Computer Architecture News, Sept. 1995, pp. 29-35

Efficient Shared Memory with Minimal Hardware Support *

Leonidas I. Kontothanassis and Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{kthanasi,scott}@cs.rochester.edu
http://www.cs.rochester.edu/u/kthanasi/cashmere.html

July 1995

Abstract

Shared memory is widely regarded as a more intuitive model than message passing for the development of parallel programs. A shared memory model can be provided by hardware, software, or some combination of both. One of the most important problems to be solved in shared memory environments is that of cache coherence. Experience indicates, unsurprisingly, that hardware-coherent multiprocessors greatly outperform distributed shared-memory (DSM) emulations on message-passing hardware. Intermediate options, however, have received considerably less attention. We argue in this position paper that one such option—a multiprocessor or network that provides a global physical address space in which processors can make non-coherent accesses to remote memory without trapping into the kernel or interrupting remote processors—can provide most of the performance of hardware cache coherence at little more monetary or design cost than traditional DSM systems. To support this claim we have developed the *Cashmere* family of software coherence protocols for NCC-NUMA (Non-Cache-Coherent, Non-Uniform-Memory Access) systems, and have used execution-driven simulation to compare the performance of these protocols to that of full hardware coherence and distributed shared memory emulation. We have found that for a large class of applications the performance of NCC-NUMA multiprocessors rivals that of fully hardware-coherent designs, and significantly surpasses the performance realized on more traditional DSM systems.

1 Introduction

As a means of expressing parallel algorithms, shared memory programming models are widely believed to be easier to use than message-passing models. Small-scale, bus-based shared-memory multiprocessors are now ubiquitous, and several large-scale cache-coherent multiprocessors have been designed in recent years. Unfortunately, large machines have been substantially more difficult to build than small ones, because snooping does not scale. Moreover there is a growing consensus that very large machines will be built by connecting a number of small- to medium-scale multiprocessors. Many researchers have therefore undertaken to harness the parallel processing potential of networks of smaller machines.

Unfortunately, the current state of the art in software coherence for networks and multicomputers provides acceptable performance on only a limited class of applications. To make software coherence efficient, one would need to overcome several fundamental problems with existing distributed shared memory (DSM) emulations [2, 7, 14, 23]:

*This work is supported in part by NSF Infrastructure grant no. CDA–94-01142 and ONR research grant no. N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

- Because they are based on messages, DSM systems must trap into the kernel and then interrupt the execution of remote processors in order to perform such time-critical inter-processor operations as directory maintenance and synchronization. Synchronization is a particularly serious problem: the time to acquire a mutual exclusion lock can be three orders of magnitude larger on a DSM system than it is on a CC-NUMA (hardware cache coherent) machine.

- Because messages are so expensive, DSM systems must warp their behavior to avoid them whenever possible. They tend to use centralized barriers (rather than more scalable alternatives) in order to collect and re-distribute coherence information with a minimal number of messages. They also tend to copy entire pages from one processor to another, not only to take advantage of VM support, but also to amortize message-passing overhead over as large a data transfer as possible. This of course works well only for programs whose sharing is coarse-grained. A few systems maintain coherence at a finer grain using in-line checks on references [4, 19, 23], but this appears to require a restricted programming model or very high message traffic.

- In order to mitigate the effects of false sharing in page-size blocks, the fastest virtual-memory-based DSM systems permit multiple copies of a page to be writable simultaneously. The resulting inconsistencies force these systems to compute diffs with older versions of a page in order to merge the changes made by different processors [2, 7]. Copying and diffing pages is expensive not only in terms of time, but also in terms of storage overhead, cache pollution, and the need to garbage-collect old page copies, diffs, and other bookkeeping information.

Hardware cache coherence avoids these problems by allowing inter-node communication without operating system intervention, and without interrupting normal processor execution on the receiving end. This in turn makes it possible to maintain coherence efficiently for cache-line-size blocks, even with only a single concurrent writer.

Our thesis is that most of the benefits of hardware cache coherence can be obtained on large machines simply by providing a global physical address space, with per-processor caches but without hardware cache coherence. Machines in this non-cache-coherent, non-uniform memory access (NCC-NUMA) class include both tightly-coupled (single chassis) multiprocessors (e.g. the BBN TC2000, the Toronto Hector [21], and the Cray Research T3D), and memory-mapped network interfaces for workstations (e.g. the Princeton Shrimp [1], the DEC Memory Channel[5], and the HP Hamlyn [22]). In comparison to hardware-coherent machines, NCC-NUMA systems can more easily be built from commodity parts, and can follow improvements in microprocessors and other hardware technologies closely.

As part of the Cashmere[1] project we have developed a family of protocols for NCC-NUMA systems with various levels of hardware support. As noted in section 2, we are also building a prototype in collaboration with Digital Equipment Corporation.

The Cashmere protocols share several central features. They

- Use ordinary loads and stores to maintain and access directory information for the coherence protocol, avoiding the need for distributed data structures that track the partial order in which events occur on different processors.

- Use uncached remote references to implement fast synchronization operations both for the applications themselves and for the protocol operations that modify directory data structures.

- Allow multiple writers for concurrency, but avoid the need to keep old copies and compute diffs by using ordinary hardware write-through to a unique (and often remote) main-memory copy of each page.

These uses of the global address space constitute performance optimizations unavailable to DSM systems on message-passing hardware.

To maximize concurrency and minimize unnecessary invalidations, the Cashmere protocols employ a variant of invalidation-based lazy release consistency [6]. They postpone notifying other processors that a page has been written until the processor that made the modification(s) reaches a synchronization *release* point. They also postpone

[1] CASHMERe stands for Coherence Algorithms for SHared MEmory aRchitectures and is an ongoing effort to provide an efficient shared memory programming model on modest hardware.

invalidating the page on those other processors until they reach a synchronization *acquire* point. As an additional optimization, some of the protocols dynamically identify pages with high invalidation rates (either due to true or false sharing), and employ an alternative strategy in which releasing processors are not required to multicast write notices, but acquiring processors must either blindly invalidate the page or actively query directory information to see if invalidations are necessary.

The differences among the Cashmere protocols are designed to accommodate different hardware implementations of the global physical address space as well as different interconnect latencies and bandwidths. Specifically, the protocols differ in terms of:

- The mechanism used to implement write-through to remote memory. Some NCC-NUMA hardware will not write through to remote locations on ordinary store instructions. In such cases we can achieve write-through by editing program binaries to include special instruction sequences [11], in a manner reminiscent of the Blizzard [19] and Midway [23] projects.
- The existence of a write-merge buffer. If neither the processor nor the cache controller includes a write buffer capable of merging writes to a common cache line (with per-word dirty bits for merging into memory), then the message traffic due to write-through increases substantially. In such cases performance tends to be better with a write-back policy.
- The mechanism for cache-miss detection. Some NCC-NUMA hardware—including the Memory Channel—will not fetch cache lines from remote memory on a miss. This shortcoming can be addressed by copying pages to local memory in response to a post-invalidation page fault, and then fetching from local memory. Alternatively, on some machines, one can use an idea developed for the Wisconsin Wind Tunnel [17] to generate ECC protection faults on each cache mess, in which case the handler can use a special instruction sequence to fetch from remote memory explicitly.
- The granularity of data transfer. Even on a machine that supports cache fills from remote memory, one has the option of creating a full-page copy in local memory instead. We have found that neither remote fills (cache-line-size transfers) nor full-page copies is superior in all cases. We are currently investigating hybrids; we believe we can choose the strategy that works best on a page-by-page basis, dynamically.

Our work borrows ideas from several other systems, including Munin [2], TreadMarks/ParaNet [7], Platinum [3], and the thesis work of Karin Petersen [15, 16]. It is also related to ongoing work on the Wind Tunnel [19] and the Princeton Shrimp [1] project and, less directly, to several other DSM and multiprocessor projects. Full protocol details and comparisons to related work can be found in other papers [8, 10, 11].

2 Results and Project Status

We have evaluated the performance of NCC-NUMA hardware running Cashmere protocols using execution-driven simulation on a variety of applications. The applications include two programs from the SPLASH suite [20] (`mp3d` and `water`), two from the NASA parallel benchmarks suite (`appbt` and `mgrid`), one from the Berkeley Split-C group (`em3d`), and three locally-written kernels (`Gauss`, `sor`, and `fft`). The results appear in two figures, one comparing NCC-NUMA and CC-NUMA and the other comparing NCC-NUMA and DSM systems. The technological constants (latency, bandwidth, etc.) in the first comparison are characteristic of tightly-coupled machines such as the Cray T3D; the constants in the second comparison are characteristic of more loosely-coupled networks.

Figure 1 compares Cashmere performance to that of the Stanford Dash protocol [13] (modified for single-processor nodes). We have assumed identical architectural constants for both platforms; the only difference is the mechanism used to maintain coherence. The results indicate that software coherence on NCC-NUMA systems can provide performance that approaches or even exceeds that of cache-coherent hardware. For some applications the write-through strategy used by the software protocols proves superior to the write-back strategy used by the hardware protocols and results in a small performance advantage for the software approach.

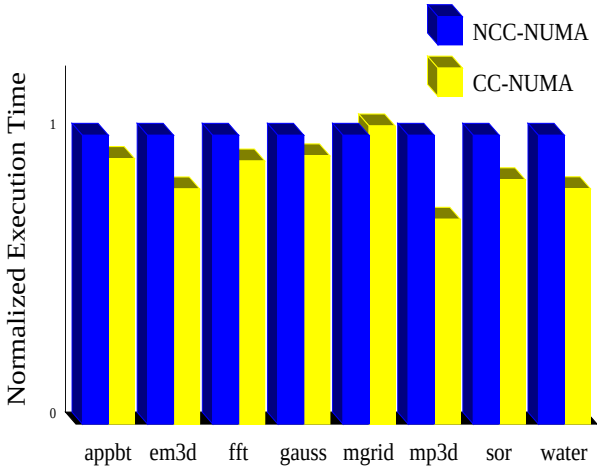

Figure 1: Normalized running time on CC-NUMA and NCC-NUMA hardware

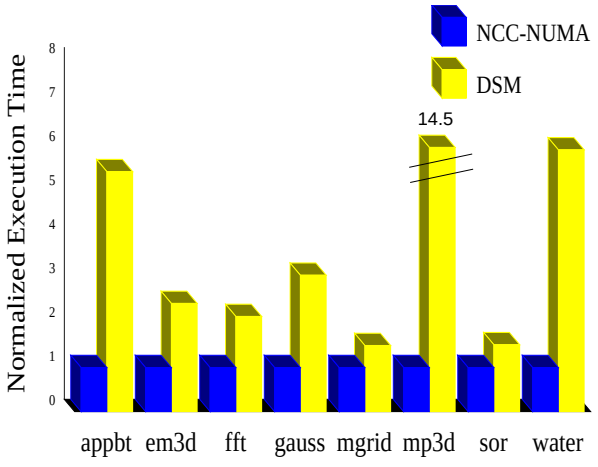

Figure 2: Normalized running time on DSM and NCC-NUMA hardware

Figure 2 compares Cashmere performance to that of the TreadMarks DSM system [7]—the other end of the hardware-software spectrum. Once again we have assumed identical architectural constants for both systems; the only difference is that Cashmere exploits the ability to access remote memory directly. The results indicate that one can achieve substantial performance improvements over more traditional DSM systems by exploiting the additional hardware capabilities of NCC-NUMA systems. A complete discussion and explanation of the results can be found in previous papers [8, 10, 11].

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 [12, 18]. Our studies indicate that "modestly-lazy" multi-writer protocols for such machines will outperform eager single-writer protocols by 5 to 20 percent [9]. It is not yet clear whether the performance advantages of programmable controllers will justify their design time and cost. Our suspicion, based on current results, is that NCC-NUMA systems will remain more cost effective in the near to medium term. More specifically, we speculate that NCC-NUMA systems lie near the knee of the price-performance curve for large-scale multiprocessors (see figure 3): by building network interfaces that turn a network of workstations into an NCC-NUMA machine, we can realize a major increase in performance for a modest increase in cost. At present, our argument is based on comparisons drawn from simulations of the design points. To augment these with practical experience, we have begun to build an NCC-NUMA prototype using DEC's Memory Channel network and 4-processor AlphaServer (2100 4/200) nodes.

Figure 4 depicts the hardware configuration of the Cashmere prototype. The Memory Channel adaptor plugs into any machine with a PCI bus. It permits each node on the network to map local I/O addresses onto a global Memory Channel address space, and likewise to map Memory Channel addresses to local physical memory. Each node retains complete control over which parts of its memory are remotely accessible. Establishing mappings requires a kernel operation, but once created these mappings allow low-latency ($< 3\mu sec$) writes to remote memory from user space. Loads and synchronization operations are similarly fast. Bandwidth for the first generation of the network is modest: 35–50 Mbytes/sec end-to-end, 100 Mbytes/sec aggregate. Aggregate bandwidth on the second generation network, due in 1996, is expected to be an order of magnitude higher, while latency will be reduced by almost a factor of two.

Building the prototype will allow us to extend our research in several directions:

- Our work has so far addressed only small, data-parallel scientific codes. Simulation constraints have prevented us from considering large data sets and applications with long running times. The development of a working prototype will allow us to validate the simulation results and extend our research to domains that are not amenable to simulation studies. Multiprogramming, non-scientific (e.g. interactive) codes, and multi-media

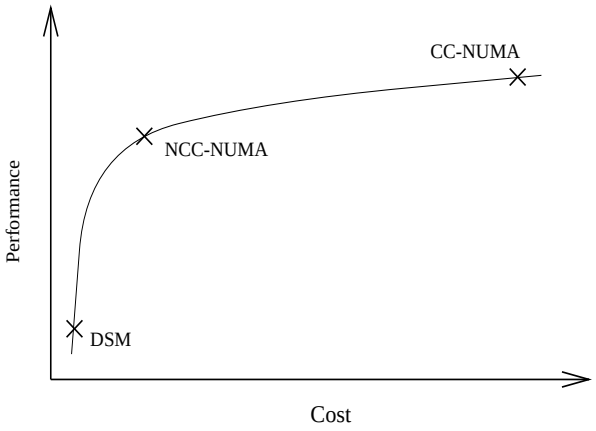

Figure 3: Conceptualization of the price-performance argument for NCC-NUMA systems

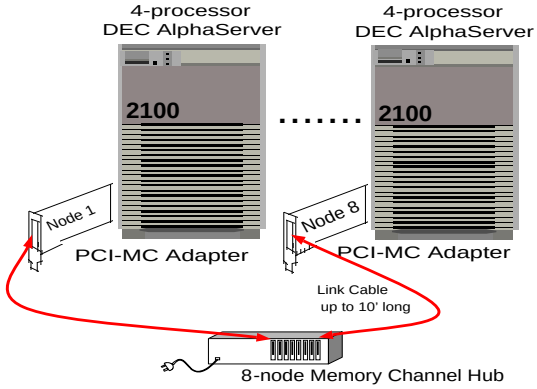

Figure 4: Hardware configuration of the Cashmere prototype

parallel applications are examples of programs that cannot be addressed in simulation and for which the existence of the prototype will prove invaluable.

- The Memory Channel provides some valuable features that we have not so far considered in our work. One is a "clustered" configuration consisting of multiprocessor nodes with internal hardware coherence. Such a configuration seems likely to be more cost-effective than single-processor nodes. The existing Cashmere protocols will run on a clustered system, but it is unclear if they will provide optimal performance. We plan to develop new protocols that cooperate with local hardware cache coherence, rather than competing with it. We will evaluate these protocols first in simulation and then using the prototype hardware. A second attractive feature of the Memory Channel is inexpensive broadcast, which lends itself to update protocols.

- The existence of multiple processors per node presents the opportunity to dedicate one processor to protocol processing and/or data prefetching to local memory, in an attempt to hide protocol processing and data access latency. Whether such a scheme will use the extra processor effectively is still an open question.

We also believe that software coherence can benefit greatly from compiler support. When a compiler is able to completely analyze the sharing patterns in a program, it clearly makes sense to generate code that specifies placement and communication completely, obviating the need for "behavior-driven" coherence at run time. When the compiler is not able to analyze a program completely, however (due to intractable aliasing, input dependencies, etc.), partial information gained at compile time may still be of value in tuning a behavior-driven system. In conjunction with our local compiler group we are pursuing the design of annotations that a compiler can use to provide hints to a software coherence protocol, allowing it to customize its actions to the sharing patterns of individual data structures and/or program phases.

We are also pursuing issues in fault tolerance and heterogeneity. The Memory Channel hardware is designed in such a way that failure of one processor or node does not prevent the continued operation of the remainder of the system. We are designing mechanisms for thread and data management that will allow Cashmere to recover from hardware failures. In addition, since the Memory Channel connects to nodes through an industry-standard PCI bus, we will be able to employ nodes with different types, speeds, and numbers of processors, as well as varying memory and disk capacities.

3 Conclusions

For tightly-coupled systems, our simulations indicate that software coherence on NCC-NUMA hardware is competitive in performance with hardware cache coherence. For networks of workstations, the simulations indicate that the ability

to access remote memory without trapping into the kernel or interrupting remote processors allows software coherence to achieve substantial performance gains over more traditional DSM systems. Based on these results, we believe that NCC-NUMA systems lie at or near the knee of the price-performance curve for shared-memory multiprocessors.

Recent commercial developments suggest that NCC-NUMA networks will be increasingly available in future years. We are currently using one of the first such networks in the construction of a Cashmere prototype. With the higher bandwidths expected in the second-generation, we expect by 1996 to demonstrate low-cost shared-memory supercomputing on a collection of commodity machines.

References

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

[2] 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.

[3] 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.

[4] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy. Integrating Coherency and Recovery in Distributed Systems. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[5] R. Gillett. Memory Channel: An Optimized Cluster Interconnect. *IEEE Micro*, 16(2), February 1996.

[6] 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.

[7] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter '94 Technical Conference*, pages 115–131, San Francisco, CA, January 1994.

[8] L. I. Kontothanassis and M. L. Scott. High Performance Software Coherence for Current and Future Architectures. *Journal of Parallel and Distributed Computing*, 29(2):179–195, November 1995.

[9] L. I. Kontothanassis, M. L. Scott, and R. Bianchini. Lazy Release Consistency for Hardware-Coherent Multiprocessors. In *Proceedings Supercomputing '95*, San Diego, CA, December 1995. Earlier version available as TR 547, Computer Science Department, University of Rochester, December 1994.

[10] L. I. Kontothanassis and M. L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the First International Symposium on High Performance Computer Architecture*, pages 286–295, Raleigh, NC, January 1995.

[11] L. I. Kontothanassis and M. L. Scott. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, San Jose, CA, February 1996. Earlier version available as "Distributed Shared Memory for New Generation Networks," TR 578, Computer Science Department, University of Rochester, March 1995.

[12] 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.

[13] 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.

[14] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, August 1991.

[15] 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.

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

[17] S. K. Reinhardt, B. Falsafi, and D. A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Second Usenix Symposium on Microkernels and Other Kernel Architectures*, San Diego, CA, September 1993.

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

[19] 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. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, San Jose, CA, October 1994.

[20] 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.

[21] Z. G. Vranesic, M. Stumm, D. M. Lewis, and R. White. Hector: A Hierarchically Structured Shared-Memory Multiprocessor. *Computer*, 24(1):72–79, January 1991.

[22] J. Wilkes. Hamlyn — An Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewllett Packard Laboratories, Palo Alto, CA, November 1992.

[23] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for Distributed Shared Memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.