Research Feature

Boosting the Performance of Shared Memory Multiprocessors

Proposed hardware optimizations to CC-NUMA machines—shared memory multiprocessors that use cache consistency protocols—can shorten the time processors lose because of cache misses and invalidations. The authors look at cost-performance trade-offs for each.

Per Stenström
Chalmers University of Technology

Mats Brorsson
Lund University

Fredrik Dahlgren
Chalmers University of Technology

Håkan Grahn
University of Karlskrona-Ronneby

Michel Dubois
University of Southern California

Shared memory multiprocessors make it practical to convert sequential programs to parallel ones in a variety of applications. Most such machines incorporate caches in each node, to allow data replication, and use a cache coherence protocol to ensure that a processor accesses the latest copy of the replicated data.

An emerging class of shared memory multiprocessors—*nonuniform memory access* machines with private caches and a cache coherence (CC) protocol—use a directory-based write-invalidate scheme. They are also *sequentially consistent*, which simplifies program development because it guarantees that the result of every execution is as if the operations from each processor are completed in the order set forth in the program. Among these sequentially consistent CC-NUMA machines are Stanford University's Dash, MIT's Alewife, Convex's Exemplar, NUMA-Q from Sequent Computer Systems, and the most recent Origin 2000 from Silicon Graphics Inc. Although machines with this architecture offer much promise in boosting the performance of parallelized programs, they suffer delays from cache misses (data is not close to the processor) and invalidations (the deletion of stale copies)—both of which cause the processor to stop until the problem can be resolved.

Invalidations and the resulting cache misses typically take tens or hundreds of processor cycles in the Dash and Alewife, even though both use an efficient mechanism to interconnect processing nodes.

In this article we review four proposed optimizations to sequentially consistent CC-NUMA machines. The four differ with respect to which application features they attack, what hardware resources they require, and what constraints they impose on the application software. However, their common goal is to reduce the time lost through cache misses and/or invalidations:

- *Release consistency*.[1] The idea here is to relax the order in which memory operations are performed between two synchronization points. Release consistency models further classify acquire (lock) and release (unlock) requests so that write requests can actually overlap.
- *Adaptive sequential prefetching*.[2] The goal of this cache protocol optimization, which we developed at Lund University, is to enhance the performance of applications that suffer from many misses. The idea is to exploit the merits of bigger cache blocks without introducing the penalties caused by sharing them.
- *Migratory sharing detection*.[3] The idea here is simply to detect when several processors read and modify cache blocks in turn and remove all invalidations. This decreases the number of cycles the processor is stalled.
- *Hybrid update/invalidate with a write cache*.[4,5] In this optimization, the idea is to enhance basic write-update protocols to cut down update traffic, thus making the use of an update-based coherence policy (as opposed to a write-invalidate policy) feasible.

Our review is based on parallel application case studies and detailed architectural simulations conducted at Lund University and the University of Southern California. Our goal was to measure the degree of performance improvement using these four optimizations in isolation and in combination and to look at the trade-offs in hardware and programming complexities. We used a consistent framework and four applications, which we describe later, to pinpoint application characteristics that cause performance problems.

0018-9162/97/$10.00 © 1997 IEEE

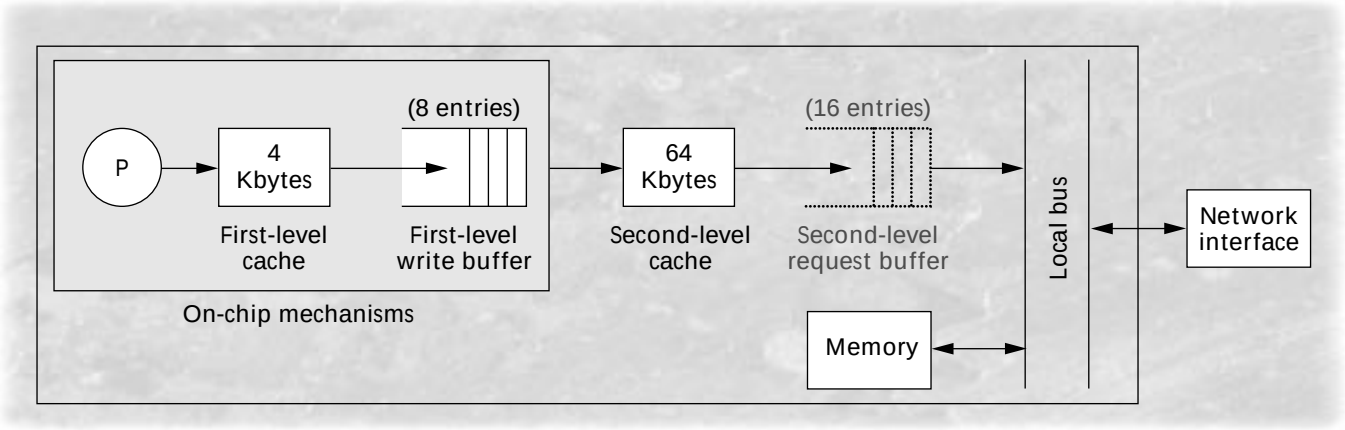

Figure 1. Processing node organization in our generic multiprocessor model.

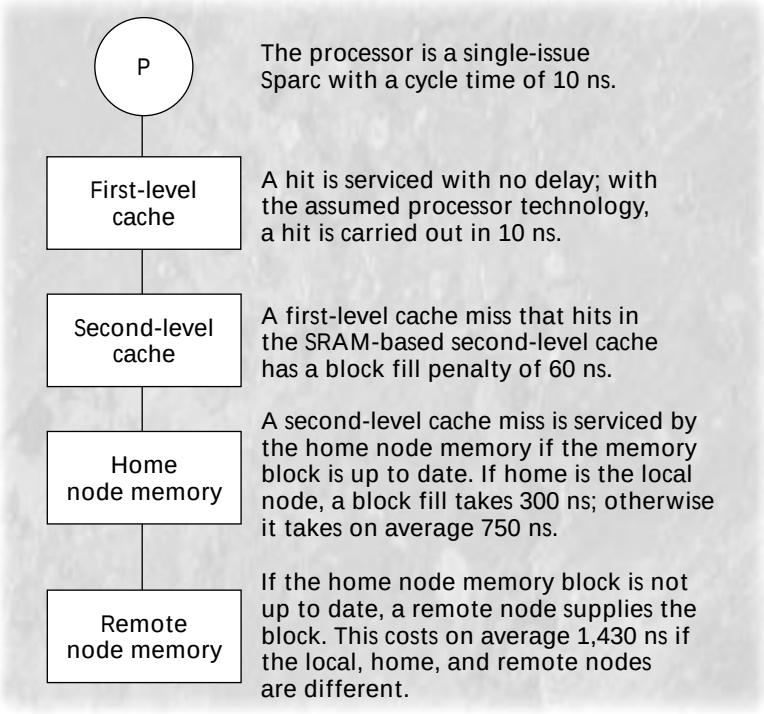

Figure 2. Memory hierarchy and miss penalties in the generic multiprocessor system.

EVALUATION FRAMEWORK

To make sure that we fairly evaluated each optimization and its trade-offs, we used the CacheMire Test Bench[6] to develop a detailed simulation model of a generic CC-NUMA machine. Figure 1 shows the organization of one of its 16 processing nodes.

Memory hierarchy

A central part of the processing node is the two-level cache hierarchy, which consists of a fast on-chip, direct-mapped, write-through first-level cache. This cache is connected via its write buffer to the slower off-chip second-level cache.

The off-chip second-level cache acts as a shield for the long latency of memory operations that are typical of CC-NUMA architectures. Performance centers on the miss penalty of the second-level cache (the number of cycles the processor is stalled on a miss). First, the second-level cache is larger, so first-level cache misses are likely to hit there. Also, the second-level cache implements parts of the directory-based write-invalidate protocol, which enables replication of memory blocks across nodes.

The second-level request buffer (dashed lines) applies only to the proposed optimizations, which we describe later.

Figure 2 shows the four levels of the memory hierarchy of the machine in Figure 1. The first- and second-level caches constitute the first two levels. If the block is not in the second-level cache, the processor sends a miss request to the third level of the memory hierarchy, the *home node memory*, or *home*. Home is the memory with the page that has the missing block and is identified by the least significant bits of the physical page address. When home is the local node, the second-level cache controller fills the first- and second-level caches by accessing the node memory module via the local bus, as Figure 1 shows. Otherwise, it sends a miss request outside the local node through the network interface.

The problem

Home can respond to a second-level cache miss only if the memory copy is up to date. If some other node has modified the block (as indicated by the state of the memory copy), the directory entry identifies the remote node that keeps the only copy and the home memory controller forwards the miss request to that node. Here's the problem: Although the processor can service load operations satisfied by the first two hier-

archical levels within a handful of cycles, it may need as many as two node-to-node transfers to satisfy the third level and four transfers to satisfy the fourth level. This typically takes tens to hundreds of cycles.

Another problem is that a second-level cache block may reside in many caches and writes to such a block must delete stale copies. As with miss request transactions, if no other node has a copy, the cache controller requests an exclusive copy from home directly; if other caches have copies, home must send explicit invalidations to eliminate these copies and wait for acknowledgments before it can notify the local processor to proceed. This is a severe performance bottleneck in sequentially consistent systems.

The four optimizations we describe attempt to address these problems in various ways.

Assumptions

Each node in the generic CC-NUMA machine contains a Sparc processor clocked at 100 MHz. The first-level cache and its write buffer run at the same speed as the processor. The second-level cache and its request buffer are built-in SRAM technology with an access time of 50 ns. The memory is built with DRAMs and has an access time of 90 ns. The first-level cache is 4 Kbytes, the second-level cache is 64 Kbytes, and the line size is 16 bytes. The caches are small—partly to compensate for the small application data sets we used. The first-level write buffer holds eight entries, and the second-level request buffer holds 16.

Our four-by-four mesh interconnect clocks at 100 MHz and transports *flits* (packets) of 64 bits; a message with a single flit travels from one node to another in 53 ns, on average. Figure 2 shows the miss penalties that result from our timing assumptions.

We used the MP3D, Water, LU, and Ocean applications, which were developed by researchers at Stanford University. We picked these programs more to demonstrate interesting performance effects than to form a representative application mix. In this way we hoped to see the spectrum of problems the optimizations could address.

RELEASE CONSISTENCY

Because process coordination in programs that rely on a shared memory model is often based on explicit synchronizations such as lock (acquire), unlock (release), and barriers, correctness is not compromised if the order in which memory operations are performed is relaxed between two synchronization points. Under a *weakly ordered* [7] model, invalidations do not cause access penalties as long as the processors await pending invalidations at synchronization points.

Release consistency [1] is a refinement of the relaxed consistency model, in which synchronizations are further classified into acquire-release pairs. The model assumes the following when two processors issue memory requests to the same variable and at least one of these is a write request:

- The two requests are separated by a release-acquire pair.
- Write requests can overlap as long as they complete before the processor issues a release request.

Hardware/software costs

This technique can hide the latency of an invalidation transaction, which can be long. However, it is not without cost to both software and hardware design.

Software cost results because the memory system must distinguish between synchronization accesses to the shared memory and to ordinary loads and stores, which do not involve synchronizing. Consequently, the program must always use synchronization primitives such as acquire and release. Moreover, there is added work when porting applications from uniprocessor environments that have been run in a concurrent (pseudoparallel) fashion. These applications may use ordinary loads and stores to synchronize because accesses from one process will be seen by others in program order on uniprocessors. The program being ported must have some way to identify ordinary loads and stores before it will run correctly on a multiprocessor that supports a release consistency or other form of relaxed consistency model.

Another cost is machine design. As we said, the memory system must distinguish between ordinary accesses (loads and stores) and synchronization accesses. Moreover, if the processor is able to overlap stores, its designer must include some buffering in the memory hierarchy. This is represented by the last box in Figure 1, the second-level request buffer. This component (which sequentially consistent CC-NUMA machines do not need) buffers each invalidation request along with the data, thus freeing the second-level cache to continue servicing requests from the first-level cache. Besides acting as a first-in, first-out mechanism, the second-level request buffer keeps information about each outstanding request. Miss and invalidation requests are retired from the buffer when their acknowledgments come back from home.

Buffering outstanding write requests lets the latency of pending invalidation requests overlap both each other and local computation as long as the second-level request buffer does not overflow. When the buffer is full, the second-level cache controller blocks the second-level cache at the next invalidation or miss request, which can then back up the first-level write buffer and eventually stall the processor. The second-level cache must also make sure that when the processor executes a release operation (a specially tagged store access), the controller also completes all

We used the CacheMire Test Bench to ensure that we fairly evaluated each optimization and its trade-offs.

previous write requests posted in the second-level request buffer before the controller issues the release request.

Performance trade-offs

We compared the performance of a release consistency machine against that of a sequentially consistent machine using the relevant mechanisms and the timing assumptions in the generic machine model, described earlier. The sequentially consistent machine is essentially the generic machine without the second-level request buffer. The release consistency machine is the complete generic model. Our goal in the comparison was to determine if the performance gains outweigh the costs just described. Figure 3 shows the execution times for the four applications.

To see how release consistency boosts application performance, the figure breaks execution time into components that reflect the cause:

- *Busy*, the fraction of time processors spend doing useful work.
- *Read*, the read stall time, essentially the time the processor spends servicing cache read misses.
- *Write*, the write stall time, in which a writing processor must wait until an exclusive copy is provided. This time does not occur in machines with release consistency.
- *Synch*, the synchronization stall time, the time spent acquiring and releasing locks. As Figure 3 shows, in sequentially consistent systems, cache-miss and invalidation transactions cause significant performance losses across all applications. When release consistency is implemented instead, execution time range decreases from 12 percent (Water) to 39 percent (MP3D). This performance must be traded off against the cost of providing synchronizations to separate accesses to the same variable by different processors. Also, the second-level cache must be able to buffer write requests. Fortunately, this buffer does not have to be very large; we found that 16 entries can keep the processors going.

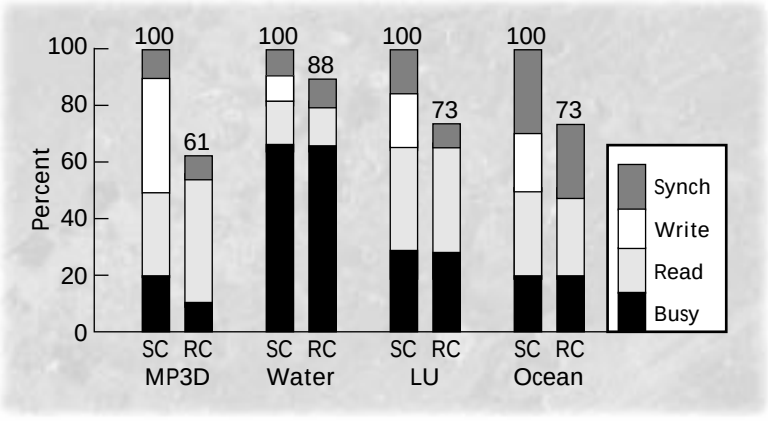

Figure 3. Execution time in relative percentages for each application for a sequentially consistent machine (SC) and a machine with release consistency (RC).

However, release consistency still yields a high miss rate, as Table 1 shows. The reasons stem from the characteristics of the applications.

MP3D. MP3D simulates the movement of 10,000 particles in a wind tunnel for 10 time steps. Each processor moves its particles by reading and modifying the variables that reflect their positions and speed in a space represented by a cell array. Particles may end up in the same cell. This causes several processors to read a cell element and modify it in turn—the *migratory sharing* access pattern, in which blocks that contain cell elements literally migrate from cache to cache, and each migration causes a cache miss followed by an invalidation.

In Table 1, the total miss rate is the miss rate of shared data accesses for each application. Cold misses are caused by a processor's first access to a particular block; replacement misses are caused by the finite size of the cache and/or by an imperfect address mapping. Coherence misses, which are caused by invalidations, dominate the miss rate. This is due to migratory sharing, which also causes devastating write stall time in the sequentially consistent machine (see Figure 3).

Water. Water implements a molecular dynamics simulation of the forces and potentials in a system of 288 water molecules for four time steps. Again coherence misses dominate because of the migratory sharing as different processors access the molecule data structures.

LU. LU performs the lower and upper decomposition of a 200×200 matrix. Each processor is assigned a number of columns and first modifies its columns before making them available to other processors. Thus, most of the cache misses in this application are cold and replacement misses, as Table 1 shows. The miss rate has a dramatic effect on the read stall time (see Figure 3). Because a processor stores consecutive column elements at consecutive memory locations, spatial locality (the probability that a processor accesses adjacent memory locations) is very high.

Ocean. Ocean uses the successive overrelaxation algorithm to iteratively update a grid of 128×128 elements. The basic computation does an update of a grid element by computing a weighted sum of the nearest neighbor elements. Each processor updates all elements in its subgrid in each iteration. Thus, *producer-consumer* sharing could result, in which one processor updates a grid element that another processor is using to update a different grid element. Table 1 shows that

replacement misses and coherence effects are responsible for the overall miss rate, while cold misses are negligible. The high replacement miss rate is common for numerical applications because each processor typically sweeps through a large matrix. Coherence misses are caused by producer-consumer sharing and by false sharing—sharing blocks that happen to cross boundaries between subgrids.[8]

Table 1. Miss rates in the release consistency machine.

Application	Total miss rate	Cold miss rate	Replacement miss rate	Coherence miss rate
MP3D	18.0%	2.4%	2.7%	13.0%
Water	1.9%	0.065%	0.59%	1.2%
LU	2.9%	1.7%	1.1%	0.063%
Ocean	3.1%	0.039%	0.74%	2.3%

SEQUENTIAL PREFETCHING

Sequential prefetching is beneficial to applications (like the four we chose) that have high miss rates and for which spatial locality is high, such as numerical algorithms with a short distance between the vector elements that must be accessed.

As Table 1 shows, each application had replacement and cold misses, many of which were caused by using blocks that are too small. The solution, however, is not just to use larger blocks, because they can generate more coherence misses. These misses are due to false sharing, the effects of which can actually increase the total number of misses. Sequential prefetching exploits the merit of larger cache blocks without affecting false sharing.

The optimization works like this: A mechanism associated with the second-level cache prefetches K nonresident, consecutive memory blocks on each miss. If a processor sequentially accesses consecutive memory blocks, they will be prefetched into the second-level cache on the first miss.

To see how the sequential prefetcher is as effective in exploiting spatial locality as a block K times bigger, but without increasing false sharing misses, consider processors A and B. The two processors alternately read and modify blocks at block addresses n and $n + 1$, respectively. Coherence is maintained at the block level, so block n ends up in processor A's cache, while block $n + 1$ ends up in processor B's. Had block n and $n + 1$ been in the same block, the block would have ping-ponged between A's and B's second-level caches.

Unfortunately, a fixed value of K can create problems. If K is too small, too few blocks are prefetched and the miss rate decreases only marginally. Conversely, if K is too large, blocks that will not be referenced in the future are prefetched, wasting memory bandwidth and increasing contention.

Our sequential prefetcher[2] is adaptive, dynamically adjusting K to reflect the spatial locality in the application. It uses a simple heuristic of prefetch efficiency that counts the fraction of all prefetched blocks that are later referenced. If this fraction exceeds a certain threshold, K is incremented; conversely, if the measured prefetch efficiency is low, K is decremented. The implementation associates only three counters (of four bits each) with each second-level cache. Moreover, this scheme (as are other prefetching schemes[9,10]) is transparent to the software and works equally well for sequentially consistent machines and those with release consistency.

MIGRATORY SHARING DETECTION

Migratory sharing shows up when processes that execute on different processors read and modify a data structure in turn. The migratory sharing pattern can be specified as

$$R_i \ldots W_i \ldots R_j \ldots W_j \ldots R_k \ldots W_k \ldots$$

where R_i and W_i denote a read and a write access from processor i, respectively. Under a write-invalidate protocol, the block ends up in the exclusive state after processor i issues its write (W_i). When processor j reads from the block (R_j), it experiences a coherence miss, followed by an invalidation of processor i's copy at the subsequent write (W_j). Because processor i's cache services both the coherence-miss and the invalidation requests, an obvious optimization is to request an exclusive copy when cache j experiences a coherence miss and to remove the subsequent invalidation. This optimization benefits applications running on sequentially consistent machines, and to some degree applications running on machines with release consistency or other form of relaxed consistency. However, the main benefit in the latter case is reduced memory traffic; if the memory bandwidth is sufficient, the optimization doesn't add much.

The write-invalidate protocol can detect migratory sharing. In our generic machine in Figure 1, this task is greatly simplified because home sees all coherence miss requests as well as all invalidation requests. Migratory sharing detection is invoked when home has seen a sequence like $W_i \ldots R_j \ldots W_j$ in the pattern given earlier. When home sees a write from processor j, it checks if a processor other than j wrote to the block most recently and if there are two copies. Even though this condition is sufficient but not necessary for migratory sharing, this heuristic is simple to implement; we needed a pointer of only $\log_2 P$ bits per memory block to implement it in our generic machine.[3] A similar scheme can be adapted to bus-based multiprocessors.[11]

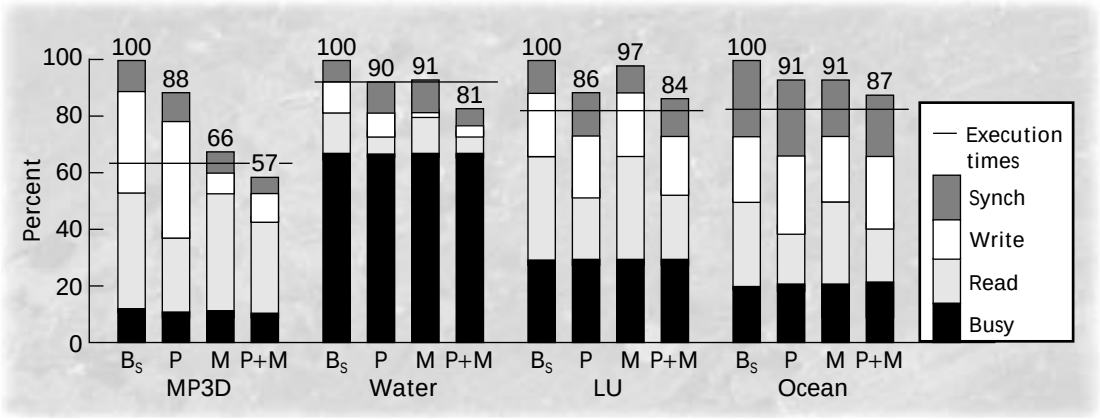

Figure 4. Execution time in relative percentages of four sequentially consistent machines: an unoptimized machine (B_S) a machine with adaptive sequential prefetching (P), a machine with migratory sharing detection (M), and a machine with a combination of these (P+M). The horizontal lines reflect the execution times for a basic machine that supports release consistency.

HYBRID UPDATE-INVALIDATE

Although the sequential prefetcher effectively attacks misses that exhibit a high spatial locality, it does not handle coherence misses very efficiently. A brutal way to remove coherence misses is to adopt an update-based instead of an invalidation-based coherence policy (which we have assumed so far). However, we do not advise taking this approach in its current form. Even if interconnection networks for CC-NUMA machines have a substantial bandwidth, our experience is that memory traffic is an order of magnitude higher for write-update than for write-invalidate protocols. This extra traffic causes severe queuing delays, which make the latency of the remaining cold and replacement misses prohibitively long.

However, if we were to enhance basic write-update protocols to cut down update traffic, adopting an update-based coherence policy might be feasible.

A problem in updating remote copies is that you do not know ahead of time if another processor will read an updated value before the next update to the same address. The result is that some updates are never needed and should be removed because they unnecessarily increase traffic.[8] One solution is to adopt a mechanism that invalidates the local copy when it has received a predefined number of updates with no intervening access by the local processor. Competitive updating is an example of such a protocol. We have taken this approach by associating a counter of $\log_2 n$ bits with each second-level cache line[4] (an approach similar to competitive snooping[12]). We found in studying this protocol that for modest sizes of n (say 4), traffic goes down drastically. However, even then it is still twice as high as it would be with a write-invalidate protocol.

To further cut traffic, we have tried *write caching*,[5] an especially efficient approach. As the name implies, a write cache is a copy-back cache for write accesses only; read accesses do not cause a block to be allocated in a write cache. In a machine with release consistency, write accesses need not propagate to other caches until the next synchronizing access, so there is room to exploit the locality inherent in write accesses between two consecutive synchronizing accesses. This can cut down the traffic caused by updates even more than protocols like the hybrid update/invalidate protocol just described. Our experience shows that a write cache with only four entries is sufficient to exploit virtually all locality in write accesses.

Moreover, it is fairly straightforward to incorporate a write cache because it acts as a small cache that the processor can access in parallel with the second-level cache. As with all the other techniques, it also requires some modifications to the coherence protocol, which we do not address here. However, this optimization works only for a machine that uses release consistency or some other form of relaxed consistency. Consequently, the software must be modified to identify synchronization accesses, as described earlier.

OPTIMIZATION COMPARISONS

Figures 4 and 5 show how effective each optimization was in isolation as well as in combination with others. Figure 4 shows the performance of the basic (no optimizations) sequentially consistent model (B_S), relative to the same machine with different optimizations. Figure 5 shows the performance of the basic (no optimizations) release consistency model (B_R), relative to the same machine with different optimizations.

Sequential consistency

As Figure 4 shows, the proposed optimizations lessen the effects of both read and write stalls. With adaptive sequential prefetching (P), read stall times decrease for all applications. In Water, in particular, P exploits the high spatial locality resulting in a decrease in read stall time of more than half. Conversely, P offers limited opportunities when spatial locality is poor and the coherence miss rate high. In MP3D and Ocean, where coherence misses dominate (see Table 1),

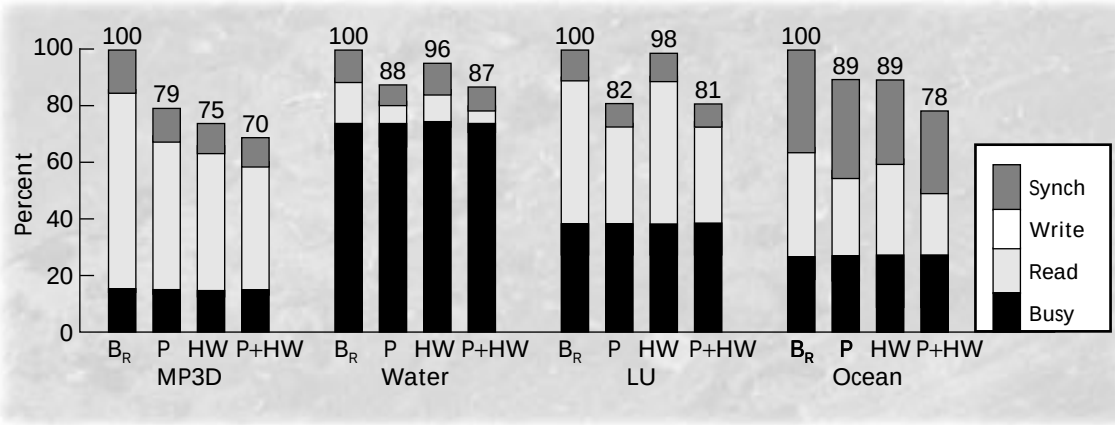

Figure 5. Execution time in relative percentages of four machines with release consistency: a basic machine (B_R), a basic machine with adaptive sequential prefetching (P), a basic machine with hybrid update/invalidate and a write cache (HW), and a basic machine with a combination of these.

P is less effective, indicating that data is exchanged between processors at a fine granularity in applications with poor locality.

With migratory sharing detection, we see dramatic effects on the write stall times for MP3D and Water, the applications in which migratory sharing dominates. In fact, almost the entire write stall time is gone, meaning that these applications run almost as efficiently as on a system with a release consistency model, but without the software complications that such models introduce.

A combination of sequential prefetching and migratory sharing detection (P+M) is particularly effective. P attacks the read stall time, while M effectively cuts write stall time. In Water, P+M virtually removes all read and write stall times. Conceptually, this scheme prefetches an exclusive copy of a block if home has deemed it migratory. Thus, the processor gets rid of the initial miss as well as the subsequent invalidation transaction and has many more cycles for useful work. The horizontal line in Figure 4 shows that P+M on a sequentially consistent machine performs as well as a machine with release consistency. This is significant because it demonstrates that a sequentially consistent system can reach a performance level comparable to a system with release consistency with little or no software changes, which the latter would require.

Release consistency

For machines with release consistency, designers can remove the write stall time by overlapping write accesses with useful computation, so migratory sharing detection is not needed. Instead, we focus on techniques that attack read miss penalties alone. Also, because write accesses can overlap, we also look at update-based protocols—in particular, techniques that buffer writes so that no memory bandwidth is wasted. We consider B_R with sequential prefetching (P) and the hybrid-invalidate with a threshold of one with write caching (HW).

As Figure 5 shows, P removes a significant part of the read stall time for Water and LU. With HW, MP3D and Ocean read stall times also decrease because these applications suffer mainly from coherence misses. P+HW cuts read stall times significantly for all applications, which means the programmer can worry less about how the memory system should access data structures.

Although application designers tend to favor sequentially consistent machines, this model suffers from the latencies of cache misses and invalidations. The optimizations we consider aim at boosting application performance with only a modest increase in machine complexity and minimal constraint on the application software.

Although one combination of the proposed optimizations (prefetching and migratory sharing detection) can boost a sequentially consistent machine to perform as well as a machine with release consistency, release consistency models offer significant performance improvements across a broad application domain at little extra complexity in the machine design. Moreover, a combination of sequential prefetching and hybrid update/invalidate with a write cache cuts the execution time of a sequentially consistent machine by half with fairly modest changes to the second-level cache and the cache protocol. Because of results like these, we believe designers will begin to turn more to the release consistency model.

We also believe that, although we studied these techniques in the context of CC-NUMA architectures, they are applicable to other machine models, such as cache-only memory architectures.[13] These architectures overcome the CC-NUMA machine restriction that data structures be allocated to memories in page chunks. Because COMAs convert memories to huge caches, they can allow data to migrate and replicate in cache-line chunks. This reduces cache-miss penalties, but at the expense of more aggressive memory implementations. ❖

Acknowledgments

This research was supported in part by the Swedish National Board for Industrial and Technical Development (NUTEK) under Contract 9001797 and by the US National Science Foundation under Grant CCR-9115725.

References

1. D. Lenoski et al., "The Stanford Dash Multiprocessor," *Computer*, Mar. 1992, pp. 63-79.
2. F. Dahlgren, M. Dubois, and P. Stenström, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, July 1995, pp. 733-746.
3. P. Stenström, M. Brorsson, and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing," *Proc. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 109-118.
4. H. Grahn, P. Stenström, and M. Dubois, "Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models," *Future Generation Computer Systems*, June 1995, pp. 247-271.
5. F. Dahlgren and P. Stenström, "Using Write Caches to Improve Performance of Cache Coherence Protocols in Shared-Memory Multiprocessors," *J. Parallel and Distributed Computing*, Apr. 1995, pp. 193-210.
6. M. Brorsson et al., "The CacheMire Test Bench—A Flexible and Efficient Approach for Simulation of Multiprocessors," *Proc. Simulation Symp.*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 41-49.
7. M. Dubois and C. Scheurich, "Memory-Access Dependencies in Shared-Memory Multiprocessors," *IEEE Trans. Software Eng.*, June 1990, pp. 660-673.
8. M. Dubois, J. Skeppstedt, and P. Stenström, "Essential Misses and Data Traffic in Coherence Protocols," *J. Parallel and Distributed Computing*, Sept. 1995, pp. 108-125.
9. T.F. Chen and J.L. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes," *Proc. Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 223-233.
10. T. Mowry, M. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. Conf. Architectural Support for Programming Languages and Operating Systems*, ACM, New York, 1992, pp. 62-75.
11. A. Cox and R. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proc. Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 98-108.
12. C. Anderson and A. Karlin, "Two Adaptive Cache Coherency Protocols," *Proc. Symp. High-Performance Computer Architecture*, IEEE CS Press, Los Alamitos, 1996, pp. 303-313.
13. E. Hagersten, A. Landin, and S. Haridi, "DDM: A Cache-Only Memory Architecture," *Computer*, Sept. 1992, pp. 44-54.

Per Stenström is a professor of computer engineering at Chalmers University of Technology. His research interests are in computer architecture with emphasis on multiprocessor design and performance analysis as well as compiler optimization techniques. He has published more than 40 papers and two texts on computer architecture and organization and is on the editorial board of Journal of Parallel and Distributed Computing. *Stenström received an MSEE and a PhD in computer engineering from Lund University. He is a member of the IEEE, IEEE Computer Society, ACM, and SIGArch.*

Mats Brorsson is an assistant professor in information technology at Lund University. His research interests are in multiprocessor systems, particularly workload modeling and characterization and tools for performance debugging. Brorsson received a PhD in computer engineering from Lund University. He is a member of the IEEE Computer Society and ACM.

Fredrik Dahlgren is a research professor in computer engineering at Chalmers University of Technology. His research interests include computer architecture, memory systems for shared memory multiprocessors, and performance evaluation techniques. Dahlgren received an MS in computer science and engineering and a PhD in computer engineering—both from Lund University. He is a member of the IEEE Computer Society.

Håkan Grahn is an assistant professor of computer engineering at the University of Karlskrona-Ronneby, twin cities in Sweden. His main interests are computer architecture, shared memory multiprocessors, cache coherence, and performance evaluation. Grahn received an MSc in computer science and engineering and a PhD in computer engineering—both from Lund University. He is a member of the IEEE Computer Society.

Michel Dubois is an associate professor of electrical engineering at the University of Southern California. His main interests are computer architecture and parallel processing, with a focus on multiprocessor architecture, performance, and algorithms. He also leads the Rapid Prototyping Engine for Multiprocessors, a project to develop a hardware platform for implementing multiprocessor systems with widely different architectures. Dubois received a PhD from Purdue University, an MS from the University of Minnesota, and an engineering degree from the Faculté Polytechnique de Mons in Belgium—all in electrical engineering. He is a member of the ACM and a senior member of the IEEE Computer Society.

Contact Stenström at Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden; pers@ce.chalmers.se; http://www.ce.chalmers.se/~pers.