Evaluating the Potential of Programmable Multiprocessor Cache Controllers

John B. Carter
Mike Hibler
Ravindra R. Kuramkote

UUCS-94-040

Department of Computer Science
University of Utah

Abstract

The next generation of scalable parallel systems (e.g., machines by KSR, Convex, and others) will have shared memory supported in hardware, unlike most current generation machines (e.g., offerings by Intel, nCube, and Thinking Machines). However, current shared memory architectures are constrained by the fact that their cache controllers are hardwired and inflexible, which limits the range of programs that can achieve scalable performance. This observation has led a number of researchers to propose building *programmable multiprocessor cache controllers* that can implement a variety of caching protocols, support multiple communication paradigms, or accept guidance from software. To evaluate the potential performance benefits of these designs, we have simulated five SPLASH benchmark programs on a virtual multiprocessor that supports five directory-based caching protocols. When we compared the off-line optimal performance of this design, wherein each cache line was maintained using the protocol that required the least communication, with the performance achieved when using a single protocol for all lines, we found that use of the "optimal" protocol reduced consistency traffic by 10-80%, with a mean improvement of 25-35%. Cache miss rates also dropped by up to 25%. Thus, the combination of programmable (or tunable) hardware and software able to exploit this added flexibility, e.g., via user pragmas or compiler analysis, could dramatically improve the performance of future shared memory multiprocessors.

Evaluating the Potential of Programmable Multiprocessor Cache Controllers

John B. Carter
Mike Hibler
Ravindra R. Kuramkote

Department of Computer Science
University of Utah

1 Introduction

There are two basic ways that parallel processes typically communicate: via *message passing* and via *shared memory*. The current generation of massively parallel processors (e.g., machines by Intel [19] and Thinking Machines [28]) support message passing as the sole means of communication because of its architectural simplicity and relative scalability. However, message passing has failed to establish itself as being as straightforward and intuitive a programming model as shared memory. Message passing places all of the burden of data decomposition, synchronization, and data motion on the programmer and compiler. Despite substantial effort, compiler technology has not yet advanced to the stage where it can automatically extract and exploit parallelism (via the insertion of message passing calls) on a wide range of programs, which forces the programmer to do much of the work. Therefore, while message passing is, and will remain, an important communication mechanism, parallel programmers are increasingly demanding support for shared memory.

Traditionally, this demand has been satisfied in two ways: (i) via software implementations of distributed shared memory (DSM) on message-passing hardware [5, 8, 9, 12, 23] or (ii) via hardware implementations of scalable shared memory architectures [6, 11, 22, 32]. Research in both of these areas has been very successful in recent years. The performance of software DSM systems has improved dramatically by addressing the problems of false sharing and excessive DSM-related communication [9, 12]. Efficient DSM systems can now perform as well as hardware shared memory for large grained programs [12], but the overhead associated with software implementations of DSM limits its value for fine-grained computations. Spurred by scalable shared memory architectures developed in academia [11, 21, 32], the next generation of massively parallel systems will support shared memory in hardware (e.g., machines by Convex [2, 3] and KSR [6]). However, current shared memory multiprocessors all support a single, hardwired consistency protocol and do not provide any reasonable hooks with which the compiler or runtime system can guide the hardware's behavior. Using traces of shared memory parallel programs, researchers have found there are a small number of characteristic ways in which shared memory is accessed [4, 15, 17, 29]. These characteristic "patterns" are sufficiently different from one another that any protocol designed to optimize one will not perform particularly well for the others. Since all existing and announced commercial multiprocessors implement a single hardware consistency mechanism, they will perform well only for programs that access memory in the way that the hardware designers expect.

This research was supported in part by the National Science Foundation under the grant CCR-9308879 and the Hewlett-Packard Research Grants Program, and the University of Utah.

These observations have led a number of researchers to propose building *programmable multiprocessor cache controllers* that can execute a variety of caching protocols [7, 31], support multiple communication paradigms [10, 18], or accept guidance from software [20, 25]. Programmable controllers would seem at first glance to be an ideal combination of software's greater flexibility and hardware's greater speed – dedicated hardware could be used to handle the common cases efficiently, while software could be used to handle uncommon cases and/or tune the way in which the hardware handles the common cases. However, this greater power and flexibility increases hardware complexity, size, and cost. To determine if this added complexity and expense is worthwhile, we must determine the extent to which it can improve performance. In this paper, we describe a study that we undertook: (i) to determine the potential value of programmable cache controllers, (ii) if they should prove valuable, to identify particular features that should be included in the design of future programmable controllers, and (iii) to determine how much of an impact softwared can have by tuning the hardware's behavior. We should make it clear that we attempted to derive *general* conclusions about the value of programmable controllers and *not* specific conclusions about the value, or lack thereof, of any particular design that has been proposed. Our intent is to guide designers of these systems and to provide a framework under which to design a scalable shared memory multiprocessor of our own. By using a programmable controller to reduce the amount of communication required to maintain consistency, multiprocessor designer can either use a slower (and thus cheaper) interconnect and achieve performance equal to that of a static controller and a fast interconnect, or use the same fast interconnect, but support more processors.

We simulated the performance of the SPLASH benchmark suite [26] on a directory-based shared memory multiprocessor that supported five different cache consistency protocols. To determine the value of adding flexibility to the node cache controllers, we varied the number of protocols that were supported, the size of cache lines, and the degree to which we assumed that software was able to inform the controller of the "optimal" protocol for each cache line. In addition, we tested a number of variants of the basic protocols. Varying the number of caching protocols and the size of cache lines lets us determine how much hardware complexity is useful, while varying the degree to which we assume that software correctly tunes the controller's behavior lets us determine the importance of higher level software (e.g., user pragmas or compiler analysis). We found that the use of the "optimal" protocol for each cache line reduced consistency traffic by 10-80%, with a mean improvement of 25-35%. Furthermore, for several of the applications, the use of the "optimal" protocol in terms of minimizing consistency traffic also reduced the cache miss rate by up to 25%. These numbers were relatively stable as we varied cache line sizes and the number of processors involved in each computation, although the specific protocol that was "optimal" for a given line was greatly affected by the cache line size (but, in general, not the number of processors).

These findings indicate that the use of flexible, programmable or software-tunable, multiprocessor cache controllers can dramatically improve the performance of shared memory multiprocessors. The key will be determining the appropriate hardware-software interface and being able to determine and specify to the controller the "optimal" consistency protocol for handling pieces of shared data.

The remainder of this paper is organized as follows. In Section 2, we describe the experiments that we ran, with special emphasis on the system model that we used and the assumptions that we made. Section 3 contains the results of our series of experiments, including a detailed look at two specific refinements that we evaluated. Related work is discussed in Section 4, and we draw conclusions and make recommendations for future programmable cache controller designs in Section 5.

2 Experimental Setup

2.1 Overview

We took great effort not to assume any specific hardware implementation of the programmable cache controller or any specific machine architecture so as not to bias our results.

The reference system model that we used, illustrated in Figure 1, is a generalization of the combined features of Alewife [10, 11], FLASH [18, 20], and Typhoon [25]. We assume a distributed shared memory (DSM) model, whereby the main memory in the machine is equally distributed across the processing nodes. Every block of physical memory has associated with it a "home node," which is the node at which the main memory copy of the block resides [21]. A directory-based cache is used to support shared memory spanning all of the nodes in the machine [1, 11, 21]. When a node accesses a word of shared data that is not local and that does not currently reside in the local cache, the local cache controller interacts with the data's home node, and possibly other nodes depending on the particular caching protocol, to acquire a copy of the data and put it in the local cache. The cache controller is responsible for maintaining the consistency of data in the local cache, and for responding to requests from remote cache controllers to perform operations on either the local cache or the local memory. The operations that are performed on local cache and memory depend on the particular consistency protocol being used. Section 2.4 describes the various protocols that we examined, and the operations that each performs to maintain consistency. Associated with each line in the local cache are a number of tag bits, state bits, and protocol bits, as shown in Figure 2. The specific number of these "overhead" bits depends on the number and composition of protocols

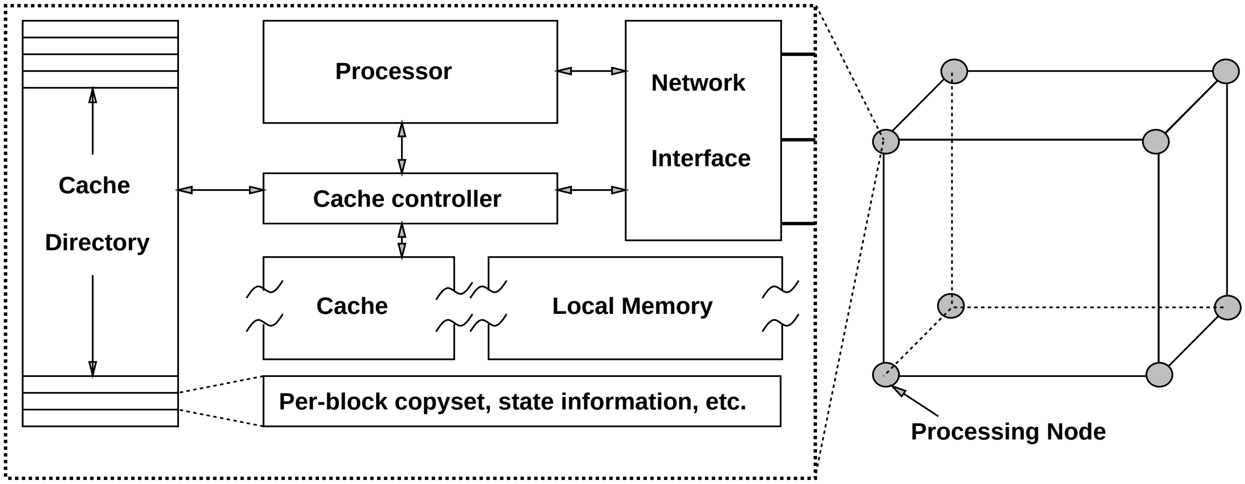

Figure 1 Reference System Model (not to scale)

State Bits	Tag Bits	Protocols	Data
(2-8)	(0-32)	(0-2)	(32-512 bytes)

Figure 2 Layout of Cache Line

supported by the simulated controller. The number of overhead bits varies from 1-4% of the data bits, depending on the size of the cache line and the complexity of the modeled hardware. The cache directory in our model is used in the same way as the directories in DASH [21] and Alewife [11]. The home node for each block of physical memory maintains a directory entry corresponding to the block with information about its global state, such as the set of processors with copies of the block (the "copyset") and the state of the block (read-shared, exclusive, etc.). Finally, although Figure 1 shows a cubic topology, we do not assume any particular internode communication fabric. We merely assume that point-to-point messages can be sent between any two nodes in the system.

In addition to this hardware model, we assume that software is capable of providing a limited amount of direction to the hardware to tune its operation. Specifically, for some of our comparisons we assume that software can specify which protocol to use to maintain consistency on a per-cache-line basis, and can further tune the individual protocols by setting state bits appropriately. For the purposes of this study, we do not specify how this is accomplished, since our intent is only to measure the performance impact that correct tuning can provide. Alewife [10], Typhoon [25], and FLASH [20] all provide mechanisms to allow software access to the caching hardware, to varying degrees.

2.2 MINT multiprocessor simulator

We used the MINT memory hierarchy simulator [30] running on a collection of Silicon Graphics Indigo2's to perform our simulations. MINT simulates a collection of processors and provides support for spinlocks, semaphores, barriers, shared memory, and most Unix system calls. It generates multiple streams of memory reference events, which it uses to drive a user-provided memory system simulator. We chose MINT over several similar multiprocessor simulators, such as Tango [14] and the Wisconsin Wind Tunnel (WWT) [24], for three reasons. First, it runs on fairly conventional hardware, namely MIPS-based workstations from Silicon Graphics and DEC. This separates it from WWT, which requires the use of a prohibitively expensive Thinking Machines CM-5. Second, it is very efficient. While programs run under MINT run between 15 and 70 times slower than they run in native mode, this was more than adequate for the experiments that we ran. Depending on the number of processors and the complexity of the cache controllers being simulated, our simulation runs took between five minutes and two hours to complete. MINT is approximately 100 times faster than Tango [30], which would have required hours or days per simulation run. Finally, MINT has a clean user interface and is relatively easy to program. This let us modify our algorithms repeatedly while refining our experiments, and in conjunction with its efficiency, made it possible for us to test a large number of hypotheses.

2.3 SPLASH benchmark programs

We used five programs from the SPLASH benchmark suite [26] in our study, `mp3d`, `water`, `barnes`, `LocusRoute`, and `cholesky`. We were unable to compile the remaining two programs in our test environment[1]. Table 1 contains the inputs for each test program. `mp3d` is a three-dimensional particle simulator used to simulated rarified hypersonic airflow. Its primary data structure is an array of records, each corresponding to a particular molecule in the system. `mp3d` displays a high degree of fine-grained write sharing. `water` is a molecular dynamics simulator that solves a short range N-body problem to simulate the evolution of a system of water molecules. The primary data structure in water is a large array of records, each representing a single water molecule and a set

[1] We are working to remedy this problem, and will include more complete results in the final version of the paper.

of forces on it. `water` is fairly coarse-grained compared to `mp3d`. `barnes` simulates the evolution of galaxies by solving a hierarchical N-body problem. Its data structures and access granularities are similar to that of `water`, but its program decomposition is quite different. `locus` evaluates standard cell circuit placements by routing them efficiently. The main data structure is a cost array that keeps track of the number of wires running through the routing cell. `locus` is relatively fine-grained, and the granularity deviates by no more than 5% for all problem sizes. Finally, `cholesky` performs a sparse Cholesky matrix factorization. It uses a task queue model of parallelism, which results in very little true sharing of data, although there is a moderate degree of false sharing when the cache lines are fairly large. We detected data races in `mp3d`, `barnes` and `LocusRoute`, but most of the them were localized to few words.

2.4 Protocols Investigated

We evaluated the performance of five basic consistency protocols: (i) a sequentially consistent multiple reader, singler writer, write invalidate protocol ("Conventional"), (ii) a no-replicate migratory protocol ("Migratory"), (iii) a release consistent [16] implementation of a conventional multiple reader, single writer, write invalidate protocol ("Dash"), (iv) a protocol that adapts to the way that it is used to dynamically switch between Dash and Migratory, depending on how a cache line is being used [13, 27] ("Adaptive") and (v) a release consistent multiple reader, multiple writer, write update protocol ("Munin"). We selected these five protocols because they covered a wide spectrum of options available to system designers. Table 2 summarizes the design parameters of the four protocols. The rest of this section describes the protocols that we evaluated, including the message count that was used to simulate various consistency operations.

Program	Input parameters
`mp3d`	5000 particles, 50 time steps, `test.geom`
`water`	`LWI12`, `sample.in`
`cholesky`	`bcsstk14`
`barnes`	`sample.in`
`locus`	`bnrE.grin`

Table 1 Programs and Problem Sizes Used in Experiments

Consistency Protocol	Replicate on reads?	Invalidate or Update	Number of Writers	Consistency Model
Conventional	Yes	Invalidate	1	Sequential
Migratory	No	Invalidate	1	Sequential
Dash	Yes	Invalidate	1	Release
Adaptive	Yes	Invalidate	1	Release
Munin	Yes	Update	Multiple	Release

Table 2 Summary of Consistency Protocols Investigated

The Conventional protocol represents a direct extension of a conventional bus-based write-invalidate consistency protocol to a directory-based implementation [1]. A node can only write to a shared cache line when it is the owner and has the sole copy of the block in the system. To service a write miss (or a write hit when the block is in read-shared mode), the faulting node sends an ownership request to the block's home node. If the block is not being used or is only being used on the home node, the home node gives the requesting node ownership of the block (2 messages). Otherwise, the home node forwards the request to the current owner, which forwards data to the requesting node (5 messages including 2 messages for change of ownership). If the block is read shared then the home sends invalidate messages to all other nodes that still have cached copies of the block and the requesting node waits for acknowledgements ($2 * N$ messages, where throughout this discussion N is the number of remote nodes caching the data). To service a read miss, the local processor requests a copy of the block from the block's home node. If the home node has a clean copy of the block, it responds directly (2 messages). If not, it forwards the request to the current owner, which forwards a copy to the requester and the home node (4 messages).

Cache blocks being kept consistent using the Migratory protocol are never replicated, even when read by multiple processors with no intervening writes. Thus, both read and write misses are treated identically. When a processor misses on a cache block, it requests a copy of the block from the home node. If the home node has a copy, it returns it directly (2 messages), otherwise it forwards the request to the current owner, which responds directly (3 messages). This protocol is optimal for data that is only used by a single processor at a time, such as data always accessed via exclusive RW locks, because it avoids unnecessary invalidations or updates when the data is written after it is read. Several researchers have found that the provision of a migratory protocol can improve system performance significantly [13, 27].

Dash is identical to the Conventional protocol, except that the new owner of a cache block does not have to stall while it waits for acknowledgements to its invalidation messages. This optimization assumes that the program is written using sufficient synchronization to avoid data races, which is most often the case. The details of why this results in correct behavior is beyond the scope of this paper – we refer you to the original DASH papers for a detailed explanation [16, 21]. For simplicity, we assume that all acknowledgement messages arrive before the processor reaches a release point, and thus do not count invalidation acknowledgements for the Dash protocol.

The Adaptive protocol is a version of Dash modified to detect migratory sharing patterns and dynamically switch between the replicate-on-read-miss protocol of Dash and the migrate-on-read-miss protocol of Migratory [13, 27]. All cache lines start out in Dash mode and transition to migratory if on a write hit requiring an invalidation there are exactly two copies of the line and the current invalidating processor is not the last processor to perform an invalidation. A read miss on a line in Migratory mode will then migrate the line to the requesting processor rather than replicate it. A line drops out of Migratory mode on any miss for a line that has not been dirtied since the last migration.

The Munin protocol is similar to the "write-shared" protocol employed in the Munin DSM system [8, 9]. Unlike the other three protocols, it uses an update-based consistency mechanism and allows multiple nodes to modify the cache line concurrently. Like Dash, it exploits the power of release consistency to improve performance. Specifically, a node is allowed to write to any cache block that it has without communicating with any other node, regardless of whether or not the block is currently replicated. To maintain consistency, when a node performs a release operation as defined by release consistency [16] (releases a lock, arrives at a barrier, etc.), it is required to update all replicas of blocks that it has modified since the last release. To support this protocol, the hardware is assumed to have a per-word dirty bit so that a node only need send the particular words within the block that it has modified. In this way, multiple nodes are able to write to different words in the same block concurrently. With this scheme, write races can be detected easily [7, 9]. In the basic scheme, a read miss is handled by requesting a copy of the data from the home node, which always has a usable copy (2 messages). Writes hits are performed completely locally, and entail only setting a dirty bit for the word that is being written and keeping track of the fact that the line has been modified (0 messages). A write miss is handled as a read miss followed by a write hit (2 messages). Unlike the previous protocols, MUNIN must perform memory consistency operations when it arrives at a release point. In the basic implementation, the MUNIN protocol sends an update to the home node of each dirty line and wait for all of these updates to be acknowledged before performing the release operation. An update message consists of a starting address, a bitmap of the modified words in the line, and the new values of the modified words. When a home node receives an update message for a cache line, it forwards a copy of the update message to each remote node caching that line and waits for the update messages to be acknowledged. After all of the update messages have been acknowledged, it sends an acknowledgement message to the node performing the release. The update messages can be performed asynchronously. This basic protocol requires $2*N$ messages per dirty cache line during each release operation (where N is the number of copies of each line).

There are a number of obvious problems with this basic implementation of the MUNIN protocol. First, it does not take advantage of the fact that (i) most updates are to small subsets of the complete cache line, especially for large cache lines and (ii) often many updates need to be sent to the same remote nodes. These two facts allow multiple updates to be combined into single messages, as was done successfully in the Munin system [9]. Except when otherwise noted, we assume that the hardware is smart enough to put several small updates destined to the same remote node into a single message when there is space. This combining occurs both when a releasing node is sending multiple updates to the same home node, and when a particular home node is updating multiple cache lines on a remote node simultaneously. We do not assume that the hardware is capable of fragmentation or reassembly, so no updates are allowed to span multiple messages. With this optimization, the MUNIN protocol requires as many as $2*N$ messages per dirty line, but often quite a bit fewer. This optimization proved especially valuable.

A second problem with the basic implementation is that, like conventional write-update protocols, there is no limit to how long a node will continue to receive updates for a block of data that it is no longer using unless the block happens to be deleted from the cache due to contention for a cache line. This can result in a large number of unnecessary updates to cache lines that are no longer being used (so-called "stale" lines). To address this problem, cache lines being maintained using the MUNIN protocol are automatically invalidated when they appear to have become stale. We evaluated two strategies for timing out stale data. In the first strategy, a cache line is invalidated after receiving T updates since it is last referenced, where T denotes some timeout count. In the second strategy, a cache line that is not referenced for two consecutive releases is invalidated by the processor performing the release. Both strategies require only a small (one or two bit) counter associated with each cache line. We found that the second strategy, apart from being simple to implement, resulted in very good performance for all of the SPLASH programs and hence was employed in our comparisons. To invalidate a local cache line, a node simply sends an invalidation message to the line's home node, including a copy of the modified data if the line is dirty.

Finally, we also measured what we will refer to as the OPTIMAL algorithm. This algorithm assumes that software (e.g., the compiler) has correctly specified to the hardware on a per-cache-line basis the optimal protocol to use for that cache line during that particular execution of the application being simulated. We do this by determining off-line which protocol required the least communication for each cache line, and using this protocol's results for that line in our calculation of Optimal's performance. This measurement gives us a best case measurement of the *potential* performance of a programmable cache controller-based system using the five protocols described above, if software is able to perfectly specify in advance how each block of memory should be handled and does not modify this hint during program execution. While it is probably not reasonable to assume that this performance is achievable in general, it provides us with the answer to a very fundamental question: Is it worthwhile to even consider building a programmable cache controller, or are the potential benefits too small to be worth the effort? As we shall see, there are clear benefits to building and utilizing a programmable controller.

Table 3 summarizes the number of messages transmitted by each protocol in response to different system events.

2.5 Assumptions

In all simulation studies, it is important to be clear exactly what is and what is not being measured and what assumptions are being made. Since the goal of this research is to explore the potential impact of programmable cache controllers and not to evaluate the performance of one particular design, our multiprocessor model was constrained in a number of ways. Specifically, our multiprocessor model makes the following assumptions:

- Shared data is never removed from the cache due to contention for a limited number of cache blocks (i.e., there is "infinite" cache).
- All program code and non-shared data fits into local "per processor" memory (i.e., we do not measure message traffic for accesses to instructions or non-shared data).
- We do not assume any particular network topology or performance. This has a number of effects, including:
  - We could not model contention in the network.
  - We could not assume that the network supported multicast.
  - We could not assume that the network reliably delivered packets in order.

Consistency Protocol	Read Miss	Read Hit	Write Miss (Replicated)	Write Hit (Replicated)	Release
Conventional	(2,4)	0	$(2,5)+2*N$	$2+2*N$	0
Migratory	(2,3)	0	$(2,3)$	—	0
Dash	(2,4)	0	$(2,5)+N^{\dagger}$	$2+N$	0
Munin	2	0	2	0	$\sum^{\forall d:dirty} 2*N_d$[‡]

Table 3 Summary of Messages Required to Maintain Consistency

[‡] There are actually $2*N$ messages, but we only count N of them. This is explained in Section 2.5.

[‡] This value is the upper bound assuming one update per message. In practice, it is often 20-60% lower.

- The network can send a complete cache line in a single message. However, we did evaluate the effect of fixing the maximum packet size at 128 bytes (see Section 3.4).

- We do not measure *latency* or *cycles*, since these measurements require a specific system model. Instead, we measure hardware independent parameters such as the cache hit ratio, and the number of "messages" transmitted to maintain consistency.

- We assume that the write buffer in DASH never fills up and stalls the processor on a write. We also assume that all DASH invalidation messages are acknowledged before the following release, and thus do not count ownership changes or invalidate acknowledgements in the message totals for DASH. This matches well with empirical experience with DASH.

The first two assumptions restrict our attention to the effect that using a programmable cache controller has on consistency maintenance, which is the goal of this research. By choosing a simple abstract model for the processor interconnection network, our results should translate well across multiple communication topologies. We measure the number of cache misses, the number of messages used to maintain consistency, and the size of these messages, so a rough estimate of system performance can be calculated for any particular processor interconnect design. Although assuming that the network can transmit cache lines in a single message may not be entirely realistic for very large cache lines, e.g., 512 bytes, we assume that in a scalable shared memory multiprocessor the network will be designed to not require fragmentation and reassembly of cache line messages. We evaluated the impact of this assumption and present the results of this evaluation in Section 3.4.

3 Results

We simulated the performance of our multiprocessor model on a large number of configurations. For each of the five applications, we measured the performance of each protocol individually on from 2 to 32 virtual processors (by powers of 2) and for cache line sizes ranging from 32 to 512 bytes. We collected a large number of statistics for each run, including the number of cache hits and misses (reads and writes), the number of data request messages, the number of invalidate or update messages, and the number of acknowledgement messages. In addition, we calculated the number of hits, misses, messages, and other characteristics for an off-line "optimal" protocol. The extent to which the OPTIMAL protocol improves upon individual protocols is an indication of the potential performance improvements that can be achieved by supporting multiple protocols in hardware and letting software statically select the protocol to use on a per-cache-line basis. Further improvement may be achievable with dynamic selection, i.e., hints that change during the execution of the program, but we have not yet evaluated this possibility.

To evaluate the value of supporting multiple protocols in hardware and allowing software to select the "optimal" protocol for each cache line, we performed two basic sets of experiments. In the first set, we measured the performance of the various protocols for each application on a fixed number of processors (8) as we varied the cache line size from 32 to 512 bytes. The results of these experiments are discussed in Section 3.1. In the second set of basic experiments, we measured the performance of the various protocols for each application for a fixed line size (128 bytes) as we varied the number of processors from 2 to 32. These results, discussed in Section 3.2, did not show as much variance as the results obtained when we modified the cache line size, so we chose to present only the results for only one application, `mp3d`.

In addition to the two basic sets of experiments, we measured the impact of supporting a limited number of protocols and of fixing the maximum packet size in the interconnect to 128 bytes (as opposed to setting it equal to the cache line size). The former experiment is discussed in Section 3.3 and the latter is discussed in Section 3.4.

For all of our experiments, measured the *cache miss rate* and the *number of messages* required to maintain consistency. Together these values give a good indication of the overall performance of a shared memory multiprocessor.

3.1 Effect of Varying Line Size

The results for our experiments on the effects of varying line size on the choice of an "optimal" protocol are presented in Figures 3 through 12 and Tables 4 through 7. The figures plot the cache miss rates and number of messages required to maintain consistency for each of the five applications on eight processors as we vary the cache line size. The values presented in the tables are the percentage of cache lines in each configuration for each application that are "optimally" maintained using each protocol. For example, in Table 5, 34.9% of the cache lines in `mp3d` are read-only (and thus should be handled by any protocol but Migratory, 64/9% of the cache lines should be handled using the Migratory protocol, and 0.3% of the cache lines should be handled using the Munin protocol.

MP3D

As illustrated in Figure 4, using the optimal distribution of consistency protocols across the cache lines can reduce the number of consistency messages from 10% to over 80% compared to any single fixed protocol, depending on the line size and specific protocol being compared against. This observation, which holds true to varying degrees across all five of the applications, strongly supports the notion that shared memory multiprocessor performance can be significantly improved by the use of flexible cache controllers, if the software is capable of determining the optimal (or near optimal) protocol to use to keep a particular block of data consistent.

As can be seen in Tables 5-7, `mp3d` has a high percentage of migratory data. Thus, for small line sizes, the Migratory and Adaptive protocols work best, which is evident in Figure 4. As the size of the cache lines is increased, the effects of false sharing cause Munin to perform better and the invalidation-based protocols to perform more poorly, as expected given Munin's good handling of false sharing [9].

Figure 3 illustrates that the use of the "optimal" protocol for each cache line can also reduce the cache miss rate, despite the fact that our measure of "optimal" minimizes the number of consistency messages, with no attention paid to the effect on cache miss rate. As with messages, the Munin protocol performs relatively poorly for small cache lines (32 bytes) because it is unable to combine updates, while Dash and Adaptive perform best overall. The reverse becomes true as cache lines grow to and beyond 128 bytes. While Migratory requires the fewest messages for small cache lines, its inability to support read sharing results in a high miss rate. These results are another indication that using the "Optimal" protocol for each cache line can improve performance. For `mp3d` with small cache lines, the write-shared lines (lines that are shared and written at least once) are best handled by the Migratory protocol, while the read-shared lines (lines that are shared, but never written after initialization) should be supported by any of the other protocols, all of which support read replication. As we can see in Tables 5 - 7, as the cache line size grows, the write-shared data becomes better handled by the Munin protocol than the Migratory protocol. In the case of `mp3d`, the standard Dash protocol is rarely optimal, although its average performance is quite good (i.e., it is a good compromise if only one protocol can be supported).

There is a "crossover" in the curves at the point where cache lines grow to 128 bytes. Before this point, the more conventional invalidate-based protocols perform best, but after this point their performance degrades while Munin's improves as it is able to more often pack multiple updates in a single message. For large cache lines and messages (512 bytes), the optimized Munin protocol was able to send an average of three updates per message.

Water

Overall, water is far better behaved as an application than mp3d, which is notorious for the fine granularity of its sharing. As seen in Figures 5 and 6, the miss rates for water are approximately one-third those of mp3d, and water requires approximately an order of magnitude fewer consistency messages, albeit to perform one-third as many memory operations (see Table 3). Nevertheless, tuning the way in which individual cache lines are maintained still reduces the amount of communication from 10% to 75%, with a mean improvement of approximately 40%. However, in this case optimizing for message traffic does not significantly improve the cache miss rate, and in fact the "optimal" performance can dip slightly below that of Adaptive (see Figure 5). For water, both Dash and Adaptive perform almost as well as the "optimal" distribution, while Munin performs relatively poorly. This result is another indication that it is useful to tune the cache's behavior based on the way in which data is being accessed. Somewhat surprisingly, although Migratory performs quite poorly in the aggregate, as seen in the graphs, it contributes a significant percentage of the "optimal" allocation of protocols, an indication that even in a single program, the "optimal" protocol varies from line to line.

Barnes-Hut

Figures 7 and 8 show the performance of barnes. The results for Migratory are omitted, as for both the miss rate and messages, it performed an order of magnitude worse than any other protocol for all line sizes. The primary reason for this poor performance is that there is a large amount of read-shared data, which Migratory does not support effectively. Compared to the other protocols, the Optimal protocol requires from 10% to 50% less communication to maintain consistency, while performing at least as well as the alternatives in terms of miss rates (except for Munin for very large line sizes).

Like mp3d, there is clearly a point after which the performance of the invalidation-based protocols drops off, while Munin becomes increasingly important. This observation is reinforced by the data in Tables 5-7. The distribution of "optimal" protocols across cache lines changes dramatically as the line size is varied. For small cache lines, Dash performs almost as well as the much more complicated Optimal protocol. However, for larger line sizes, Dash does not fare well. However, the miss rates and message counts for barnes are quite low, so probably any protocol would achieve acceptable performance.

Cholesky Factorization

As illustrated in Figures 9 and 10 and Tables 5-7, the performance of the various protocols on cholesky is very stable, regardless of the line size. Individually, all of the protocols except Migratory perform approximately equally in terms of both miss rate and messages. However, the use of an "optimal" distribution of protocols results in 25% to 75% fewer messages being transmitted than is done for any single protocol, with no negative impact on the miss rate. As with water, although

Migratory performs poorly when it is the only protocol supported, it is a major contributor to the optimal distribution.

LocusRoute

Like water and cholesky, what is the best protocol varies wildly from line to line in locus. Thus, the "optimal" distribution results in 30% to 70% fewer messages and a 20% to 50% lower miss rate than any individual protocol. Like mp3d and barnes, there is a threshold at approximately 128 bytes where the optimal protocol for most write-shared data changes from Dash/Adaptive to Munin. In any event, the performance of locus gives us another strong indication of the potential impact of a programmable cache controller.

3.2 Effect of Varying Number of Processors

All five of the applications displayed similar characteristics when we varied the number of processors. The reason for this is that, in general, the number of processors caching any given write-shared line was not affected by the number of processors involved in the computation. Thus, while the miss rate and total number of messages grew as we added processors, thus reducing the granularity of sharing and increasing the number of read misses to read-only data, the relative performance of the various protocols and the impact of using the Optimal protocol varied very little. Figures 13 and 14 illustrate this effect for mp3d. Other than the relatively poor performance in terms of message count of the Munin protocol for 32 processors, the relative performance of the different protocols did not change significantly as we varied the number of processors. Thus, the observations made in Section 3.1 would seem to be relatively machine-size independent, with the exception of the Munin protocol. We conjecture that the poor scalability of the Munin protocol would be eliminated if we limited the number of processors that can simultaneously cache a given line of data, which is required in large scale machines already because bitmaps are not scalable data structures [11].

3.3 Effect of Limiting the Number of Protocols

To evaluate how effective a controller could be with limited "smarts," we measured the optimal performance of a programmable controller when only a subset of the protocols was implemented.

Comparing the performance of the Adaptive protocol with an Optimal run involving just Dash and Migratory shows how effective Adaptive is at approximating Dash or Migratory depending on the dynamic access behavior. The former dynamically switches cache lines between the two protocols whereas the latter is a static partitioning of lines between the two protocols based on which protocol performs better over the life of the application. Figures 15 and 16 gives the results of this comparison, in terms of the number of messages required to maintain consistency. In Figure 15, it can be seen that there are applications, in this case locus, where Adaptive is a poor approximation to the optimal combination of Dash and Migratory, which is in turn a poor approximation to the overall Optimal protocol combination. This was also the case for barnes and mp3d. Figure 16, on the other hand, illustrates the fact that sometimes Adaptive is a good approximation of Optimal, in this case for cholesky. Nevertheless, while Adaptive works noticeably better than static implementations of Dash or Migratory, in general it does not compare well with an optimal combination of the two. The fact that the optimal static protocol assignment consistently outperforms the dynamic approximation is yet another indication of the potential of programmable cache controllers.

Program	Reads	Writes	Total
mp3d	3288072	2206416	5494488
water	1726569	191781	1918350
cholesky	7914864	287770	8202634
barnes	16622122	3243245	19865367
locus	1510982	418220	1929202

Table 4 Number of Shared Data Accesses (eight processors)

Protocol	mp3d	water	barnes	cholesky	locus
Read-only	34.9%	38.6%	12.1%	22.5%	46.2%
Dash	0.0%	10.4%	45.3%	0.0%	1.6%
Migratory	64.9%	51.0%	32.5%	77.4%	50.2%
Munin	0.3%	0.0%	10.1%	0.0%	2.0%

Table 5 Contributions of the Various Protocols to "Optimal"
(eight processors, 32 byte cache lines)

Protocol	mp3d	water	barnes	cholesky	locus
Read-only	33.5%	34.9%	10.8%	22.4%	40.2%
Dash	0.1%	17.6%	8.8%	0.1%	0.5%
Migratory	58.4%	45.6%	7.8%	76.6%	46.4%
Munin	8.1%	1.8%	72.6%	0.9%	12.9%

Table 6 Contributions of the Various Protocols to "Optimal"
(eight processors, 128-byte cache lines)

Protocol	mp3d	water	barnes	cholesky	locus
Read-only	39.4%	31.7%	14.9%	22.1%	33.5%
Dash	0.2%	54.8%	0.3%	0.0%	0.8%
Migratory	13.9%	10.3%	0.0%	67.1%	41.8%
Munin	46.5%	3.2%	84.8%	10.7%	23.9%

Table 7 Contributions of the Various Protocols to "Optimal"
(eight processors, 512-byte cache lines)

More generally, limiting the number of protocols can significantly impact performance when you do not include at least one update-based and one invalidate-based protocol, at least for large cache line sizes, as can be seen in Figure 17 for mp3d. Similar behavior was noted for both barnes and locus. For water and cholesky, however, the level of sharing was sufficiently low that the impact of restricting the number of protocols supported by the hardware was minimal, as illustrated in Figure 18 for cholesky.

In summary, when there is not a high degree of sharing, the number of protocols has little impact on optimal performance. However, for programs with a significant amount of sharing, the provision of at least one invalidate and one update protocol is important, especially for larger cache line sizes.

3.4 Effect of Fixed Message Sizes

Finally, to verify that our assumption that the network could transmit cache lines in a single message does not make the programmable controller appear to be overly effective, we examined the impact of fixing the packet size at some reasonable minimum, specifically 128 bytes. We chose this value because it matches the payload size of the high-speed network controller that we are developing as the backbone of our scalable parallel architecture. Figure 19 shows the effect that this has on mp3d for eight processors. We compared the performance of the Dash and two variants of the Munin protocols, each for fixed and variable sized messages. For Dash fixing the message size at 128 bytes has no impact for small cache line sizes, because the small cache lines fit entirely in 128 byte messages and Dash never has an opportunity to combine multiple events in a single packet. However, fixing the message size at 128 bytes significantly increases the number of messages required to maintain consistency for large line sizes, because Dash always transmits full cache lines (being an invalidate protocol). The variant of Munin labeled as Munin' does not combine update messages being sent to the same node, so like Dash, it does not benefit from the large message size for small line sizes. However, its performance does not degrade as rapidly as Dash's for large line sizes, because updates often consist of far fewer bytes than a full cache line. mp3d was chosen because it tends to send large updates, unlike several of the other applications, to give Munin less of an advantage. Finally, the version of Munin that combines messages clearly outperforms the other two protocols. It can both benefit from large message sizes when the cache lines are small, or from cache-line-sized messages when cache lines are large. This experiment also illustrates the power of combining updates, a scheme that worked well in software for Munin [9] and seems to translate well to hardware.

4 Related Work

There are a number of ongoing academic efforts with the goal of designing a scalable shared memory multiprocessor. All of these projects are currently emphasizing the mechanisms needed to implement a flexible cache controller, rather than the policies and software required to exploit this flexibility.

The proposed user level shared memory in Tempest and Typhoon system [25] will support cooperation between software and hardware to implement a scalable shared memory and message passing abstraction. Tempest provides an interface allowing user level code to efficiently utilize the underlying low-level communication and shared memory mechanism. Typhoon is a proposed hardware implementation for this interface. Like the Alewife system, the proposed system uses low level software handlers and provides more flexibility. As such, it requires extensive program modifications and user effort to achieve the required performance.

The Stanford DASH multiprocessor [22, 21] uses a directory-based cache design to interconnect a collection of 4-processor SGI boards based on the MIPS 3000 RISC processor. DASH's cache consistency protocol was described in Section 2.4. A second generation DASH multiprocessor is being developed that adds a limited amount of processing power and state at the distributed directories to add some flexibility to the consistency implementation. This machine, called FLASH [20] will support both DASH-like shared memory and efficient message passing, although it should have the power to implement other caching protocols such as the ones described above.

The MIT Alewife machine [10, 11] also uses a directory-based cache design that supports both very low-latency message passing and shared memory based on an invalidation-based consistency protocol. The Alewife designers are currently extending the Alewife directory implementation to add a limited amount of flexibility by allowing the controller to invoke specialized low-level software trap handlers to handle uncommon consistency operations. This extension could allow multiple consistency protocols to be supported by the hardware, but currently the Alewife designers are only planning to use this capability to support an arbitrary number of "replica" pointers (a list of the nodes in the system that are caching a given line) per cache-line, and a limited number of specialized synchronization operations.

Sesame is a fast hardware network interface that supports distributed shared memory on a network of high-speed workstations [32]. Sesame aggressively attacks the problem of communication latency caused by demand-driven data transfers by selectively sending updates to shared data in advance of when they are requested, so-called *eager sharing*. Their results indicate that eager sharing allows a hardware DSM system to scale well under circumstances that limit conventional demand-driven consistency protocols to an order of magnitude less performance.

5 Conclusions and Recommendations

We have explored the impact that the use of a programmable cache controller can have on the performance of a shared memory multiprocessor, and found that it has the potential to be dramatic. In our simulation study of five of the SPLASH programs running on a directory-based multiprocessor that supported multiple consistency protocols and could accept software pragmas, we found that the optimal distribution of protocols to individual cache lines could reduce the amount of communication by from 10-80%, generally from 25-35%, compared to using a single, hardwired protocol. Furthermore, for several of the applications, this tuning process reduced cache miss rates by up to 25% despite being optimized for reducing the number of consistency messages. This indicates that it is worthwhile investigating (i) hardware techniques for supporting software control of multiprocessor caches [10, 20, 25] and (ii) software techniques for fully exploiting this flexible hardware [25]. If successful, such systems could allow shared memory multiprocessors to scale to far larger designs than is currently feasible, without requiring prohibitively expensive and complicated controllers and interconnects. We plan to refine this study to determine whether its conclusions remain valid on a more detailed system model, and use its results to aid in the design of a controller and software environment like that described above.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer*

Architecture, pages 280–289, June 1988.

[2] G. Astfalk. Past progress and future abilities in high performance computing. In *Proceedings of a 1993 meeting of the Max Planck Society in Gemrany*, 1993.

[3] G. Astfalk, T. Breweh, and G. Palmeh. Cache coherency in the convex mpp. *Convex Computer Corporation*, February 1994.

[4] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 125–134, May 1990.

[5] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *COMPCON '93*, pages 528–537, February 1993.

[6] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR1 computer system. Technical Report KSR-TR-9002001, Kendall Square Research, February 1992.

[7] J.B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency.* PhD thesis, Rice University, August 1993.

[8] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[9] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[10] D. Chaiken and A. Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324, April 1994.

[11] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[12] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, May 1994.

[13] A.L. Cox and R.J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[14] H. Davis, S. Goldschmidt, and J. L. Hennessy. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Stanford University, 1990.

[15] S.J. Eggers and R.H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, May 1988.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 1990.

[17] A. Gupta and W.-D. Weber. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.

[18] J. Heinlein, K. Gharachorloo, and A. Gupta. Integrating multiple communication paradigms in high performance multiprocessors. Technical Report CSL-TR-94-604, Stanford Computer Systems Laboratory, February 1994.

[19] Intel Supercomputers Systems Division. Paragon technical summary, 1993.

[20] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, May 1994.

[21] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

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

[23] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[24] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[25] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[26] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[27] P. Stenström, M. Brorsson, and L. Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[28] Thinking Machines Corporation. The Connection Machine CM-5 technical summary, 1991.

[29] J.E. Veenstra and R.J. Fowler. A performance evaluation of optimal hybrid cache coherency protocols. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 149–160, September 1992.

[30] J.E. Veenstra and R.J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *MASCOTS 1994*, January 1994.

[31] A. Wilson and R. LaRowe. Hiding shared memory reference latency on the GalacticaNet distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 15(4):351–367, August 1992.

[32] L.D. Wittie, G. Hermannsson, and A. Li. Eager sharing for efficient massive parallelism. In *1992 International Conference on Parallel Processing*, pages 251–255, St. Charles, IL, August 1992.

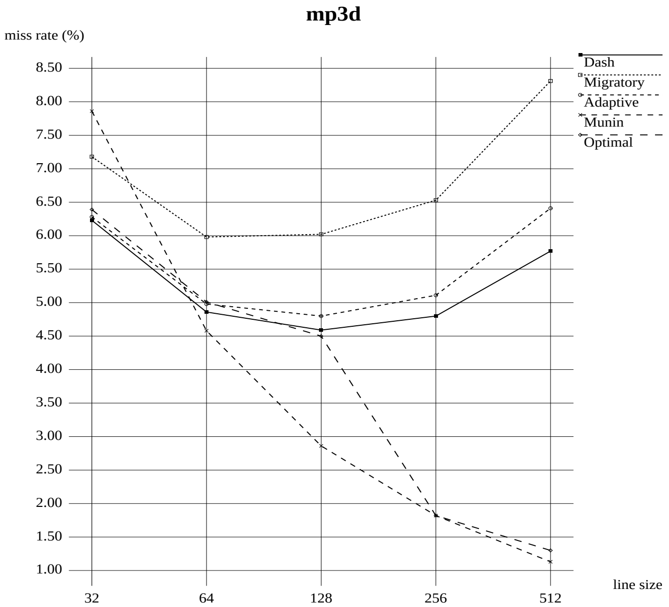

Figure 3 Miss Rate vs Line Size (`mp3d`)

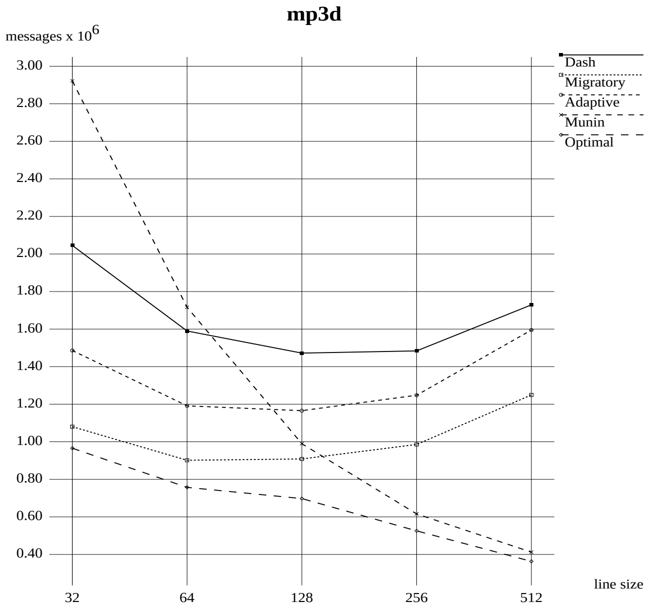

Figure 4 Messages vs Line Size (`mp3d`)

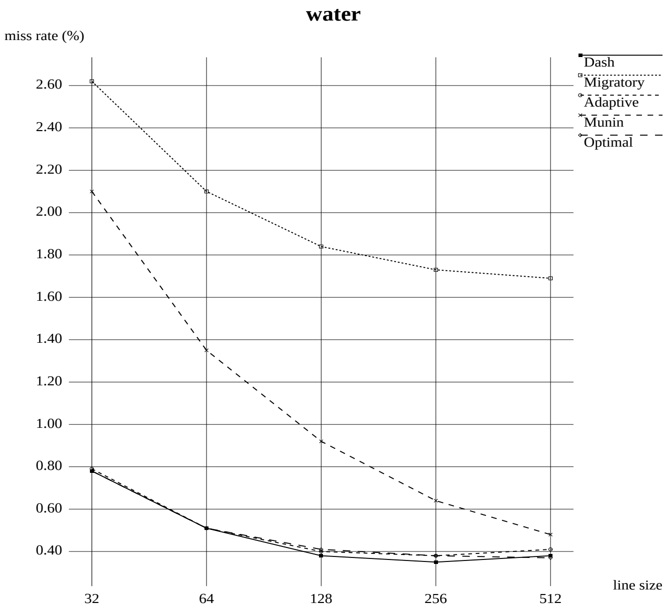

Figure 5 Miss Rate vs Line Size (`water`)

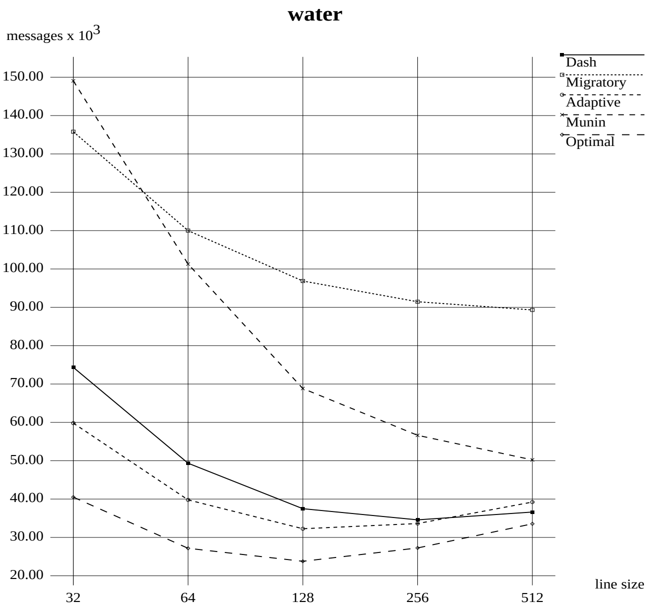

Figure 6 Messages vs Line Size (`water`)

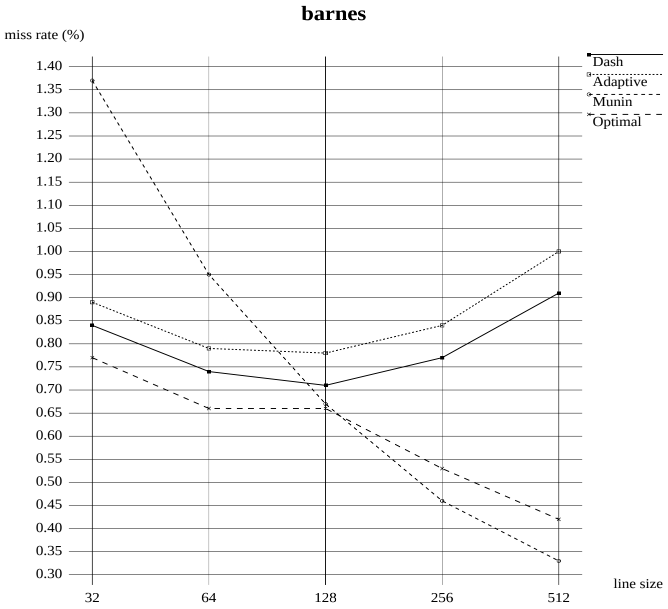

Figure 7 Miss Rate vs Line Size (`barnes`)

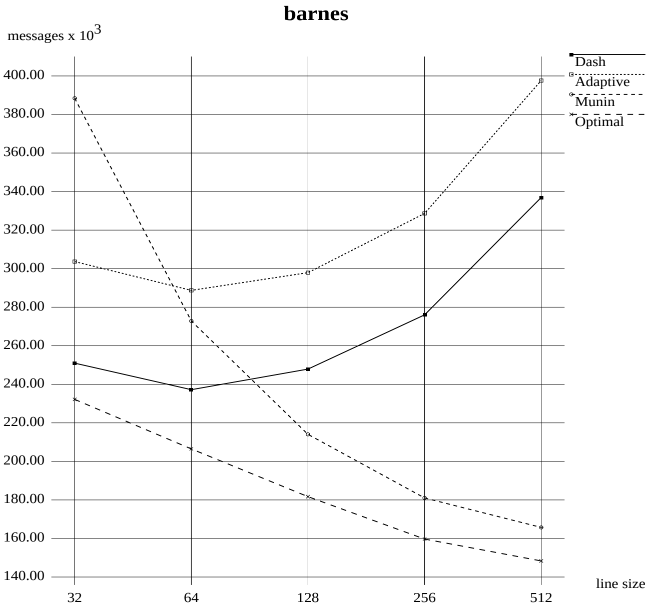

Figure 8 Messages vs Line Size (`barnes`)

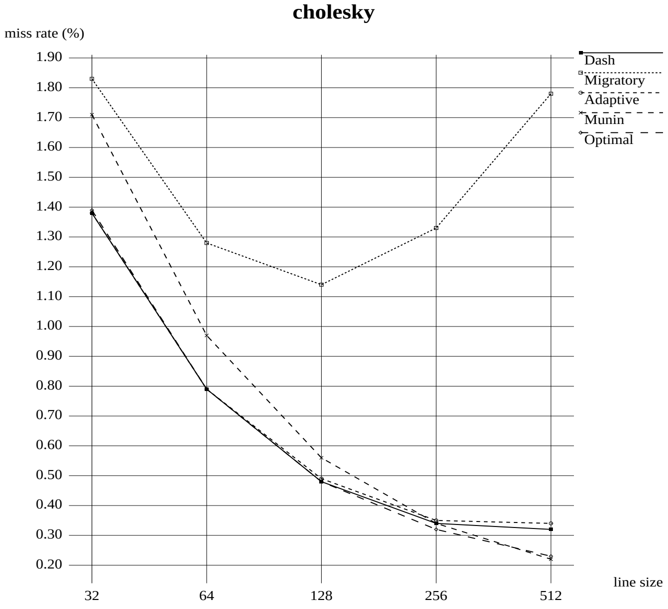

Figure 9 Miss Rate vs Line Size (`cholesky`)

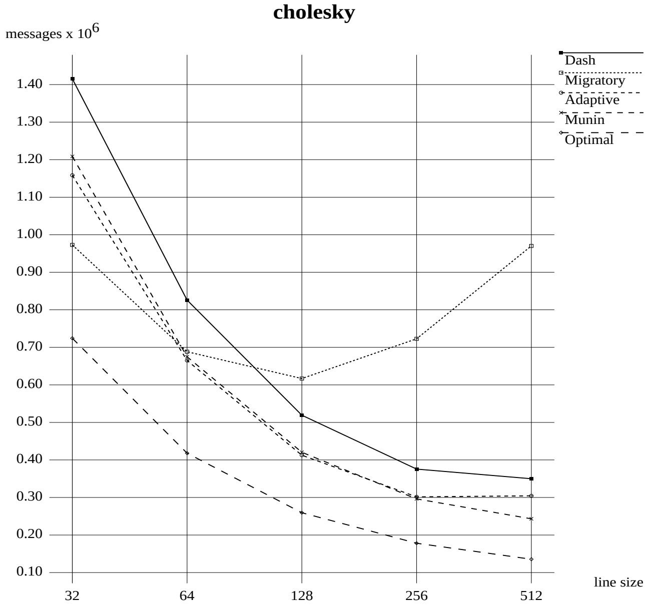

Figure 10 Messages vs Line Size (`cholesky`)

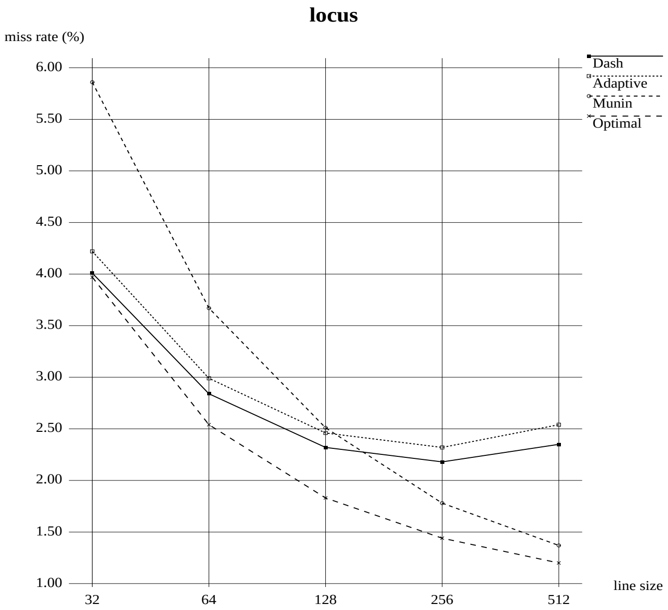

Figure 11 Miss Rate vs Line Size (`locus`)

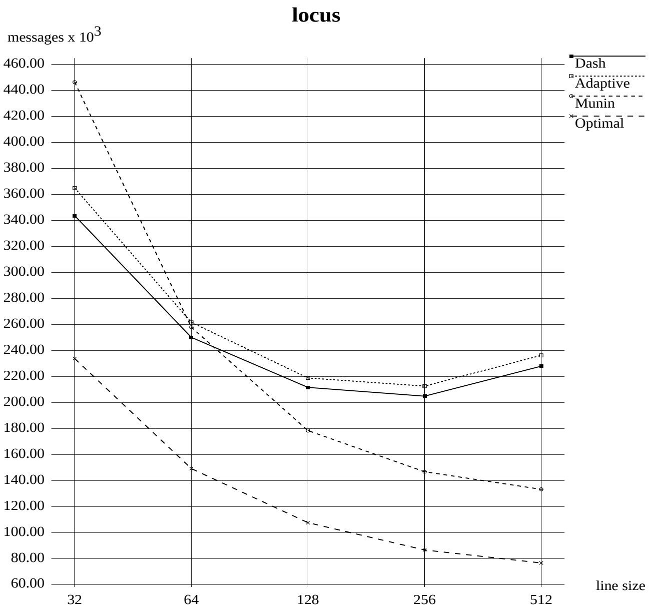

Figure 12 Messages vs Line Size (`locus`)

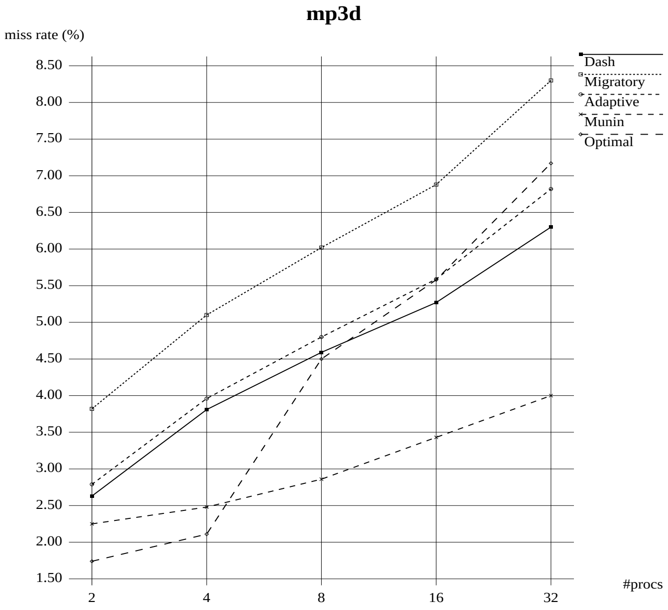

Figure 13
Miss Rate vs # of Processors
(`mp3d`, 128-byte cache lines)

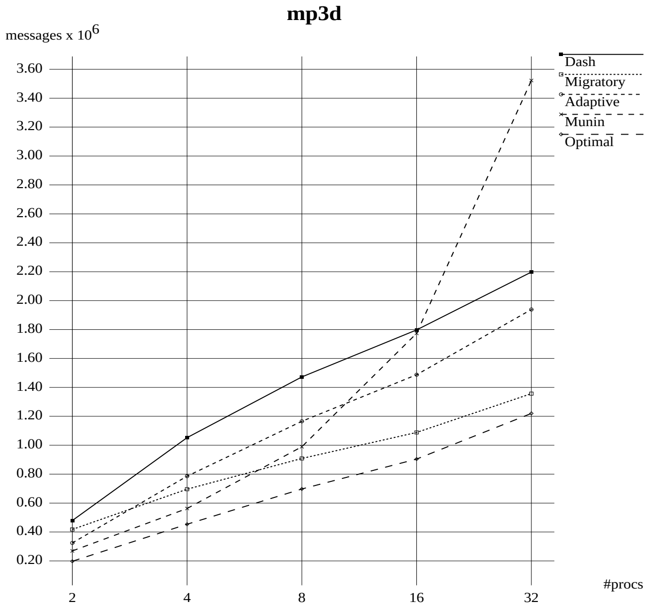

Figure 14
Messages vs # of Processors
(`mp3d`, 128-byte cache lines)

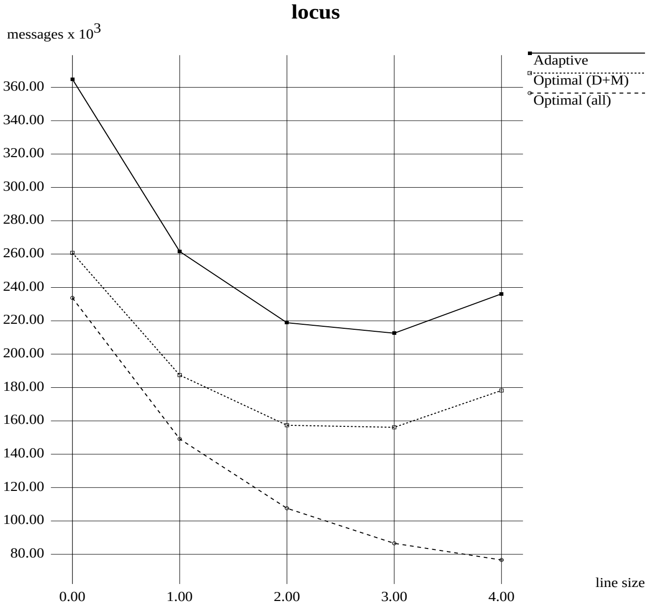

Figure 15 ADAPTIVE VS OPTIMAL(DASH + MIGRATORY)

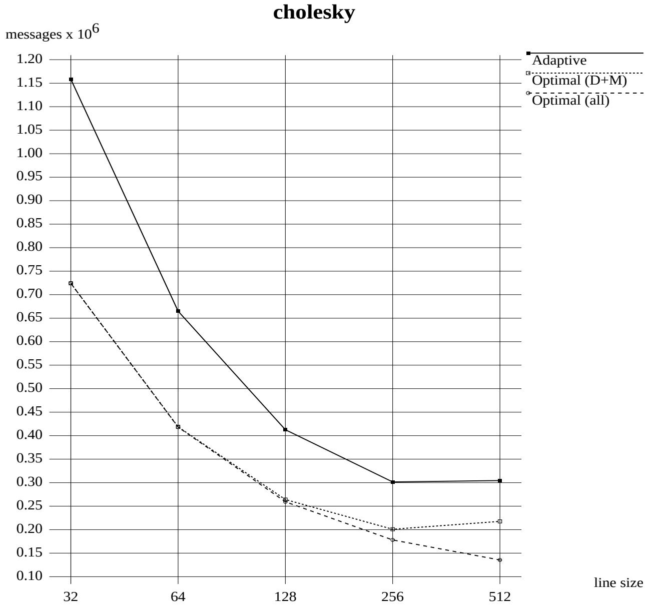

Figure 16 ADAPTIVE VS OPTIMAL(DASH + MIGRATORY)

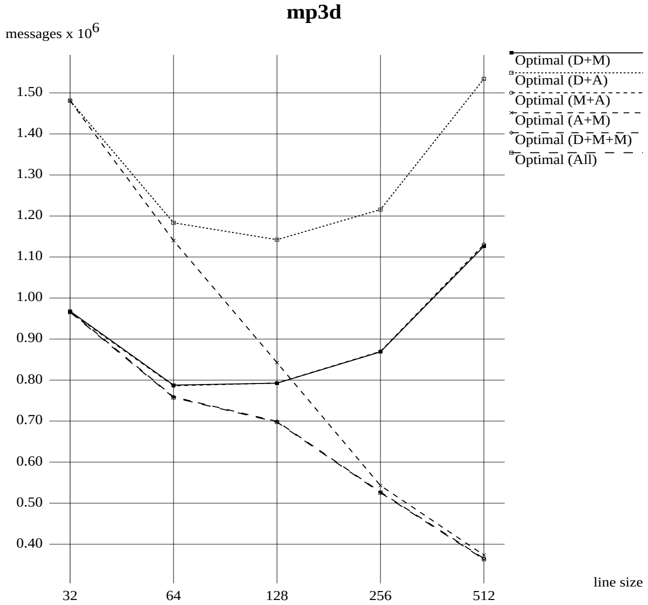

Figure 17 Effect of Limiting Protocols (`mp3d`)

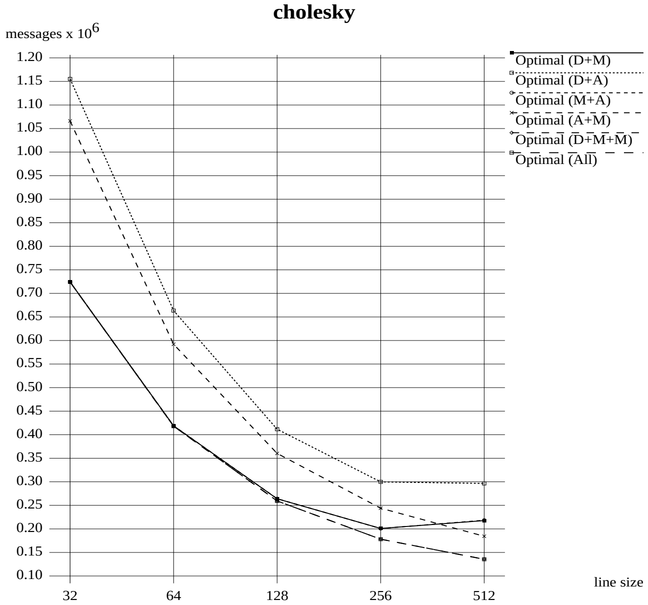

Figure 18 Effect of Limiting Protocols (`cholesky`)

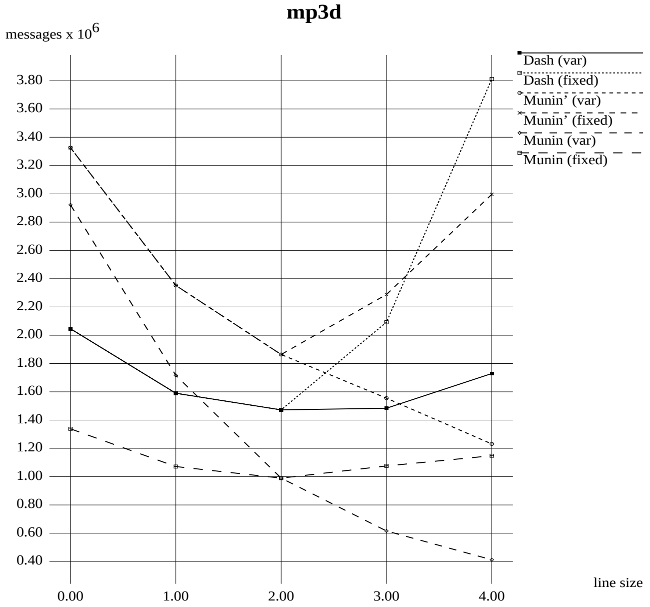

Figure 19 Effect of Fixing the Message Size (128 byte messages, `mp3d`, 8 processors)