A Performance Tuning Approach for Shared-Memory Multiprocessors*

Per Stenström and Jonas Skeppstedt

Dept. of Computer Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
email: pers@ce.chalmers.se

Centre for Computer Systems Architecture
Halmstad University
SE-301 18 Halmstad, Sweden
email: jonas@cca.hh.se

Abstract. Performance tuning of applications for shared-memory multiprocessors is to a great extent concerned with removal of performance bottlenecks caused by communication among the processors. To simplify performance tuning, our approach has been to extend the hardware/software interface with powerful memory-control primitives in combination with compiler optimizations to remove communication bottlenecks in distributed shared-memory multiprocessors. Evaluations have shown that this combination can yield quite dramatic application performance improvements. This raises the fundamental question of how the hardware/software interface in future distributed shared-memory machines should be defined to serve as a good target for performance tuning of shared-memory programs, either automatically or by hand. An approach along those lines is discussed.

1 Introduction

Shared-memory multiprocessors have emerged on the commercial arena as powerful platforms for a wide spectrum of high-performance applications. They provide a particularly smooth transition from sequential to parallel processing for two major reasons. One is that they leverage on commodity high-performance microprocessors and memory components as key implementation technologies. The other is that they provide the software system with the illusion of a single memory image which simplifies manual or automatic parallelizations and improves resource utilization at run-time. While it has long been thought that shared-memory multiprocessors cannot scale to large configurations, the commercial emergence of distributed shared-memory (DSM) machines such as Silicon Graphics Origin [7] and Sequent's NUMA-Q [8] has proven the opposite. A critical mechanism in these machines to obtain scalable performance is a hardware cache coherence scheme that allows for programmer transparent replication and migration of data across the compute nodes. In contrast to message-passing

* This research has been sponsored by the Swedish Research Council on Engineering Science (TFR) under contract 94-315.

programming paradigms where communication is explicit in the program, communication takes place implicitly through loads and stores to shared data in a shared-memory parallel program. While this greatly simplifies the design of parallel applications, loads and stores may trigger coherence actions at the hardware level that can have a dramatic impact on performance. It is not uncommon that these reader and writer-initiated inter-node transactions take several hundreds of processor cycles. Techniques to relieve the programmer from dealing with the impact of these long-latency transactions on the application performance are therefore essential.

Our approach [13, 14] has been to let the compiler remove the impact of inter-node coherence transactions by either overlapping them with useful computation or eliminating them altogether. Much like optimizing compilers must have a model of the underlying processor architecture to remove pipeline hazards, a model of the communication architecture in a DSM system must be exposed to the compiler. A key part of our methodology has therefore been to carefully extend the hardware/software interface with primitives that make it possible for the compiler to remove some of the overhead associated with inter-node coherence transactions.

This paper overviews and summarizes our experience in using this approach in a realistic environment. We implemented the hardware primitives in a detailed simulation model of a DSM machine. We then incorporated the compiler algorithms that exploit these primitives in a research compiler and compiled a number of scientific and engineering applications with as well as without the compiler optimizations. We observed quite significant reductions of the execution times—some applications actually ran almost twice as fast. The next section introduces the performance tuning obstacles associated with coherence maintenance in DSM systems. Sections 3 and 4 present our approach and summarize our experiences. This project has focused on the communication that is inherent in shared-memory parallel programs but has not dealt with the artifactual communication caused by the demand-driven nature of managing the complex memory hierarchies in DSM systems. Section 5 concludes by putting our work in this larger context and discussing some prospects for future research.

2 Communication Transactions in DSM Machines

DSM machines are typically built from a number of highly optimized commodity compute nodes with their processors and local memory subsystems as shown in Figure 1. Scalable inter-node bandwidth and latency are achieved by modular high-performance interconnection networks. To efficiently support a shared-memory model on top of this distributed organization, hardware support taking the form of a network interface controller is associated with each node to implement a cache coherence protocol for automatic replication and migration of data across the private caches in each node.

The shared data structure in a shared-memory application is typically allocated statically on a per-page basis across the memory modules in each node.

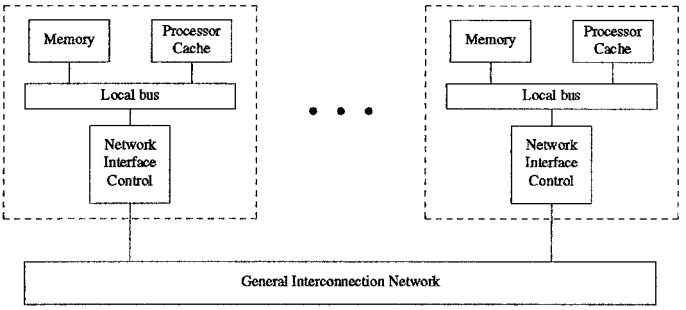

Fig. 1. Organization of a distributed shared-memory machine.

Because the memory access time of remote memory can be 3 to 5 times longer than local memory access time, methods to maximize the number of accesses that are satisfied by either the local cache or the local memory are important. The cache coherence mechanism that allows memory blocks to be replicated across the caches in each node is a key method to obtain this goal.

To understand how the coherence mechanism typically found in such machines impacts on application performance, let us review one favorite method; the one originally proposed by Censier and Feautrier [3]. When a processor references a memory block that is not in its local cache, a cache miss request is directed to the home location of that block. The home node controller keeps track of which nodes have copies of each of its memory blocks and whether the memory copy is up to date using a presence flag vector with the same number of bits as the number of nodes. If the memory copy is up to date, the node controller supplies a copy; otherwise, the request is directed to the node that keeps the up-to-date copy. This node returns its copy to the home node and the home node returns an up-to-date copy to the requesting node. When a processor modifies the block, it must first make sure that no other node can read or write to the memory block. This is achieved by acquiring ownership. An ownership request is sent to the home node that either returns ownership directly if no other copy exists; otherwise, it multicasts invalidation requests to each node with a copy that informs them to remove (and/or give up ownership of) their copies. We note that a read-miss as well as an ownership request can be either a two or a three-party transaction. In recent commercial machines, these transactions can take on the order of a hundred processor cycles. While the latency of ownership transactions can be hidden under relaxed memory consistency models, the processor usually has to stall if the more intuitive sequential consistency model is assumed [6] such as in SQI Origin [7].

3 A Performance Tuning Approach

Our approach to reduce the impact of communication-induced inter-node transactions on the performance of parallel applications has been to augment the hardware/software interface with suitable primitives and then let the compiler

schedule them to overlap the transaction latency with useful computations. In Section 3.1 we introduce the primitives and in Section 3.2 we overview the compiler algorithms that schedule these primitives.

3.1 Memory Control Primitives

To motivate our choice of primitives let us concretely look at one typical situation when read-miss as well as ownership transactions are particularly devastating to application performance. In many parallel applications, mutual access to shared data structures are orchestrated using critical sections. Typically data is read and modified inside these critical sections. If a number of processors subsequently enter the critical section, say P1, P2 and P3, the following communication-induced transactions will occur. After P1 has exited from the critical section, the modified memory blocks are kept in P1's cache. This means that P2 will experience a read miss for each block that P1 modified. After a copy is brought into P2's cache, the store access to that copy will result in an ownership transaction that causes P1's copy to be invalidated. When P3 enters the critical section, the same transactions will of course occur. This particular sharing behavior is called *migratory sharing* [4] because each block involved literally migrates from node to node. We note that the read-miss as well as the ownership transactions are in most cases three-party transactions unless one of the two nodes involved is the home location of the node.

One approach to eliminate the ownership transaction would be to grab an exclusive copy from the previous owner at the time the read miss is taken. We have tried this approach by extending the hardware/software interface with a primitive called **load-exclusive**. This primitive does the same as a load but instructs also the local cache controller to bring an exclusive copy by invalidating other copies in the system.

One approach to shorten the latency of read-miss transactions is to flush back the block to the home location when a processor is done with its modifications. This would have the effect that a three-party transaction is converted to a two-party transaction. It is tempting to be more aggressive and ship the block to the next reader. This optimization would be restricted in the cases when the next reader is not known (see [1] and the references therein). We have considered the former approach and extended the hardware/software interface with a primitive called **write-back**. This primitive instructs the cache to update the home location with the content of the locally cached copy of the corresponding memory block.

In order to make these primitives useful, a compilation framework that analyzes the code and schedule these primitives effectively is needed. In previous work, we have developed compiler algorithms for the load-exclusive and the write-back primitives. Next, we will provide an overview of the approach taken to insert these primitives; for details, consult the original papers [13, 14].

3.2 Compiler Analyses

While interprocess sharing would be best analyzed using a parallel compilation framework that incorporates data dependence analysis, this would restrict its application to regular matrix-oriented computations. As multiprocessors find applicability in a wide range of less regular computations where hand-parallelization is unfortunately state-of-the-art, our approach has been to devise a framework that is useful in this latter context. This means that the application model for the compiler is the code executed by a process. As we shall see, dataflow analysis techniques have been especially effective as a base for scheduling load-exclusive and write-back instructions.

Scheduling load-exclusive instructions The load-exclusive primitive is especially useful when data is first read and then modified meaning that there is a load followed by the store to the same location. The task for the compiler is to detect such load-store sequences that will be unconditionally executed at runtime. We have approached this problem using dataflow analysis in the following way. We assume that a location is referred to using a base pointer b and an offset o. The fact that a load and store instruction refer to the same location can be inferred by using the pair (b, o) as the unit of dataflow. This unit is defined when a store instruction is found in a backward dataflow analysis pass. The dataflow unit reaches a load if and only if the base pointer is not changed and if the store is guaranteed to be executed if the load is executed. If a branch is reached the unit of dataflow will be live in the basic block containing that branch if and only if the dataflow unit is live in both paths corresponding to the branch taken and not taken cases. If the dataflow unit is live at the point the corresponding load is reached, the load will be marked and replaced by a load-exclusive instruction.

Intuition says that data is most often read and modified within the same basic block of code. Following this intuition, the dataflow analysis could be simplified by just considering each basic block in isolation. To test this case, we considered two variations of our compiler algorithm; one that only considers basic blocks in isolation—called **Local**—and one that does dataflow analysis at the intraprocedural level by examining entire flow graphs—called **Conservative**.

There are three sources of limitations to this approach: aliasing, word versus block analysis, and interprocess communication. First, our algorithm does not cover cases where a load uses one base pointer and a subsequent store uses another. Secondly, while it is tempting to extend the algorithm to infer load-store sequences to the same memory block—the granularity at which read-miss and ownership transactions take place—this would require that the compiler is aware of how data structures are aligned with respect to memory block boundaries. The third limitation is associated with our compilation framework that is unable to analyze how loads and stores from different processes interfere. If a load is marked, the data will be brought into the cache in exclusive mode. Now if another processor executes a load to the same address before data is modified, our approach will increase the number of misses. In Section 4, we will comment on how severe these limitations are.

Scheduling write-back instructions Write-back instructions come in two flavors. One is the update instruction which simply informs the cache to propagate the content of the block corresponding to the base pointer b and offset o to the home location. The other is store-update which does a normal store followed by an update. While it is preferable to use the latter instruction to reduce instruction overhead, it is not always possible. The task of the compiler is to detect when a block is modified the last time in a flow graph and insert an update or replacing a store with a store-update there. If data structures were aligned on a memory block boundary, the analysis would have been fairly simple. However, aligning data structures on block boundaries can drastically reduce spatial locality in applications where small records are used [16]. We have therefore augmented our analysis to deal with unaligned data structures although we assume that the memory block size is known at compile time.

Conceptually, our algorithm uses two units of dataflow: the block and the word modified by a store instruction. The approach is to use dataflow analysis applied to these units to infer the earliest point at which all the words contained in a block are modified the last time in a procedure. If the data structures were aligned, the algorithm can infer these points by propagating dataflow information in the forward as well as in the backward direction until a point where the base pointer changes value, or the end of the procedure, which kills the dataflow unit. The fact that the alignment is not known complicates the analysis. By considering two consecutive blocks with respect to the base pointer start address, i.e, b_k and b_{k+1}, it is possible to infer that the memory block containing word $b_{k+1} \times B$, where B is the block size, is modified the last time if the units of dataflow corresponding to blocks b_k and b_{k+1} both are dead at a point where a store to say word $b_{k+1} \times B$ is executed. In this particular case, it is possible to also replace the store instruction with a store-update instruction. This replacement is not in general possible. Consider for example two possible execution paths from a store instruction to address A. If one path contains a store instruction to the same address, that store can sometimes be replaced by a store-update to A. If the other path does not contain a store to the same address, i.e., the corresponding units of dataflow are not live in that path, an update instruction must be scheduled just after the branch.

Like the algorithm that schedules load-exclusive instructions, this algorithm cannot take interprocess communication into account. There are two important implications of this. One issue is that if data is accessed solely by the same process over and over again, shared data will be flushed back to the home node location at every invocation of the procedure. This may cause a substantial increase in memory traffic which as a secondary effect can increase the internode transaction latency of other requests. We have addressed this issue by using a simple heuristic in the cache coherence protocol to detect when a block is accessed by one process only. If the block was not shared when it was brought into the cache, it is considered to be privately accessed. Write-back instructions to such blocks will not cause the block content to be flushed back to the home location. A similar heuristic is actually part of many cache protocols for SMP

systems such as MESI. We will present experimental results with as well as without this optimization. Another issue is how to deal with critical sections. Intuitively, data will most probably be read by other processors after a critical section is exited. Therefore, we have incorporated a static heuristic that considers all units of dataflow associated with blocks and words modified in critical sections as killed at the enter and exit points of these sections.

4 Experimental Evaluation

4.1 Methodological Approach

In order to evaluate the effectiveness at which our approach can remove the performance impact of communication-induced inter-node transactions on shared-memory applications, we did as follows. We developed an accurate simulation model of a distributed shared-memory machine that incorporates the memory control primitives. We then incorporated the compiler algorithms in a research C compiler and compiled a set of scientific/engineering parallel applications with as well as without the optimizations. Finally, we used the simulator to analyze in detail the effectiveness/limitation of our approach.

We used a simulation framework called CacheMire Test Bench that we developed in a previous project [2]. This framework contains a number of SPARC instruction-set simulators that access a single address space using the memory model assumed by the ANL parallel macros. To carry out detailed performance measurements, a model of the memory system architecture (the local node memory subsystem and the interconnection network) was developed. This model preserves timing of memory accesses by delaying each processor for as long as it takes to handle a memory load or a store. The architecture we developed for this project has the overall organization according to Figure 1. It contains 16 nodes and infinite caches in each node to focus on the impact of communication-induced transactions. The block size of each cache is 32 bytes. A read-miss and ownership transaction commits in 30 processor clocks if it is satisfied locally and otherwise 54 processor clocks are charged for each hop in the interconnection network for inter-node transactions. A two-party inter-node transaction therefore takes 30 plus 54 plus 54 adding up to 138 processor cycles.

We have driven our performance analysis using six applications developed at Stanford University. These applications are Water, Cholesky, MP3D, LU, Ocean, and Barnes-Hut. All but LU and Ocean are part of the SPLASH-1 suite [11]. The problem sizes assumed for these applications are: 288 molecules for 4 steps for Water; the bcsstk14 matrix for Cholesky; 10,000 particles for 10 steps for MP3D; a 200 $\times$ 200 matrix for LU; a 128 $\times$ 128 matrix for Ocean; finally Barnes-Hut used a 128-body problem.

A problem with using a research compiler is whether the performance results will be indicative of production quality compilers. The compiler we have used incorporates many of the standard optimizations in e.g. *gcc* and the code effectiveness on the applications used in this study is comparable to what is achieved with *gcc -O2*.

4.2 Effectiveness of Compiler-Inserted Load-Exclusive Instructions

Recall that load-exclusive will be effective for load-store sequences to the same location and where the load results in a cache miss. In such cases, the load can bring an exclusive copy into the cache which eliminates the subsequent ownership transaction. We first measured how many of the loads that result in cache misses that are followed by a store to the same location. We then measured how many of these loads that each of the algorithms could convert to load-exclusive; this is the coverage of the algorithms. The top diagram of Figure 2 shows the coverages of the Local and the Conservative algorithms. The first four applications exhibit high coverages for L as well as C whereas Ocean and Barnes-Hut do not benefit much. Below each application we show the fraction of loads that result in cache misses that are followed by a store to the same location. For the first three applications, which exhibit substantial migratory sharing, more than 80% of the load misses would benefit from load-exclusive so we would expect the algorithms to be effective for these. By contrast, in LU only 10% of the misses would benefit from load-exclusive so the benefits are not expected to be great.

Figure 2 also shows the impact of the load-exclusive algorithms on the execution times of the applications. B stands for the unoptimized case, whereas L and C stand for the Local and Conservative algorithms, resp. Because this optimization can only cut the time spent acquiring ownership, we have subdivided the execution time into four sections. The bottom section is the fraction of useful computation whereas the top three sections are losses due to memory system events. The memory system events are from bottom to top due to synchronization, read misses and ownership acquisitions. As expected, the top section is almost completely wiped out for Water, Cholesky, and MP3D. For Cholesky and MP3D, the execution times are reduced by 31% and 27%, resp. By contrast, because ownership latency does not contribute much in Water, the removal of this component does not make much impact. Interestingly, L and C are equally effective suggesting that local dataflow analysis suffices. We have noticed that neither L nor C increased the number of read misses suggesting that accesses from other processors that intervene a load-exclusive and a subsequent store are rare. Further, the fact that the algorithm uses words rather than memory blocks as the unit of dataflow did not limit the effectiveness much.

4.3 Effectiveness of Compiler-Inserted Write-back Instructions

Let us now consider the effectiveness of compiler-inserted write-back instructions. In Figure 3 the execution time is shown for each application for three cases: the unoptimized case (B), the optimized case on an architecture that does not implement the private heuristic in the cache coherence protocol (S), and the optimized case on an architecture that does (S+D).

We initially did some experiments assuming a sequentially consistent system. These experiments showed that the read-miss transaction latency went down but to the expense of a higher ownership transaction latency. This discouraging result was attributed to a small fraction of updates that were useless and made the

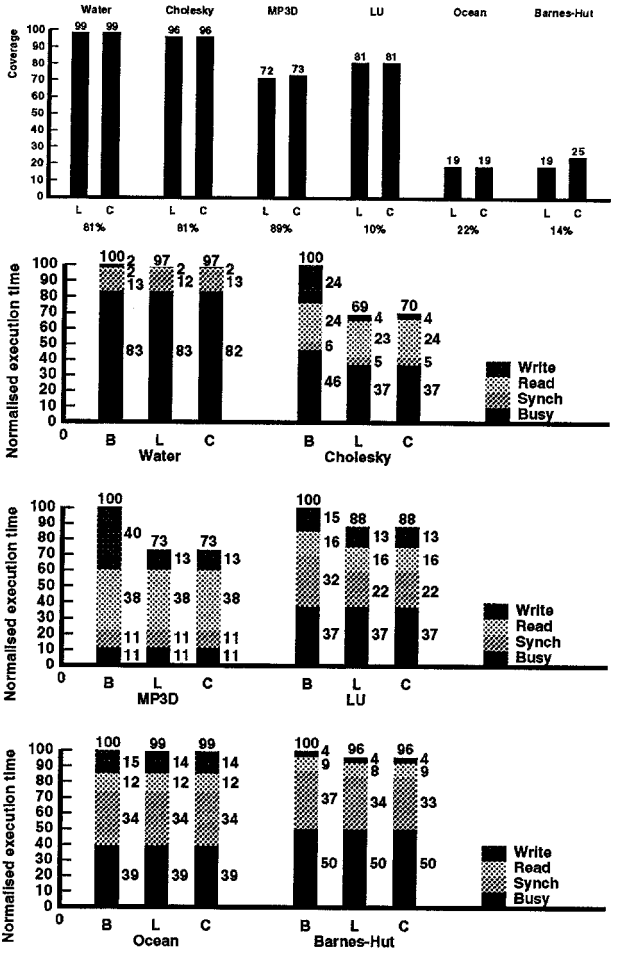

Fig. 2. The top diagram shows the coverages of the compiler algorithms for each application. The next three diagrams show the execution times of the codes optimized with the Local (L) and the Conservative (C) algorithms relative to the unoptimized case (B).

same processor stall when it reaccessed the block. The results we will show here assume a release consistent system in which ownership transaction latency is completely hidden.

The execution times are split up into four sections (from bottom to top): fraction of useful computation, synchronization, read miss penalty to blocks that are up to date in the home location, and read miss penalty to blocks that are not up to date in the home location. It is the last component that store-update can cut by converting it to miss latency to the home location.

Although the impact of the execution time ranges from 1% to 38%, the

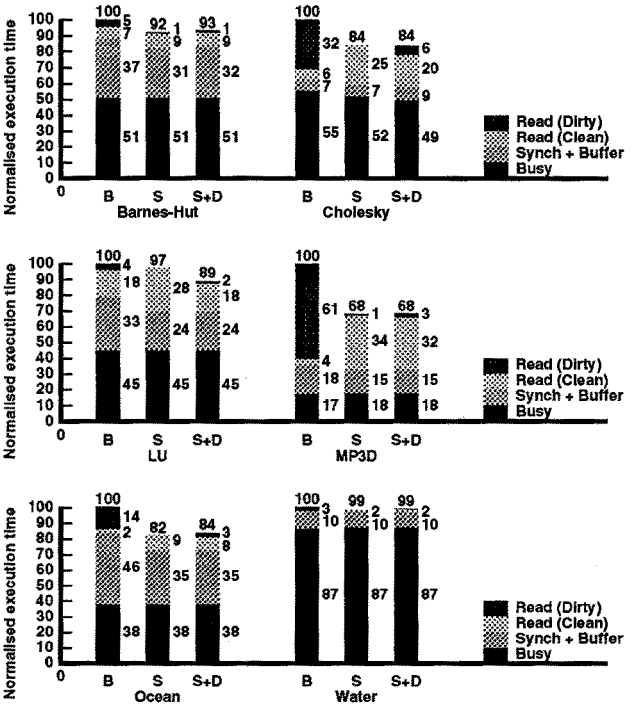

Fig. 3. Execution times of the codes optimized by the store-update algorithms using an architecture that does not implement the private access detection heuristic (S) and one that does (S+D) relative to the unoptimized case (B).

compiler algorithm managed to convert between 83% to 100% of the three-party transactions to two-party transactions. While the execution times in the S and S+D case are almost the same, we noticed that the extra traffic caused by the useless updates to shared data that is only accessed by a single processor was huge in some cases (for LU it was seven times higher). The bandwidth of the interconnection network we assumed could accommodate this extra traffic but in systems with less bandwidth, the heuristic that classifies data as shared or private is important. We also noticed that the compiler algorithms in most cases were able to use store-update instead of update; indeed, the instruction overhead was less than 4% for the studied applications.

4.4 Related Work

Most related to our work are the different flavors of write-back instructions that are discussed and evaluated in [1]. To the best of our knowledge, however, there is only one paper that reports on a compiler framework for automatic insertion of write-back instructions [5]. They consider a parallelizing compiler framework that is applicable to vector accesses with affine loop indexes. By contrast, our

framework is applicable to a more general class of programs with irregular access patterns [12].

Related to write-back instructions to hide read-miss transaction latency is also compiler-controlled prefetching [9]. Abdel-Shafi *et al.* [1] found that write-back instructions have a potential to help prefetching in those cases where it is not as effective. Indeed, our more recent work on prefetching has revealed that many of the limitations of compiler-controlled prefetching can be overcome with our compiler framework for automatic insertion of write-back instructions [12].

5 Concluding Remarks and Future Work

We have shown that quite simple extensions of the hardware/software interface with memory-control primitives in conjunction with standard compiler optimization techniques such as dataflow analysis can be quite effective at automatically tuning the performance of shared-memory parallel programs on DSM machines. While our focus has been on communication-induced overheads, other important overheads are caused by artifactual communication to bring data close to the processors that access it.

Current DSM machines as well as those we are expecting to see in the future use memory hierarchies with several levels. The upper levels are typically inside a node whereas the lower levels are in remote nodes. Given that the node-resident levels of the memory hierarchy have a capacity that matches the working set sizes of the application, the traditional demand-driven management of memory hierarchies works well because the temporal locality can then be exploited. In fact, many scientific applications have quite distinct working sets that grow slowly with the problem size as was shown in [10] and a fairly limited memory space is needed to keep the working set in the node-resident levels of the memory hierarchy. However, the big footprints and the little data reuse in database applications may not exhibit this behavior. In fact, a recent uniprocessor study based on the integer applications in the SPEC suite showed very little data reuse of some data structures [15]. Moreover, coherence actions tend to shorten the cache residency time. Data structures that exhibit this behavior may not benefit as much from caching in the upper levels of the memory hierarchy. Even worse, they may actually interfere with data structures that exhibit a high temporal locality and cause them to be replaced from the node-resident levels of the memory hierarchy. We are currently exploring an approach called *selective caching*. The idea is to control the cacheability of data structures at each level in the memory hierarchy using memory-control primitives. Data structures with working set sizes that match the capacity at a specific level should be cached there and at the lower levels whereas data structures with poor temporal locality should not be cached there. We will investigate how prefetching and other memory-control primitives discussed in this paper may help to hide the latency of the artifactual communication caused by accesses to data structures that cannot be cached at or above a certain level.

References

1. Hazim Abdel-Shafi et al. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the 3rd Int. Symp. on High-Performance Computer Architecture*, pages 204–215, February 1996.
2. Mats Brorsson, Fredrik Dahlgren, Håkan Nilsson, and Per Stenström. The CacheMire Test Bench - a Flexible and Effective Approach for Simulation of Multiprocessors. In *Proceedings of the 26th IEEE Annual Simulation Symposium*, pages 41–49. IEEE, New York, March 1993.
3. Lucien Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Trans. Comput.*, 27(12):1112–1118, 1978.
4. Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocesors. *IEEE Trans. Comput.*, 41(7):794–810, 1992.
5. D. Koufaty, X. Chen, D.K. Poulsen, and J. Torrellas. Data Forwarding in Shared-Memory Multiprocessors. *IEEE Trans. on Parallel and Distr. Systems*, 7(12):1250–1264, 1996.
6. Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Comput.*, C-28(9):690–691, 1979.
7. James Laudon and Daniel Lenoski. The SGI Origin 2000: A CC-NUMA Highly Scalable Server. In *Proceedings of 24th Annual International Symposium on Computer Architecture*, to appear, 1997.
8. Tom Lovett and Russel Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of 23rd Annual International Symposium on Computer Architecture*, pages 308–317, 1996.
9. Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233. ACM, New York, 1996.
10. E. Rothberg, J-P Singh, and A. Gupta. Working Sets, Cache Sizes, and Granularity Issues for Large-Scale Multiprocessors. In *Proc. of the 20th Int. Symp. on Computer Architecture*, pages 14–26, May 1993.
11. Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Comput. Arch. News*, 20(1):5–44, 1992.
12. Jonas Skeppstedt. *Overcoming Limitations of Prefetching in Multiprocessors by Compiler-Initiated Coherence Actions.* Technical report No 274, Dept. of Computer Engineering, Chalmers Univ., December 1996.
13. Jonas Skeppstedt and Per Stenström. A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques 1995*, pages 69–78. IFIP, Laxenburg, Austria, 1995.
14. Jonas Skeppstedt and Per Stenström. Using Dataflow Analysis Techniques to Reduce Ownership Overhead in Cache Coherence Protocols. *ACM Trans. on Programming Languages and Systems*, 18(6):659–682, 1996.
15. T. Johnson and W-M Hwu. Run-Time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proc. of the 24th Int. Symp. on Computer Architecture*, June 1997. to appear.
16. Josep Torrellas, Monica Lam, and John Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Trans. Comput.*, 43(6):651–663, 1994.