Fourth Workshop on Scalable Shared Memory Multiprocessors,
Chicago, IL, April 1994

Issues in Software Cache Coherence

Leonidas I. Kontothanassis and Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{kthanasi,scott}@cs.rochester.edu

Abstract

Large scale multiprocessors can provide the computational power needed to solve some of the larger problems of science and engineering today. Shared memory provides an attractive and intuitive programming model that makes good use of programmer time and effort. Shared memory however requires a coherence mechanism to allow caching for performance and to ensure that processors do not use stale data in their computation. Directory-based coherence, which is the hardware mechanism of choice for large scale multiprocessors, can be expensive both in terms of hardware cost and in terms of the intellectual effort needed to design a correct, efficient protocol. For scalable multiprocessor designs with network-based interconnects, software-based coherence schemes provide an attractive alternative. In this paper we evaluate a new adaptive software coherence protocol, and demonstrate that smart software coherence protocols can be competitive with hardware-based coherence for a large variety of programs. We then discuss issues that affect the performance of software coherence protocols and proceed to suggest algorithmic and architectural enhancements that can help improve software coherence performance.

1 Introduction

It is widely believed that shared memory programming models are easier to use than message passing models. This belief is supported by the dominance of (small-scale) shared memory multiprocessors in the market and the effort by compiler and operating system developers to provide programmers with a shared memory programming abstraction. On the high end market however, shared memory machines have been scarce and with few exceptions restricted to research projects in academic institutions. The main reason for this discrepancy is the difficulty of providing an efficient shared memory abstraction on large scale multiprocessors. By far the most difficult

problem that needs to be addressed is that of cache coherence. Shared memory machines use caches to reduce memory latencies, and thus introduce the coherence problem—the need to ensure that processors do not use stale data in their caches. For large scale multiprocessors with network interconnects, the hardware alternative of choice seems to be directory-based coherence [1, 10], but it is expensive both in terms of hardware cost and in terms of design time and intellectual effort required to produce a correct and efficient implementation. It is therefore not uncommon for technological progress to have rendered a machine outdated by the time it is completed.

Software coherence protocols provide an attractive alternative, with short design and implementation times, albeit at the expense of higher coherence management overheads. There are several reasons to hope that software coherence protocols may be competitive with hardware coherence, despite the higher overhead. First, trap-handling overhead is not very large in comparison to remote communication latencies, and will become even smaller as processor improvements continue to outstrip network improvements. Second, software may be able to embody protocols that are too complicated to implement reliably in hardware at acceptable cost. Third the advent of relaxed consistency models has helped mitigate the impact of false sharing which is caused by the large coherence blocks (pages instead of cache lines) used by most software coherent systems. Finally, programmers and compiler developers are becoming aware of the importance of locality of reference and are writing programs that attempt to minimize the amount of communication between processors, consequently reducing their coherence overhead.

In this paper we explore several algorithmic alternatives in the design space of software cache coherence, targeted for architectures with non-coherent caches and a globally-accessible physical address space. We describe a new, scalable software coherence protocol and provide intuition for algorithmic and architectural choices. We then proceed to show results comparing software and hardware coherence and discuss the shortcomings of software cache coherence protocols. We propose algorithmic and architectural enhancements that can help improve performance under the software protocols and conclude that software coherence can be competitive with hardware for a large variety of programs.

2 A Scalable Software Cache Coherence Protocol

In this section we outline a scalable algorithm for software cache coherence. Like most behavior-driven (as opposed to predictive compiler-based) software coherence protocols, our algorithm relies on address translation hardware, and therefore uses pages as its unit of coherence. The algorithm has several similarities to directory-based hardware cache coherence but some important differences as well. We make use of a directory (*coherent map*) data structure that maintains caching information for all processors and pages. The coherent map is physically distributed and information for a page is stored on the node where the page resides. Caching for the coherent map itself is disabled.

Each page can be in one of the following four states:

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

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

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

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

When a processor takes an access fault that would cause a page to become `weak` it sends *write*

notices for this event to all sharing processors for that page. Here the similarities with directory-based hardware coherence end. Write notices are not processed when they are received. They are stored on a per processor list of notices (protected by a lock). When a processor performs an *acquire* operation on a synchronization variable, it scans its local list and self-invalidates all pages for which it has received notices. Due to the lack of synchrony between computation and coherence management there is potential for the sending of notices to be a significant percentage of a program's execution time.

In order to eliminate the cost of sending write notices we take advantage of the fact that page behavior is likely to remain constant for the execution of the program, or at least a large portion of it. We introduce an additional pair of states, called **safe** and **unsafe**. These new states, which are orthogonal to the others (for a total of eight disjoint states), reflect the past behavior of the page. A page that has made the transition to `weak` repeatedly and is about to be marked `weak` again is also marked as `unsafe`. Unsafe pages making the transition to the `weak` state do not require the sending of write notices. Instead the processor that causes the transition to the `weak` state records the change in the coherent map entry only, and continues. An acquiring processor must now check the coherent map entry for all its `unsafe` pages and invalidate the ones that are also `weak`. A processor knows which of its pages are `unsafe` by maintaining a local list of such pages when it first maps them for access. Full details for the protocol can be found in a technical report [8].

We apply one additional optimization to the basic protocol. When a processor takes a page fault on write to a shared page (and it is not the only processor that has a read-mapping to that page) we can force the transition and post the write notices immediately or choose to wait until the subsequent *release* operation for that processor; the semantics of release consistency do not requires us to make writes visible before the next release operation. Delayed write notices have been introduced in the context of Munin [4] but the benefits of their usage is not obvious in our environment. Waiting for the subsequent release has the potential to slow things down by lengthening the critical path of the computation (especially for barriers, in which many processors may want to post write notices for the same page at roughly the same time, and will therefore serialize on the lock of the coherent map entry). We have found, however, that delayed transitions are generally a win. They reduce the number of invalidations needed in acquire operations, especially for applications with false sharing.

3 Performance Results

3.1 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 [16], which simulates the execution of the processors, and a back end that simulates the memory system. The front end calls the back end on every data reference (instruction fetches are assumed to always be cache hits). The back end decides which processors block waiting for memory and which continue execution.

We have implemented back ends that simulate memory systems for both a hardware and a software coherent multiprocessor. Both modules are quite detailed, with finite-size caches, full protocol emulation, distance-dependent network delays, and memory access costs (including memory contention). The back end for software coherence includes a detailed simulation of TLB behavior since it is the protection mechanism used for coherence and can be crucial to performance. To avoid the complexities of instruction-level simulation of interrupt handlers, we assume a constant overhead for page faults. Table 1 summarizes the default parameters used in both our hardware

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

Table 1: Default values for system parameters

and software coherence simulations.

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

We report results for six parallel programs. Three are best described as computational kernels: `Gauss`, `sor`, and `fft`. The remaining are complete applications: `mp3d`, `water` [15], and `appbt` [2]. Due to simulation constraints the input data sizes for all programs are smaller than what would be run on a real machine, a fact that may cause us to see unnaturally high degrees of sharing. Since we still observe reasonable scalability for all our applications we believe that this is not too much of a problem.

3.2 Results

Figures 1 and 2 compare the performance of our scalable software protocol to that of an eager relaxed-consistency DASH-like hardware protocol on 16 and 64 processors respectively. The unit line in the graphs represents the performance of a sequentially-consistent hardware coherence protocol. In all cases the performance of the software protocol is within 45% of the performance of the hardware protocol. In most cases it is much closer. For `fft`, the software protocol is actually faster.

For all programs the best software protocol is our protocol described in section 2 with the delayed write notice option. In some cases we have made minor changes to the applications to improve their performance under software coherence. None of the changes required more than half a day to identify and implement. The changes help improve the performance of the hardware protocols as well but to a lesser extent. We believe that the results for `mp3d` could be further improved, by restructuring access to the space cell data structure. It is surprising to see that the relaxed

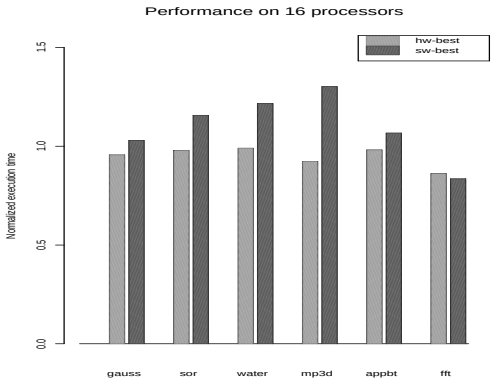

Figure 1: Comparative SW and HW performance on 16 processors

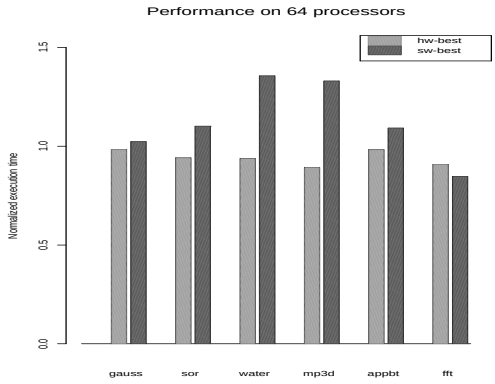

Figure 2: Comparative SW and HW performance on 64 processors

consistency hardware protocol is only marginally better than the sequentially consistent one (the best case is `mp3d`, where the difference is 11%) but the reader should keep in mind that the applications have been restructured to improve locality of reference.

4 Discussion

There are several parameters, both algorithmic and architectural, that affect the performance of software cache coherence. While we have been able to demonstrate performance competitive to hardware cache coherence for a variety of applications, this was not done effortlessly. In this section we summarize the most important issues that designers and programmers should keep in mind in order to achieve good performance on such systems.

We have distributed our directory data structure (the coherent map), essentially eliminating it as a source of memory and interconnect contention. Furthermore taking advantage of the fact that coherence blocks exhibit similar behavior for the duration of a program, we have used the `safe`/`unsafe` distinction to reduce the amount of interprocessor communication required to signal the transition of a coherence block to an inconsistent state. We have seen reductions in running time of well over 50% for our protocol when compared with similar software coherence protocols that do not perform these optimizations [11]. A more detailed comparison of software coherence protocols can be found in [8].

We have found the choice of cache architecture to also have a significant impact on the performance of software cache coherence. Write-back caches provide the best performance for almost all cases, however they require additional hardware for correctness. Our protocol writes modified data back to memory on a release operation. Since we may have multiple writers for a cache line (due to false sharing), we need a mechanism that allows us to successfully merge the modified lines. Per-word dirty bits for write-back caches suffice to perform this function with only 3% cache space overhead. The other alternatives include write-through caches, which keep main memory consistent all the time but may cause high memory and interconnect traffic, and write-through caches with a write-collect buffer [6]. The latter provide performance competitive to that of write-back caches with reduced hardware cost.

Performance is also highly dependent on the sharing behavior exhibited by the program. Due to the higher coherence overhead and the inability to overlap coherence management and computation, software coherence is more sensitive to fine grain sharing (both temporal and spatial) than hardware coherence. Furthermore the large coherence blocks used (pages) may introduce additional sharing when compared to the amount of sharing seen by a hardware coherent system with smaller coherence blocks. We have found that simple restructuring of applications to take into account the larger coherence blocks can help improve performance by well over 50% in many cases. Such program optimizations improve performance on hardware systems as well but to a lesser extent.

One of the most important problems with software coherent systems is the large granularity of coherence blocks. Use of smaller pages can help alleviate this problem but may introduce higher TLB overheads. However, as long as there is no TLB thrashing (i.e. the TLB entries can hold the working set of the application) this does not pose significant problems. Additional methods that can help designers use smaller coherence blocks include subpage valid bits, and injection of consistency code before write instructions in the executable [13]. The latter can provide very fine grained control at the expense of higher software overhead, since consistency code has to run for all memory reference instructions as opposed to the ones that violate consistency. Apart from the choice of coherence granularity, performance of software coherent systems is also dependent on several architectural constants including: cache line size, interconnect and memory latency and bandwidth, cache management instruction costs, interrupt handling costs, and tlb management costs.

Another important deficiency of a software coherent system when compared to hardware is the lack of overlap between coherence management and computation. It may be possible to hide some of the cost of coherence management by performing coherence operations while waiting to acquire a lock. While the acquiring processor will have to run the protocol again when it does acquire the lock, it may not take as long: pages that have already been invalidated need not be considered again. A protocol co-processor could also help reduce coherence management overhead, but in this case the classification of the system as hardware or software coherent is no longer clear. Several recent research projects [9, 12] are taking this intermediate approach, combining the flexibility of software systems with the speed and asynchrony of hardware implementations.

We see the choice between static and dynamic coherence management as an important dimension that affects large scale multiprocessor performance. Compiler inserted coherence directives are the most efficient mechanism for coherence maintenance when the sharing pattern can be determined (or guessed) ahead of time. Run-time coherence management can ensure correctness, allowing aggressive compiler implementations. Unfortunately there is no agreement on the interface between the static and dynamic coherence management layers. We are in the process of designing annotations that will ease the co-operation between compiler and operating system or hardware coherence layers. Under our scheme coherence could be under compiler or runtime control, annotations could have a correctness impact or be purely performance oriented, or in the worst case the absence of annotations could have a correctness impact on program execution. The *entry consistency* of the Midway system [3] employs annotations that fall in this category.

The final dimension that we view as important in the domain of software coherence is the choice of coherence mechanism. Write-invalidate has dominated in both software- and hardware-coherent systems. Recent studies however seem to indicate that write-update may provide better performance for certain sharing patterns and machine architectures. Furthermore for data structures with very fine grain sharing it may be desirable to completely disable caching in order to eliminate coherence overhead. The flexibility of software coherence provides designers with the ability to incorporate all mechanisms in a single protocol and choose the one that best fits the sharing pattern at hand.

5 Related work

Our work is most closely related to that of Petersen and Li [11]: we both use the notion of weak pages, and purge caches on acquire operations. The difference is scalability: we distribute the coherent map, distinguish between safe and unsafe pages, check the weak bits in 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 [4] and lazy release consistency [7] in its use of delayed write notices, but we take advantage of the globally accessible physical address space for cache fills and for access to the coherent map and the local weak lists.

Our use of remote reference to reduce coherence management overhead can also be found in work on NUMA memory management [5]. 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 to determine when to use remote reference.

On the hardware side our work bears resemblance to the Stanford Dash project [10] in the use of a relaxed consistency model, and to the Georgia Tech Beehive project [14] 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 overlap of coherence management and computation (possibly at the expense of extra coherence traffic) in order to avoid a higher waiting penalty at synchronization operations.

6 Conclusions

In this paper we have shown that a shared memory programming model can be supported efficiently without expensive hardware. We have demonstrated good performance on a variety of parallel programs, with coherence maintained in software. We have discussed the factors that limit performance for software coherent systems, and have proposed several optimizations and extensions to our software coherence scheme to address these limitations. We believe that the increased flexibility of software coherence can provide the custom protocols that each application may need, yielding significant performance advantages. Furthermore the higher overhead of coherence management in software becomes less significant as technology progresses. The main performance-limiting factor is memory latency; thus miss rate, and not coherence overhead, will dictate the performance of parallel programs in the near future. Lazy release consistency protocols can help keep miss rates low. Their complexity however makes hardware implementations difficult, leaving software and hybrid implementations as the better alternatives.

Acknowledgements

We want to thank Ricardo Bianchini and Jack Veenstra for their help and support with this work. 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).

References

[1] Anant 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] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Report RNR-91-002, NASA Ames Research Center, January 1991.

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

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

[5] 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.

[6] N. Jouppi. Cache Write Policies and Performance. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[7] 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.

[8] Leonidas I. Kontothanassis and Michael L. Scott. Software Cache Coherence for Large Scale Multiprocessors. TR 513, Computer Science Department, University of Rochester, March 1994. Submitted for publication.

[9] 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.

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

[11] 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.

[12] Steven K. Reinhardt, James R. Larus, and David 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.

[13] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994 (to appear).

[14] 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.

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

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