors is large. Distributing 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). For poorly structured programs, or for the occasional widely shared page in a well-structured program, a central weak list would make sense: it would replace the serialized posting of many write notices at a release operation with individual checks of the weak list on the part of many processors at acquire operations. To accommodate these cases, we modify our protocol to adopt the better strategy, dynamically, for each individual page.

Our modification takes 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 eight 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 it 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 programs with low degrees of sharing (up to 5% improvement in our application suite). It is more effective for programs with pages shared across a large number of processors (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.[4]

[4] In future work, we intend to investigate protocol in which processors always invalidate unsafe pages at an acquire operation, without checking the coherent map. This protocol may incur less overhead for coherent map operations, but will perform some unnecessary invalidations, and provides no obvious mechanism by which an unsafe page could be reclassified as safe.

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 work best on small machines, 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 under three different assumptions: write-through caches where each individual write is immediately sent to memory, write-back caches with per-word hardware dirty bits in the cache, and write-through caches with a write-merge buffer [7] that hangs onto recently written lines and coalesces any writes that are directed to the same line. The write-merge buffer also requires per-word dirty bits to make sure that falsely shared lines are merged correctly. 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 acknowledgements of write-throughs. Our experiments (see Section 5.1) indicate that performance is generally best with write-back for private data and write-through with write-merge for shared data.

The state diagram for a page in our protocol appears in Fig. 1. 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.

3. 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 [34], 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. For the software-coherent systems we assume that all data

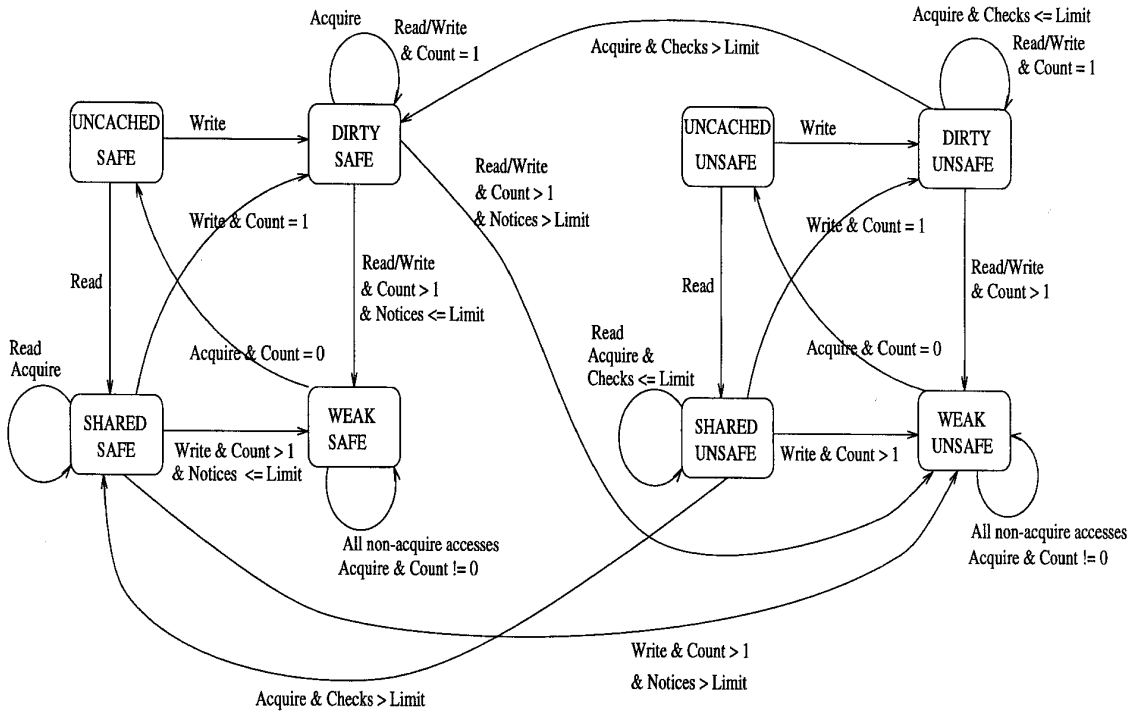

FIG. 1. Scalable software cache coherence state diagram.

transfers are completed without software intervention. The only software operations are modifying coherent map entries and updating remote weak lists. Table I summarizes the default parameters used in our simulations. For these parameters remote cache miss latencies are approximately 100 cycles, while misses to local memory are significantly cheaper (20 cycles).

Some of the transactions required by our coherence protocols require a collection of the operations shown in Table I 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 (24 cycles), (b) a processor interrupt caused by the absence of read rights (140 cycles), (c) a coherent map entry lock acquisition, and (d) a coherent map entry modification followed by the lock release (160 cycles). Lock acquisition itself requires traversing the network and accessing the memory module where the lock is located. Assuming that accessing the lock requires traversing 10 intermediate nodes, that there is no contention in the network, and that the lock is found to be free, the cost for lock acquisition is the roundtrip latency of the network plus the memory access cost, or $(2 + 1) * 10 * 2 + 12 + 1 = 73$ cycles. The total cost for the above transaction would then be $24 + 140 + 73 + 160 = 398$ cycles.

We report results for six parallel programs. We have run each application on the largest input size that could be simulated in a reasonable amount of time and that provided good load-balancing for a 64-processor configuration, which is the largest machine we simulate. Of our applications, 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 a one-dimensional FFT on a 65,536-element

TABLE I
Default Values for System Parameters

System constant name	Default value
TLB size	128 entries
TLB fault service time	24 cycles
Interrupt cost	140 cycles
Coherent map modification	160 cycles
Memory setup time	12 cycles
Memory bandwidth	4 bytes/cycle
Page size	4K bytes
Total cache per processor	128K bytes
Cache line size	32 bytes
Network path width	16 bits (bidirectional)
Switch latency	2 cycles
Wire latency	1 cycle
Directory lookup cost	10 cycles
Cache purge time	1 cycle/line

array of complex numbers. `Mp3d` and `water` are part of the SPLASH suite [33]. `Mp3d` is a wind-tunnel airflow simulation. We simulated 40,000 particles for 10 steps in our studies. `Water` is a molecular dynamics simulation computing inter- and intramolecule forces for a set of water molecules. We used 256 molecules and 3 time steps. Finally `appbt` is from the NASA parallel benchmarks suite [3]. 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. We have also chosen smaller caches than are common on real machines, in order to capture the effect of capacity and conflict misses. Our caches are still large enough to hold the working set of our applications, with capacity and conflict misses being the exception rather than the rule. The main reason for this choice is the desire to evaluate the impact of protocol performance on the applications rather than just remote memory latency. Section 5.4 studies the impact of cache size on the relative performance of our protocols. Since we still observe reasonable scalability for most of our applications, we believe that the data set sizes do not compromise our results.[5]

4. PERFORMANCE 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 presenting our applications and the changes we made to improve their performance on a software coherence protocol. We continue in Section 4.2 by evaluating the trade-offs between different software protocols. Finally, in Section 4.3, we compare the best of the software results to the corresponding results on release-consistent hardware.

4.1. Program Modifications to Support Software Cache Coherence

In this section, we show that programming for software-coherent systems requires paying attention to the same issues that are important for hardware-coherent environments, and that simple program changes can greatly improve program performance. Most of the applications in our suite were written with a small coherence block in mind, which could unfairly penalize software-coherent systems. These applications could easily be modified, however, to work well with large coherence blocks. Furthermore we show that the flexibility of software coherence can allow for optimization that may be too hard to implement in a hardware-coherent system and that can further improve performance.

[5] Mp3d does not scale to 64 processors, but we use it as a stress test to compare the performance of different coherence mechanisms.

Our program modifications are also beneficial for hardware-coherent systems; several are advocated in the literature [16]. Our contribution lies in quantifying their impact on performance in the context of a software coherent system and attributing the performance loss observed in the unmodified applications to specific interactions between the application and the coherence protocol. The remaining optimizations take advantage of program semantics to give hints to the coherence protocol on how to reduce coherence management costs and are applicable only in the context of the software protocols. Our four modifications are:

- Separation of synchronization variables from other writable program data (*Sync-fix*).
- Data structure alignment and padding at page or subpage boundaries (*pad*).
- Identification of reader-writer locks and avoidance of coherence overhead when releasing a reader lock (*RW-locks*).
- Identification of fine grained shared data structures and use of uncached references for their access, to avoid coherence management (*R-ref*).

All our changes produced dramatic improvements on the runtime of one or more applications, with some showing improvements of well over 50% under our software coherence protocols. Results for hardware-based systems (not shown here) also reveal benefits from these program changes, but to a lesser degree, with `mp3d` showing the largest improvement, at 22%.

Colocation of application data and locks on software coherent systems severely degrades performance due to 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.

Data structure alignment and padding are well-known methods of reducing false sharing [16]. 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

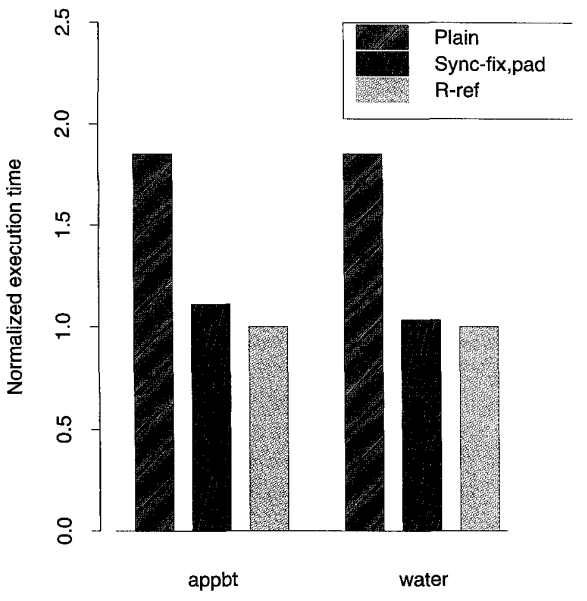

FIG. 2. Normalized runtime of `appbt` and `water` with different levels of restructuring.

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 (see the *sort* bar in Fig. 3 below).

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. 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* [35], 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 may be 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 *uncached reference*. We expect this annotation to be effective only when used on a very small percentage of a program's references to shared data.

The performance improvements for our four modified applications when running under the protocol described in Section 2 can be seen in Figs. 2 and 3. The performance impact of each modification is not independent of previous changes; the graphs show the aggregate performance improvement for each successive optimization.

As can be seen from the graphs, `Gauss` improves markedly when relocating synchronization variables to fix the lock interference problem and also benefits from the identification of reader–writer locks. Uncached reference helps only a little. `Water` gains most of its performance improvement by padding the molecule data structures to subpage 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 uncached reference shaves off another 50%. Finally, `appbt` sees dramatic improvements after relocating one of its data structures to achieve good

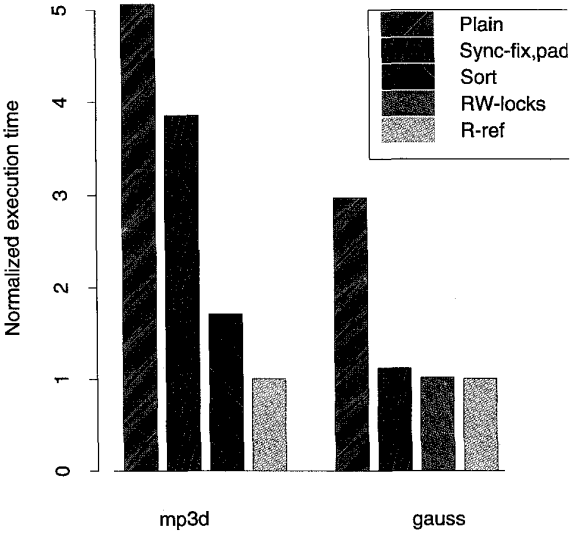

FIG. 3. Normalized runtime of `gauss` and `mp3d` with different levels of restructuring.

page alignment and benefits nicely from the use of uncached references as well. The performance of the remaining two programs in our application suite was insensitive to the changes described here.

Our program changes were simple: identifying and fixing the problems was a mechanical process that consumed at most a few hours. The process was aided by simulation results that identified pages with particularly high coherence overhead. In practice, similar assistance could be obtained from performance monitoring tools. The most difficult application 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.

4.2. Software Coherence Protocol Alternatives

This section compares our software protocol (presented in Section 2) to the protocol devised by Petersen and Li [26] (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. The TLB/page table dirty bits do not suffice to drive the protocol 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.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 of Petersen and Li [26], with the exception that while the weak list is conceptually centralized, its entries are distributed physically among the nodes of the machine.

`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.

`seq`: A sequentially consistent software protocol that allows only a single writer for every page 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 underestimate the cost of coherence management but since it is the worst protocol in most cases, the inaccuracy has no effect on our conclusions.

Figure 4 presents the normalized execution time of the different software protocols on our set of partially modified applications. We have used the versions of the applications whose data structures are aligned and padded, and whose synchronization variables are decoupled from the data they protect (see Section 4.1). We have *not* used the versions that require annotations: the identification of reader locks or of variables that should be accessed with uncached references. The distributed protocols outperform the centralized implementations, often by a significant margin. The largest improvements (almost threefold) are realized on `water` and `mp3d`, the two applications for which software coherence lags the most behind hardware coherence (see Section 4.3). This is predictable behavior: applications in

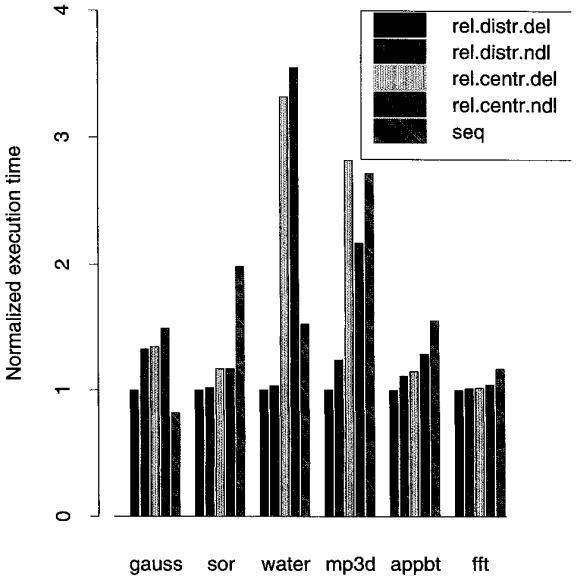

FIG. 4. Comparative performance of different software protocols on 64 processors.

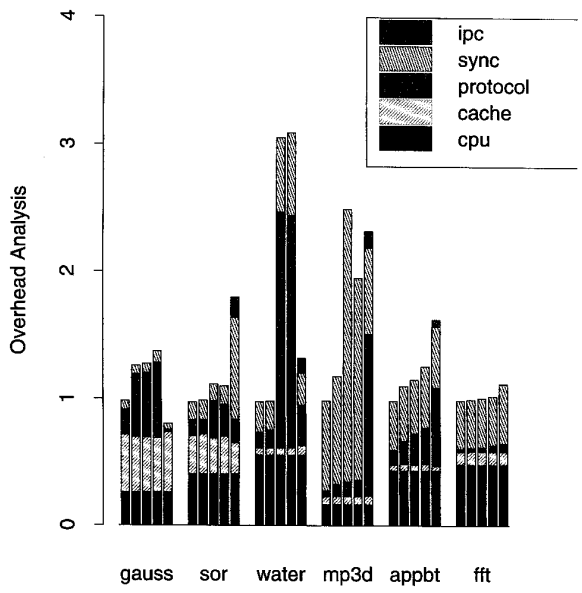

FIG. 5. Overhead analysis of different software protocols on 64 processors.

which the impact of coherence is important are expected to show the greatest variance with different coherence algorithms. However, it is important to note the difference in the scales of Figs. 4 and 6. While the distributed protocols improve performance over the centralized ones by a factor of three for `water` and `mp3d`, they are only 38 and 55% worse than their hardware competitors. In programs in which coherence is less important, the decentralized protocols still provide reasonable performance improvements over the centralized ones, ranging from 2 to 35%.

It is surprising to see the sequentially consistent protocol outperform the relaxed alternatives on `gauss`. The explanation lies in the use of locks as flags, as described in Section 4.1. It is also surprising to see the sequentially consistent protocol outperform the centralized relaxed protocols on `water`. The main reason is that restructuring has reduced the amount of false sharing in the program, negating the main advantage of relaxed consistency, and the sharing patterns in the program force all shared pages into the weak list, making processors pay very high penalties at lock-acquisition points.

While run time is the most important metric for application performance it does not capture the full impact of a coherence algorithm. Figure 5 shows the breakdown of overhead into its major components for the five software protocols on our six applications. These components are: interrupt handling overhead (*ipc*) (sequentially consistent protocol only), time spent waiting for application locks (*sync*), coherence protocol overhead (including waiting for system locks and flushing and purging cache lines) (*protocol*), time spent waiting for cache misses (*cache*), and useful processing cycles (*cpu*). 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 are slightly off in Figs. 4 and 5, because the former pertains to the parallel part of the computation, while the latter includes initialization overheads as well. Since initialization overheads were small the differences between the relative heights of bars in the graphs are minor. As can be seen from the graph, cache wait time is virtually identical for the relaxed consistency protocol. This is consistent with the fact that the protocols use the same rules for identifying which pages are weak and therefore invalidate the same pages. The performance advantage of the distributed protocols stems from reduced protocol and synchronization overhead.

4.3. Hardware vs Software Coherence

Figure 6 shows the normalized execution times of our best software protocol and that of a relaxed-consistency DASH-like hardware protocol [23] on 64 processors. Time is normalized with respect to the software protocol. The hardware protocol assumes single-processor nodes, and the consistency model allows reads to bypass writes in the write-buffers. Only one write-miss request can be outstanding at any point time; subsequent writes queue in the write buffer. If the write buffer capacity is exceeded the processor stalls. The 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 4.1, though uncached reference is used only in the context of software coherence; it does not make sense in the hardware-coherent case.

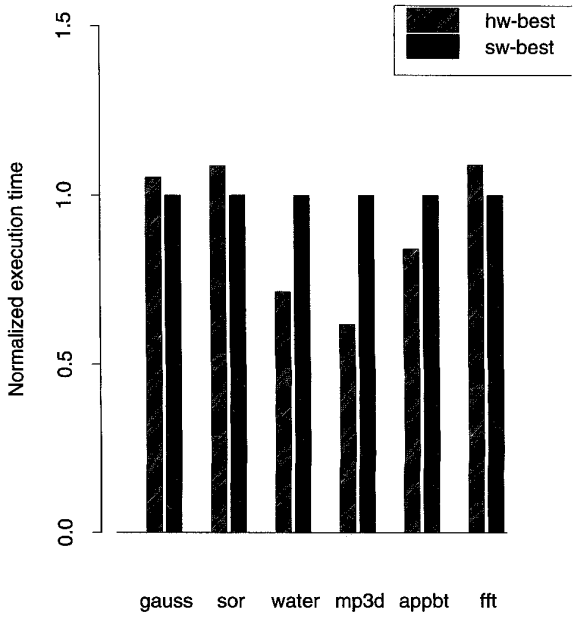

FIG. 6. Comparative software and hardware system performance on 64 processors.

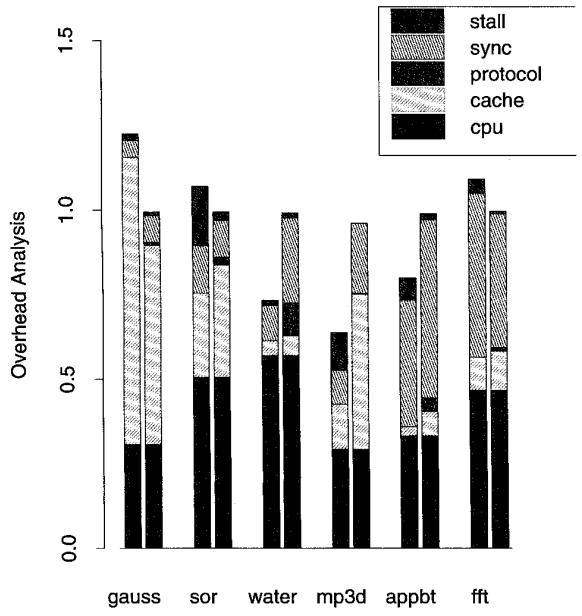

FIG. 7. Overhead analysis of software and hardware protocols on 64 processors.

In all cases, with the exception of `mp3d`, the performance of the software protocol is within 40% of the relaxed consistency hardware protocol (termed *hw-best* in our graphs). For three of the applications, the software protocol is actually slightly faster. The write-through mode eliminates three-hop transaction on cache misses reducing the miss overhead. One can argue that the hardware protocol could also use a write-through cache, but that would be detrimental to the performance of other applications. The software-based protocol also does not need to get write-access rights for each cache line. As a result writes retire immediately if the cache line is present, and write-buffer stall time is reduced.

On the other hand `mp3d` and `water` demonstrate cases in which software coherence has disadvantages over a hardware implementation. For `water` the main problem is the extremely high frequency of synchronization, while for `mp3d` the presence of both fine-grained sharing and frequent synchronization affects performance. Still the performance of `water` is within 38% of the hardware implementation. `Mp3d` on 64 processors is actually 55% worse under software coherence, but this program is known for its poor scalability. We regard its execution on 64 processors as more of a "stress test" for the protocols than a realistic example. On 16 processors (a point at which reasonable speedups for `mp3d` may still be attainable), the performance of the software protocol is only 33% worse than that of the hardware protocol. The 16-processor graphs are not shown here due to lack of space.

Figure 7 shows the breakdown of overhead for the two protocols into its major components. These components are: write-buffer stall time (*stall*), synchronization overhead (*sync*), protocol processing overheads (*protocol*), cache miss latency (*cache*) which for the software protocol also includes time spent in uncached references, and useful cpu cycles (*cpu*). Results indicate that the coherence overhead induced by our protocol is minimal, and in most cases the larger coherence block size does not cause any increase in the miss rate and consequently the miss latency experienced by the programs. Table II shows the miss rates and other categories of overhead for the programs in our application suite. The left number in the "Miss rate" column corresponds to the miss rate for the hardware-coherent system, while the right number corresponds to the software-coherent system. For the applications that exhibit a higher miss rate for the hardware system the additional misses come mainly from the introduction of exclusive requests and from a slight increase in conflict misses (use of uncached reference reduced the working set size for the software protocol). The results in the table correlate directly with the results shown in Fig. 7. Higher miss rates result in higher miss penalties, while the low page miss rate is in accordance with the low protocol overhead experienced by the applications.

5. THE IMPACT OF ARCHITECTURE ON COHERENCE PROTOCOLS

Our second goal in this work is to study the relative performance of hardware and software coherence across a wide variety of architectural settings. We start in Section 5.1 by examining the impact of the write policy on the performance of software coherence. We then proceed to examine a variety of architectural parameters that affect the performance of parallel applications. These include

TABLE II
Application Behavior under Software and Hardware Protocols

Application	Refs $\times 10^6$	Miss rate (HW, SW)	Page miss rate	Uncached refs
`gauss`	91.5	6.3%, 5.7%	0.03%	≈0%
`sor`	20.6	3.6%, 2.7%	0.02%	0%
`water`	249.9	0.34%, 0.33%	0.05%	≈0%
`mp3d`	73.1	3.2%, 1.8%	0.01%	3.3%
`appbt`	281.5	0.35%, 0.71%	0.04%	0.25%
`fft`	209.2	0.87%, 0.65%	0.01%	0%

page and cache line granularity, cache size, cache line invalidate and flush costs, TLB management and interrupt handling costs, and network/memory latency and bandwidth. The results of this study suggest that software coherence on NCC-NUMA machines may become an even more attractive alternative to hardware cache coherence for future multiprocessors.

5.1. 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 [7]. The policy is applied on only shared data. Private data uses by default a write-back policy.

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 best way to achieve 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.

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. 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 [7] employ a small 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 nonresident cache line. When it runs out of entries it randomly chooses a line for eviction and writes it back to memory. The write-merge buffer 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. In our experiments, we have used a 16-entry write-merge buffer.

Figure 8 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 provides the best performance. In both cases however the performance of the write-through cache with the write-merge buffer is within 5% of the better alternative.

We also looked at the impact of the write policy on main memory system performance. Both write-through policies generate more traffic and thus have the potential to deteriorate memory response times for other memory operations. Figure 9 presents the average cache miss latency under different cache policies. As can be seen the write-through cache with the write-merge is only marginally worse in this metric than the write-back cache. The plain write-through cache creates significantly more traffic, thus causing a much larger number of cache misses to be delayed and increasing miss latency.

Finally, we 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 significantly, with plain write-through being as much as 50 times worse in `water`.

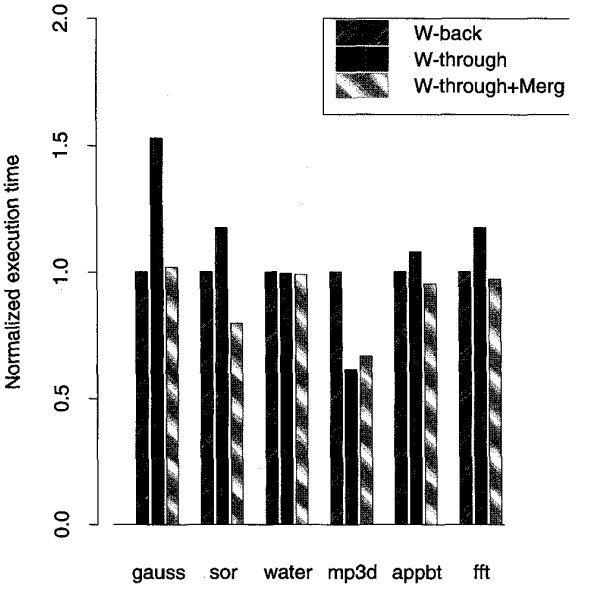

FIG. 8. Comparative performance of different cache architectures on 64 processors.

Application	W-Back	W-Through	W-Merge
Gauss	112.1	228.2	115.7
Sor	93.3	139.4	89.9
Water	84.9	111.6	85.3
Mp3d	62.8	190.2	63.8
Appbt	67.3	96.7	67.5
Fft	88.3	156.2	88.3

FIG. 9. Average read miss latency in cycles for different cache types.

5.2. Page Size

The choice of page size primarily affects the performance of software coherence, since pages are the unit of coherence. Hardware coherence may also be affected, due to the placement of pages in memory modules, but this is a secondary effect we have chosen to ignore in our study. Previous studies on the impact of page size on the performance of Software DSM systems [5] indicate that the smaller pages can provide significant performance improvements. The main reason for this result is the reduction in false sharing achieved by smaller coherence units. Moving to relaxed consistency, however, and to an architecture that uses pages for the unit of coherence but cache lines for the data fetch unit, reverses the decision in favor of large pages [26]. Relaxed consistency mitigates the impact of false sharing, and the larger page size reduces the length of the weak list that needs to be traversed on an acquire operation.

In our protocol, the absence of a centralized weak list removes one of the factors (length of the weak list) that favors larger pages. Furthermore the choice among mechanisms for data access (caching vs uncached reference) can only be made at page granularity. Smaller pages can make for more accurate decisions. On the other hand smaller pages will require a larger number of coherence transactions to maintain the same amount of shared data. When true sharing is the reason for coherence transactions, smaller pages will induce unnecessary overhead.

To evaluate the relative impact of these effects, we have run a series of experiments, varying the page size from as small as 256 bytes to as large as 16K bytes. Figure 10 shows the normalized running time of our applications as a function of page size. The normalization is with respect to the running time of the relaxed consistency hardware system using 4K-byte pages. We observe that performance improves as the page size increases for all applications. For four of our applications a 4K-byte page size provides the best performance; for the other two performance continues to improve marginally as the page size increases even further. The applications that degrade after the 4K-byte point have been restructured to work well under software coherence, with data structures aligned on 4K-byte boundaries. It is reasonable to assume that for larger data-set sizes (for which alignment to large page sizes is feasible) performance would keep improving with an increase in page size. For the unmodified versions of the programs (not shown) smaller page sizes provided a small performance advantage over larger ones, but the overall performance of software coherence was not in par with that of hardware coherence.

5.3. Cache Line Size

The choice of cache line size affects hardware and software coherence in similar ways. Increases in cache line size reduce the miss rate, by taking advantage of the spatial locality in programs. However when the cache line size gets too large it has the potential to introduce false sharing in hardware coherent systems, and unnecessary data transfers in software coherent systems. Furthermore, larger lines cause higher degrees of contention for memory modules and the network interconnect since they have longer occupancy times for those resources.

We have run experiments in which the cache line size varies between 16 and 256 bytes for both release-consistent hardware and our software coherence protocol. Figure 11 shows the normalized running times of our applications as

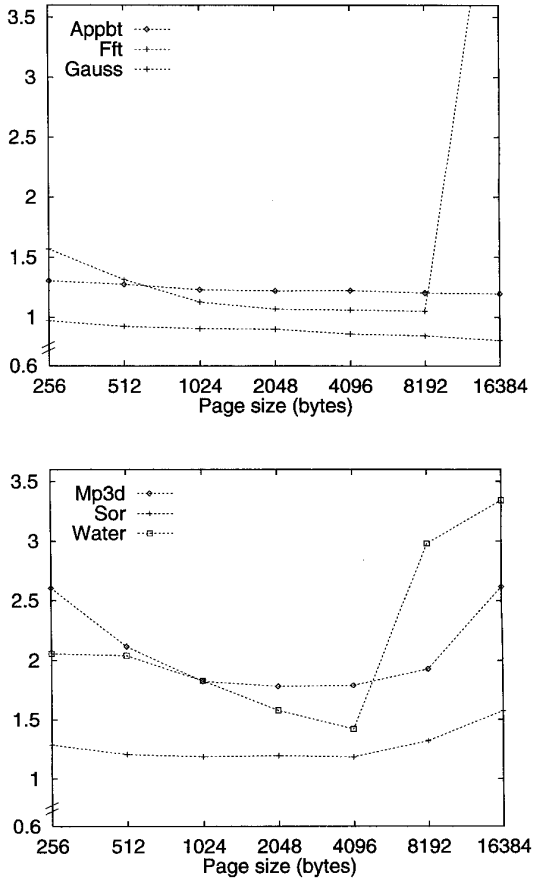

FIG. 10. Normalized execution time for our applications using different page sizes.

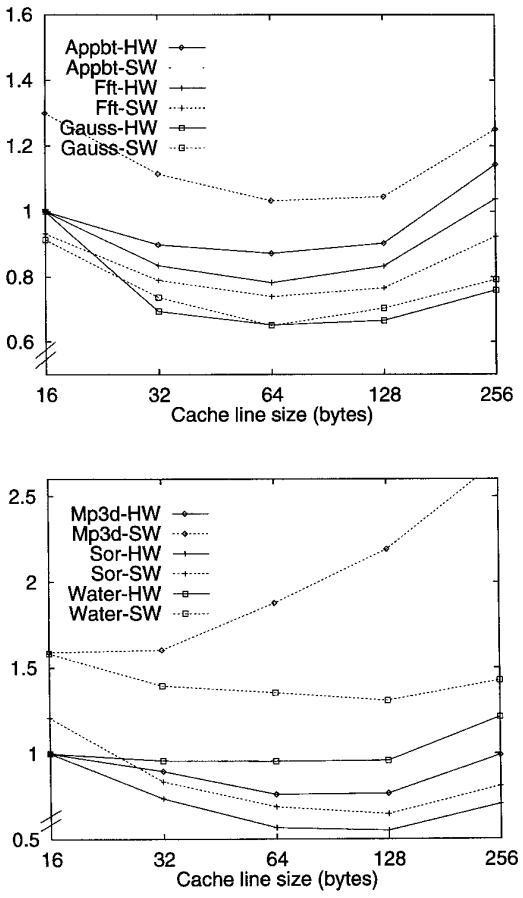

FIG. 11. Normalized execution time for our applications using different cache line sizes.

a function of cache line size. (Running time is normalized with respect to the hardware system with 16-byte cache lines.) Performance initially improves for both systems as the cache line size increases, with the optimal point occurring at either 64- or 128-byte lines. We believe that the degradation seen at larger sizes is due to a lack of bandwidth in our system. We note that the performance improvements seen by increasing the line size are progressively smaller for each increase. The exception to the above observations is `mp3d` under software coherence, where increases in line size hurt performance. The reason for this anomalous behavior is the interaction between cache accesses and uncached references. Longer cache lines have higher memory occupancy and therefore block uncached references for a longer period of time. The uncached references in the software-coherent version of the program end up waiting behind the large line accesses, degrading overall performance.

5.4. Cache Size

Varying the cache size allows us to capture how the different protocols handle conflict/capacity misses. Smaller caches increase the number of evictions, while large caches reduce misses to the intrinsic rate (that which is due to sharing and coherence) of the program in question. While there is no universal agreement on the appropriate cache size for simulations with a given data set size, recent work confirms that the relative sizes of per-processor caches and working sets are a crucial factor in performance [29]. All the results in previous sections were obtained with caches sufficiently large to hold the working set, in order to separate the performance of the coherence protocols from the effect of conflict/capacity misses. Experience suggests, however, that programmers write applications that exploit the amount of available memory as opposed to the amount of available cache. When the working set exceeds the cache size, the handling of conflict/capacity misses may have a significant impact on performance.

To assess this impact we have run experiments in which the cache size varies between 8K and 128K bytes for both the hardware and software coherent systems. While these numbers are all small by modern standards, they span the border between "too small" and "large enough" for our experiments. Figure 12 shows the normalized running time of our applications as a function of cache size. Running time is normalized with respect to the hardware system

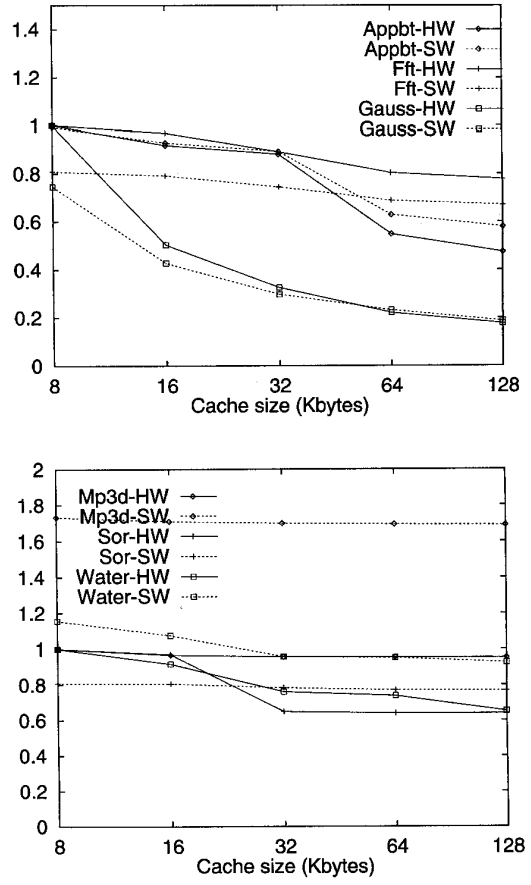

FIG. 12. Normalized execution time for our applications using different cache sizes.

with an 8K-byte cache size. The results show the performance of hardware coherence improving more quickly than that of software coherence with increases in cache size. This is to be expected, since the hardware system was shown in previous sections to handle coherence-related communication more efficiently than the software system. For smaller caches, where conflict/capacity misses constitute a larger fraction of the total miss rate, the more complicated directory structure of the hardware system (with ownership and forwarding) imposes a higher penalty on refills than is suffered by the software system. As the cache size increases the elimination of conflict/capacity misses improves the performance of both systems, with the hardware system enjoying the largest benefits.

5.5. Processor Constants

The performance of software coherent systems is also dependent on the cost of cache management instructions, interrupt handling, and TLB management. We evaluated the performance of our application suite under a variety of assumptions for these costs, but found very little variation. Specifically, we ran experiments with the following range of values:

Cache purge: 1, 2, 4, or 6 cycles to purge a line from the cache.

Interrupt handling: 40, 140, or 500 cycles between interrupt occurrence and start of execution of the interrupt handler. These values represent the expected cost of an interrupt for a very fast interrupt mechanism (e.g., Sparcle [1], a normal one, and a particularly slow one.

TLB Management: 24, 48, or 120 cycles for tlb service fault. These values represent the expected cost of a tlb fill when done in fast hardware, somewhat slower hardware, or entirely in software.

Across this range, the largest performance variation displayed by any application was less than 3% (for `water`). Statistics gathered by the simulator confirm that the cache, interrupt, and TLB operations are simply not very frequent. As a result, their individual cost, within reason, has a negligible effect on performance. (We have omitted graphs for these results; they are essentially flat lines.)

5.6. Latency and Bandwidth

The final dimension of our architectural study is memory and interconnect latency and bandwidth. Latency and bandwidth directly affect the cost of memory references and coherence protocol transactions. Current technological trends indicate that memory (DRAM) latencies and bandwidth will continue to increase in comparison to processor speeds. Network bandwidths will also continue to increase, with network latencies keeping pace with processors speeds.

We have run experiments varying the memory startup cost between 12 and 30 cycles. Given a cache line size of 8 words and memory bandwidth of 1 word per cycle, the latency for a local cache miss ranges between 20 and 38 cycles. For these experiments, we have kept the network bandwidth at 4 byte/cycle. The results show that the impact of latency variation on the performance of the hardware and software coherence protocols is application dependent. In general, the steepness of the curves is directly related to the miss rates of the applications under each protocol: the higher the miss rate the more sensitive the application is to an increase in latency. The exception to the observation is `mp3d`. Although the software protocol has a lower miss rate than the hardware alternative, it performs more uncached references, and these are also susceptible to the increase in memory latency. Figure 13 shows the normalized execution time of our applications under hardware and software coherence for different memory latencies. Running time is normalized with respect to the hardware system with a 12-cycle memory startup cost.

Bandwidth increases have the opposite effect of increases in latency on the performance of hardware and software coherence. Where an increase in latency tends to exaggerate the performance discrepancies between the

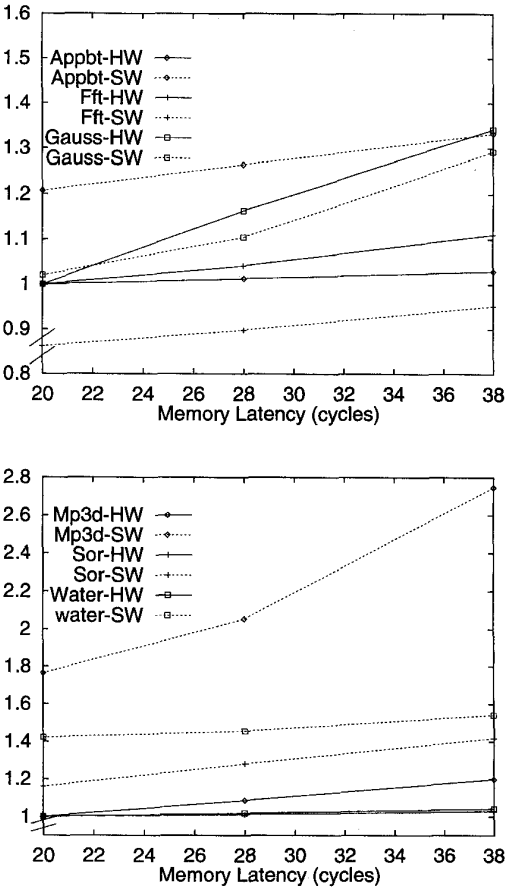

FIG. 13. Normalized execution time for our applications under different memory latencies.

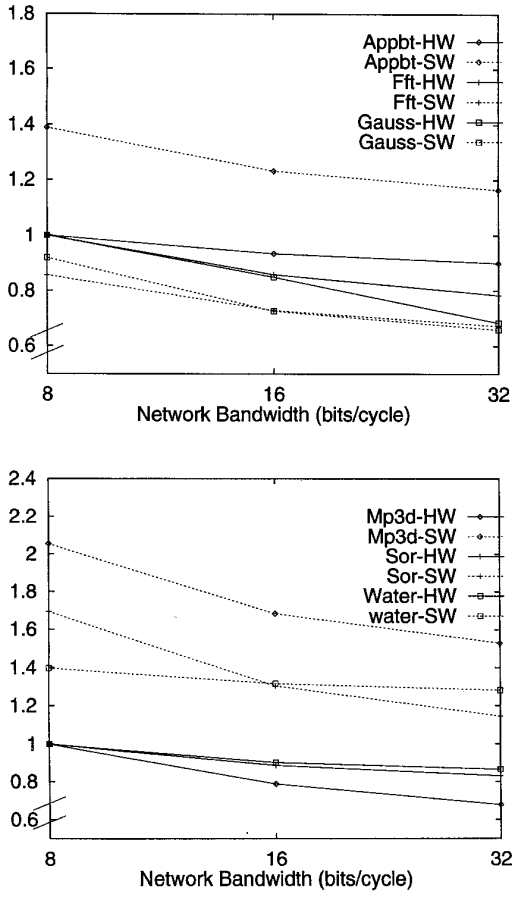

FIG. 14. Normalized execution time for our applications for different network bandwidths.

protocols, an increase in bandwidth softens them. We have run experiments varying network (point-to-point) and memory bandwidth between 1 and 4 bytes per cycle—varying them together to maintain a balanced system. Startup cost for memory access was set to 30 cycles. Figure 14 shows the normalized execution times of the applications under different bandwidth values. Running time is normalized with respect to the hardware system with a 1 byte/cycle memory/network bandwidth. Once again, in general, the steepness of the curve is directly related to the miss rates of the applications under each protocol. Higher miss rates imply a need for communication, which can best be met when a lot of bandwidth is available. `Water`, which has a very low communication to computation ratio, exhibits a relatively flat line; consistent with what we saw in almost all of the graphs, it has little use for long cache lines, low latency, or high bandwidth. `Mp3d` under software coherence once again fails to make efficient use of extra available bandwidth. The program suffers more from the cost of initiating uncached references than from a lack of bandwidth: its performance improves only moderately as bandwidth increases.

6. RELATED WORK

Our work is most closely related to that of Petersen and Li [26]: 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 [6] and lazy release consistency [17] 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 uncached references to reduce the overhead of coherence management can also be found in systems for NUMA memory management [5, 10, 22]. Designed for machines without caches, these systems migrate and replicate pages in the manner of distributed shared memory systems, but also make on-line decisions between page movement and uncached reference. We have experimented with dynamic page movement in conjunction with software coherence on NCC-NUMA machines [24], and have found that while appropriate placement of a unique page copy reduces the average cache fill cost appreciably, replication of pages provides no significant benefit in the presence of hardware caches. Moreover, we have found that relaxed consistency greatly reduces the opportunities for profitable uncached data access. 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 uncached references.

On the hardware side, our work bears a resemblance to the Stanford Dash project [23] in the use of a relaxed consistency model and to the Georgia Tech Beehive project [32] 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 overlap coherence processing and 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 [8]. 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. Alternatively, coherence can be maintained in object-oriented systems by tracking method calls or by identifying the specific data structures protected by particular synchronization operations [13, 30, 35]. Such an approach can make it substantially easier for the compiler to implement consistency, but only for restricted programming models.

7. 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 on a machine that provides a non-coherent global physical address space. We have introduced a new protocol for software cache coherence on such machines and have shown that it outperforms existing software approaches, and it is fact comparable in performance to typical schemes for hardware coherence. To improve our confidence in this conclusion, we have explored a wide range of issues that affect the performance of hardware and software coherence.

Our experiments indicate that simple program modifications can significantly improve performance on a software-coherent system, while providing moderate performance improvements for hardware-coherent systems as well. The experiments also show that write-through caches with a write-merge buffer provide the best performance for shared data on a software-coherent system. Most significantly, software coherence on NCC-NUMA machines remains competitive with hardware coherence under a large variety of architectural settings.

Several factors account for these facts. Software coherence admits a level of flexibility and complexity that is difficult to duplicate in hardware. In our experiments, it allows us to use multiple-writer lazy release consistency, to use different protocols for different pages (the safe/unsafe distinction), to forego data migration in favor of uncached references for data shared at a very fine grain, and to skip coherence operations for reader locks. In a more elaborate system, one could image combining multiple protocols such as update-based and migratory. Hardware protocols have the advantage of concurrent protocol and program execution, but this advantage is being eroded by technological trends that are increasing relative data transfer costs. Hardware protocols also have the advantage of smaller block sizes, and therefore less false sharing, but improvements in program structuring techniques, and the use of relaxed consistency, are eroding this advantage too. Recent work also suggests [31, 35] that software-coherent systems may be able to enforce consistency on small blocks efficiently by using binary editing techniques to embed coherence operations in the program text.

The best performance, clearly, will be obtained by systems that combine the speed and concurrency of existing hardware coherence mechanisms with the flexibility of software coherence. This goal may be achieved by a new generation of machines with programmable network controllers [21, 28]. It is not yet clear whether the additional performance of such machines will justify their design time and cost. Our suspicion, based on our results, is that less elaborate hardware will be more cost effective.

We have found the provision of a single physical address space to be crucial for efficient software coherence: it allows cache fills to be serviced in hardware, permits protocol operations to access remote directory information with very little overhead, and eliminates the need to compute diffs in order to merge inconsistent writable copies of a page. Moreover, experience with machines such as the IBM RP3, the BBN Butterfly series, and the current Cray Research T3D suggests that a memory-mapped interface to the network (without coherence) is not much more expensive than a message-passing interface. Memory-mapped interfaces for ATM networks are likely to be available soon [4]; we see our work as ideally suited to machines equipped with such an interface.

We are currently pursuing software 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 coprocessor. We also 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 hints to the coherence system, allowing it to customize its actions to the sharing patterns of individual data structures.

ACKNOWLEDGMENTS

We thank Ricardo Bianchini and Jack Veenstra for their help and support with this work. Our thanks also to the anonymous referees for their detailed and insightful comments.

REFERENCES

1. A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung, The MIT alewife machine: Architecture and performance. *Proceedings of the Twenty-Second International Symposium on Computer Architecture,* Santa Margherita Ligure, Italy, June 1995.
2. J. Archibald and J. Baer, Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Systems* **4**(4), 273–298 (Nov. 1986).
3. D. Bailey, J. Barton, T. Lasinski, and H. Simon, The NAS parallel benchmarks. Report RNR-91-002, NASA Ames Research Center, Jan. 1991.
4. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, Virtual memory mapped network interface for the SHRIMP multicomputer. Proceedings of the Twenty-First International Symposium on Computer Architecture, Chicago, IL, Apr. 1994, pp. 142–153.
5. W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, NUMA policies and their relation to memory architecture. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems,* Santa Clara, CA, Apr. 1991, pp. 212–221.
6. J. B. Carter, J. K. Bennett, and W. Zwaenepoel, Implementation and performance of Munin. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles,* Pacific Grove, CA, Oct. 1991, pp. 152–164.
7. Y. Chen and A. Veidenbaum, An effective write policy for software coherence schemes. *Proceedings Supercomputing '92,* Minneapolis, MN, Nov. 1992.

8. H. Cheong and A. V. Veidenbaum, Compiler-directed cache management in multiprocessors. *Computer* **23**(6), 39–47 (June 1990).
9. M. Cierniak and W. Li, Unifying data and control transformations for distributed shared-memory machines. *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation,* June 1995, La Jolla, CA.
10. A. L. Cox and R. J. Fowler, The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles,* Litchfield Park, AZ, Dec. 1989, pp. 32–44.
11. A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel, Software versus hardware shared-memory implementation: A case study. *Proceedings of the Twenty-First International Symposium on Computer Architecture,* Chicago, IL, Apr. 1994.
12. S. J. Eggers and R. H. Katz, Evaluation of the performance of four snooping cache coherency protocols. *Proceedings of the Sixteenth International Symposium on Computer Architecture,* May 1989, pp. 2–15.
13. M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy, Log-based distributed shared memory. *Proceedings of the First Symposium on Operating Systems Design and Implementation,* Monterey, CA, Nov. 1994.
14. E. Granston, Toward a Compile-Time Methodology for Reducing False Sharing and Communication Traffic in Share Virtual Memory System. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.). *Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science. Springer-Verlag, Berlin/New York, Aug. 1993, pp. 273–289.
15. D. B. Gustavson, The scalable coherent interface and related standards projects. *IEEE Micro* **12**(2), 10–22 (Feb. 1992).
16. M. D. Hill and J. R. Larus, Cache considerations for multiprocessor programmers. *Comm. ACM* **33**(8), 97–102 (Aug. 1990).
17. P. Keleher, A. L. Cox, and W. Zwaenepoel, Lazy release consistency for software distributed shared memory. *Proceedings of the Nineteenth International Symposium on Computer Architecture,* Gold Coast, Australia, May 1992, pp. 13–21.
18. P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, ParaNet: Distributed shared memory on standard workstations and operating systems. *Proceedings of the USENIX Winter '94 Technical Conference,* San Francisco, CA, Jan. 1994.
19. Kendall Square Research, KSR1 principles of operation. Waltham, MA, 1992.
20. L. I. Kontothanassis and M. L. Scott, Distributed shared memory for new generation networks. TR578, Computer Science Department, Univ. of Rochester, Mar. 1995.
21. 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. *Proceedings of the Twenty-First International Symposium on Computer Architecture,* Chicago, IL, Apr. 1994, pp. 302–313.
22. R. P. LaRowe Jr. and C. S. Ellis, Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Trans. Comput. Systems* **9**(4), 319–363 (Nov. 1991).
23. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, The directory-based cache coherence protocol for the DASH multiprocessor. *Proceedings of the Seventeenth International Symposium on Computer Architecture,* Seattle, WA, May 1990, pp. 148–159.
24. M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott, Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. *Proceedings of the Ninth International Parallel Processing Symposium,* Santa Barbara, CA, Apr. 1995.
25. J. M. Mellor-Crummey and M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Systems* **9**(1), 21–65 (Feb. 1991).
26. K. Petersen and K. Li, Cache coherence for shared memory multiprocessors based on virtual memory support. *Proceedings of the Seventh International Parallel Processing Symposium,* Newport Beach, CA, Apr. 1993.
27. Deleted in proof.
28. S. K. Reinhardt, J. R. Larus, and D. A. Wood, Tempest and Typhoon: User-level shared-memory. *Proceedings of the Twenty-First International Symposium on Computer Architecture,* Chicago, IL, Apr. 1994, pp. 325–336.
29. E. Rothberg, J. P. Singh, and A. Gupta, Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. *Proceedings of the Twentieth International Symposium on Computer Architecture,* San Diego, CA, May 1993.
30. H. S. Sandhu, Algorithms for dynamic software cache coherence. *J. Parallel Distrib. Comput.,* to appear.
31. I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, Fine-grain access control for distributed shared memory. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems,* San Jose, CA, Oct. 1994, pp. 297–306.
32. G. Shah and U. Ramachandran, Towards exploiting the architectural features of Beehive. GIT-CC-91/51, College of Computing, Georgia Institute of Technology, Nov. 1991.
33. J. P. Singh, W. Weber, and A. Gupta, SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Comput. Archit. News* **20**(1), 5–44 (Mar. 1992).
34. J. E. Veenstra and R. J. Fowler, MINT: A front end for efficient simulation of shared-memory multiprocessors. *Proceedings of the Second International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 94),* Durham, NC, Jan.–Feb. 1994, pp. 201–207.
35. M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad, Software write detection for distributed shared memory. *Proceedings of the First Symposium on Operating Systems Design and Implementation,* Monterey, CA, Nov. 1994.

Received November 1, 1994; revised April 27, 1995; accepted May 8, 1995

LEONIDAS KONTOTHANASSIS is a Ph.D. candidate in computer science at the University of Rochester, expected to complete his degree in 1995. His research interests include operating systems and runtime environments and their interaction with computer architecture. His thesis research is exploring the design space of shared memory architectures with an emphasis on flexible coherence mechanisms. He is a member of the ACM and IEEE computer societies.

MICHAEL SCOTT is an associate professor of computer science at the University of Rochester. He received his Ph.D. in computer science from the University of Wisconsin at Madison in 1985. His research focuses on systems software for parallel computing, with an emphasis on shared memory programming models. He is the designer of the Lynx programming language and a co-designer of the Psyche parallel operating system. He received an IBM Faculty Development Award in 1986.