Reducing Consistency Traffic and Cache Misses in the Avalanche Multiprocessor

John B. Carter, Ravindra Kuramkote, Chen-Chi Kuo

UUCS-95-023

Computer Systems Laboratory
University of Utah

Abstract

For a parallel architecture to scale effectively, communication latency between processors must be avoided. We have found that the source of a large number of avoidable cache misses is the use of hardwired write-invalidate coherency protocols, which often exhibit high cache miss rates due to excessive invalidations and subsequent reloading of shared data. In the Avalanche project at the University of Utah, we are building a 64-node multiprocessor designed to reduce the end-to-end communication latency of both shared memory and message passing programs. As part of our design efforts, we are evaluating the potential performance benefits and implementation complexity of providing hardware support for *multiple coherency protocols*. Using a detailed architecture simulation of Avalanche, we have found that support for multiple consistency protocols can reduce the time parallel applications spend stalled on memory operations by up to 66% and overall execution time by up to 31%. Most of this reduction in memory stall time is due to a novel release-consistent multiple-writer write-update protocol implemented using a *write state buffer*.

Reducing Consistency Traffic and Cache Misses in the Avalanche Multiprocessor

John B. Carter, Ravindra Kuramkote, Chen-Chi Kuo

Computer Systems Laboratory
University of Utah

1 Introduction

Existing "scalable" parallel architectures fail to address several critical design issues. Commercial microprocessors offer very impressive raw performance and seem to be attractive options for assembly into cost-effective parallel machines. However, the communication delay between tasks on different processors rapidly becomes the bottleneck. In the Avalanche project at the University of Utah, we are building a 64-node multiprocessor designed to reduce the end-to-end communication latency of both shared memory and message passing programs. This paper concentrates on the shared memory aspect of Avalanche. In shared memory multiprocessors, communication latency includes the time spent manipulating the hardware data structures used to manage the shared address space, the effect of contention between the local processor and remote processors for the local cache controller and memory busses, and the time spent servicing cache misses for data that has been invalidated as part of the coherence mechanism.

We have found that the source of a large number of avoidable cache misses is the use of hardwired write-invalidate coherency protocols. Conventional invalidation-based consistency protocols often exhibit high cache miss rates due to excessive invalidations and subsequent reloading of write-shared (or falsely shared) data. As deepening memory hierarchies cause main memory latencies to increase from 10's to 100's of cycles, these avoidable cache misses will seriously impede performance. Although scalability has been an important research theme over the past five years, achievement of this goal remains elusive. Evidence of this situation can be seen in the significant differences between the *peak* performance of today's fast multiprocessing systems and the *achieved* performance. For example, even highly-tuned applications often achieve well under 50% of peak performance on multiprocessors such as the CM-5[23] and Cray T3D[11] despite their powerful communication fabrics[3].

For a parallel architecture to scale effectively into the tera- and peta-op range, high latency cache misses must be avoided. Thus, cache controller designs and consistency protocols that reduce the frequency of cache misses must be developed. As part of the Avalanche project, we are modifying the memory architecture of a commercial RISC microprocessor, the HP PA-RISC 7100, to include a new multi-level context sensitive cache that is tightly coupled to the communication fabric (see Figure 1). At the core of our system is a flexible communication and cache controller unit (CCU) that will support multiple cache consistency protocols, exploit processor context information to avoid conflict misses between active tasks and incoming data, and allow dynamic prefetching of data to any level of the memory hierarchy. This paper concentrates on the first of these features of Avalanche's cache design – the use of multiple consistency protocols to reduce cache miss rates and improve overall memory utilization.

This research was supported in part by the Space and Naval Warfare Systems Command (SPAWAR) and the Advanced Research Projects Agency (ARPA), under SPAWAR contract No. N0039-95-C-0018 and ARPA Order No. B90 .

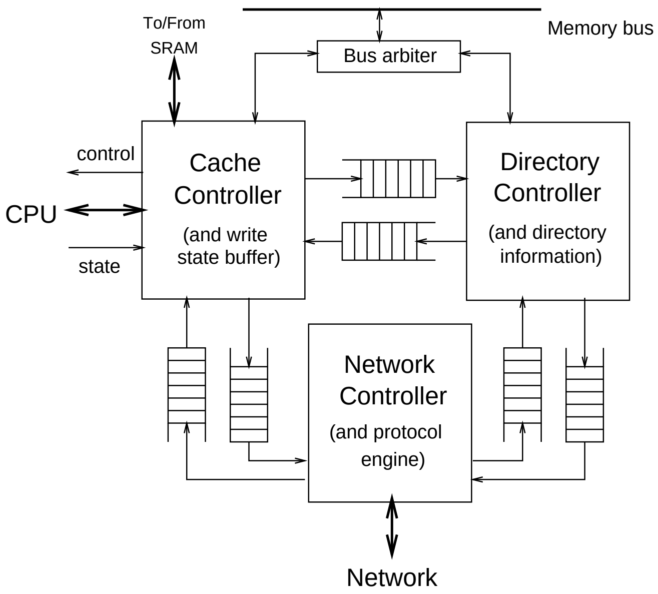

Figure 1 Overview of Avalanche Memory and Communication Architecture

To maximize performance, we believe that it is important for computer architects to provide "hooks" into their hardware to allow the software to tune the hardware's behavior at a low level (e.g., via compiler-generated or user-specified pragmas). One such "hook" that we are investigating is the ability to specify the coherency protocol that should be used to keep a shared memory program's data consistent. Using a detailed architecture simulation of Avalanche's modified CPU, memory system, and interconnect, we found that Avalanche's support for multiple consistency protocols can reduce the time parallel applications spend stalled on memory operations significantly - up to 66%. Most of this reduction in memory stall time is due to a novel release-consistent[13] multiple-writer write-update protocol implemented using a *write state buffer*. In addition, memory stall time can be further reduced by up to 6% if the protocol used to keep a particular program's data consistent can be varied on a per-page-basis, rather than on a per-program basis, via a simple TLB extension.

The remainder of this paper is organized as follows. Section 2 contains a more detailed overview of the Avalanche architecture. A description of the experimental setup used to evaluate the Avalanche CCU design (e.g., simulation environment, parameters explored, programs studied, and limitations) and the results of our simulations on a variety of workloads are presented in Section 3. Section 4 compares Avalanche with a number of related research efforts. Finally, in Section 5 we draw conclusions and outline our future endeavors.

2 Avalanche Cache Design

2.1 Basic Avalanche Architecture

The goal of the Avalanche project is to develop a communication and memory architecture that supports significantly higher *effective* scalability than existing multiprocessors by attacking all sources of end-to-end communication latency for both shared memory and message passing architectures. Our approach for achieving this goal is to design a flexible cache and communication controller that tightly integrates the multiprocessor's communication and memory systems, incorporates features designed specifically to attack the problem of excessive latency in current multiprocessor architectures, and makes extensive provisions for exploiting processor context information and software guidance.

A conceptual block diagram of an Avalanche node is given in Figure 1. Each Avalanche node contains a modified PA7100 processor, cache controller unit (CCU), directory controller (DC), network controller (NWC) and local memory. The CCU is the core "brains" of Avalanche – it manages the memory hierarchy and performs the protocol actions needed to maintain consistency between the various levels of the hierarchy and between separate nodes in the case of shared memory operation. The DC maintains the state of the distributed shared memory – each block of global physical memory has a "home" node and the DC on this home node keeps track of state information such as the protocol being used to manage that block of data and the nodes that have a copy of the data. The NWC transmits and receives messages from the inter-node network, queuing outgoing messages as necessary and routing incoming messages to the appropriate functional unit. The MMU and L1 cache controller have been stripped from the modified PA7100 and are incorporated in the CCU. The CCU, DC, and NWC are connected using separate FIFO input/output message buffers (IOMB). Messages are used to communicate between independent control units, both within a single node and across nodes. Messages contain commands (e.g., as part of handling a cache miss, the CCU may request that the DC managing the state of the required data modify its state) and to transmit data (e.g., as part of handling an incoming cache fill message, the NWC sends a message to the CCU requesting that the data be placed in the appropriate location in the memory hierarchy). Finally, the DC and CCU are connected to local memory through a local bus arbiter.

Our project differs from related related research projects [19, 18, 21] in a number of important ways. First, we do not treat the CPU as an unmodifiable black box – we are modifying the HP PA-RISC 7100 to give us complete control over all levels of the cache hierarchy and to allow us to export (import) additional control lines from (to) the processor to (from) the cache controller. In conjunction with Hewlett-Packard, we are designing a version of the HP PA-RISC 7100 chip with the cache controller and MMU moved off chip. Minor changes to the control path of the CPU and the pipeline stall blocks will be required to export the necessary context state information and maintain proper pipeline synchronization, but the processor core will not be modified. Modifying a complex chip such as the 7100 is not without its risks. However, doing so permits us to explore a wide set of design options and to determine what small set of modifications are cost effective for commodity microprocessor vendors should they desire to make their memory architectures better support highly scalable multiprocessing in addition to their core uniprocessor markets. A second difference between Avalanche and other similar research efforts is our degree of emphasis on giving the programmer/compiler the ability to tune at a fine grained level the low level actions of the cache and network controllers, and conversely the addition of hardware to collect runtime statistics (e.g., the virtual addresses causing the majority of conflict misses) that can be used to tune the software. An important aspect of this flexibility is Avalanche's support for multiple hardware coherency protocols, which we will discuss in detail in the following section.

To minimize our design effort and exploit existing commercial technology whenever possible, we are using the Myrinet[7] network as our multiprocessor backplane. Myrinet, which derived from the Caltech router project, is a high-speed mesh-connected network fabric designed for use both as a LAN network and as a multiprocessor backplane. As part of the Avalanche project we are developing an intelligent *protocol processing engine* (PPE) that will handle both conventional message passing traffic as well as consistency management traffic sent by the cache and directory controllers. We use Myrinet as the basis for our network model because even though it has a relatively high latency when compared to proprietary interconnects such as that found in the CM-

5, the Myrinet interconnect is the fastest commercially available interconnect suitable for our needs. While Myrinet's performance is slow compared to a fast special purpose interconnect like that found in the CM-5, we believe that we can mitigate this potential performance bottleneck limitation by reducing the amount of communication required to maintain coherence. Myrinet's performance characteristics are discussed in detail in Section 3.2.

2.2 Consistency Management

Spurred by scalable shared memory architectures developed in academia [1, 19], the next generation of massively parallel systems will support shared memory in hardware (e.g., machines by Convex, Cray, and IBM). However, current shared memory multiprocessors all support only a single, hard-wired write-invalidate consistency protocol[1] 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 [5, 14, 24]. These characteristic "patterns" are sufficiently different from one another that any protocol designed to optimize one will not perform particularly well for the others. In particular, the exclusive use of write-invalidate protocols can lead to a large number of avoidable cache misses when data that is being actively shared is invalidated and subsequently reloaded. The inflexibility of existing machines' cache implementations limits the range of programs that can achieve scalable performance regardless of the speed of the individual processing elements and provides no mechanism for tuning by the compiler or runtime system.

These observations have led a number of researchers to propose building cache controllers that can execute a variety of caching protocols [8, 26], support multiple communication models [10, 15], or accept guidance from software [18, 21]. We are investigating cache designs that will implement a variety of caching protocols, support both shared memory and message passing efficiently, accept guidance from software to tune its behavior, and directly support efficient high-level synchronization primitives. Our goal is to significantly reduce the number of messages required to maintain coherence, the number of cache misses taken by applications due to memory conflicts, and the overhead of interprocess synchronization. We propose to do this by allowing shared data to be maintained using the consistency or synchronization protocol best-suited to the way the programming is accessing the data. For example, data that is being accessed primarily by a single processor would likely be handled by a conventional write-invalidate protocol [2], while data being heavily shared by multiple processes, such as global counters or edge elements in finite differencing codes, would likely be handled using a delayed write-update protocol [5]. Similarly, locks will be handled using conventional distributed locking protocols, while more complex synchronization operations like barriers and reduction operators for vector sums will be handled using specialized protocols. By handling data with a flexible protocol that can be customized for its expected use, the number of cache misses and messages required to maintain consistency drop dramatically, as illustrated in the following section.

3 Performance Evaluation

The Avalanche multiprocessor design effort is a large ongoing project with many aspects, including the cache design and simulation effort discussed herein and a concurrent VSLI design effort involving the design of a stripped down version of the HP PA-RISC 7100 (as discussed in Section 2) and a

[1]Except in the case of the Cray, which does not cache shared data.

VLSI implementation of the CCU. Currently our performance results are based on a highly detailed architecture simulation system, but we will validate these results via an actual implementation of a 64-node prototype in due course.

3.1 MINT Multiprocessor Simulator

We used the MINT memory hierarchy simulator [25] running on Silicon Graphics and Hewlett-Packard workstations to perform our simulations. MINT simulates a collection of processors and provides support for spinlocks, semaphores, barriers, shared memory, and most Unix system calls. We augmented it to support message passing and multiple processes per node. MINT generates multiple streams of memory reference events, which we used to drive our detailed simulation model of the Avalanche multiprocessor. Depending on the number of processors and the complexity of the cache controllers being simulated, our simulation runs took between twenty minutes and five hours to complete.

3.2 Network Model

To accurately model network delays and contention, we have developed a very detailed, flit-by-flit model of the Myrinet fabric[7]. The Myrinet fabric is mesh-connected, with one crossbar at the core of each switching node. To ensure that the results of our architecture evaluation experiments are not excessively biased by the relatively high latency of the Myrinet interface, we also measured the performance of Avalanche for a network with one-tenth the latency of Myrinet ("fast Myrinet"). In this network model, we account for all sources of delay and contention within the network for each flit of data, including per-switching-node fall through times, link propagation delays, contention for the crossbar in switching nodes and for FIFOs at the input and output ports of both compute and switching nodes. With this model, we were able to perform very detailed measurements of the amount of contention in the interconnect. The parameters that we use are presented in Table 1 in terms of 10ns CPU clock cycles.

3.3 Memory Model

Figure 1 illustrates the high-level organization of Avalanche's memory hierarchy. The three major components are the cache controller, the directory controller, and the network controller. The cache controller is responsible for handling the local CPU's requests for data and cooperating with the directory controller to ensure that data in the local cache hierarchy is kept coherent. The directory controller maintains the memory state information associated the local physical memory and handles coherence requests sent by remote nodes, in cooperation with the cache controller. The

Network Characteristics		
Parameter	Myrinet	Fast Myrinet
Link Delay	12	1
Fall Through	38	4
Buffer size per stage	80	80
Topology	2*2 (4) switch nodes	same

Table 1 Parameters Used In Network Models (in 10ns CPU clock cycles)

network controller is responsible for handling incoming and outgoing interconnect traffic, including DMA and certain high level synchronization primitives. In particular, for incoming coherence messages, the network controller is responsible for forwarding them to the directory controller or the cache controller, as appropriate. Space constraints make it impossible to discuss all of the details of the simulation environment herein – a more detailed description of the Avalanche architecture can be found elsewhere[9].

We used the following model in our architecture simulations. We modeled a sixteen-node system, where each node was configured as illustrated in Figure 1. Each node contained a 256-kilobyte first level cache and no second level cache, which is consistent with how the HP PA-RISC 7100 processor is normally configured. This design philosophy will continue to be used in future HP products (e.g., the PA-8000, which is expected to ship in 1996). The use of a second level cache should have negligible impact on the work presented herein, as this work is primarily intended to reduce coherency misses, which are largely independent of a node's internal cache organization. In addition, we assume that each of the three control units can handle only one request at a time and model the contention that this design entails. For example, if the directory controller receives a remote data request from a remote node for data that resides in the local cache, it sends a request to the cache controller to invalidate that cache line. While the cache controller is performing this invalidation and before it forwards the invalidation message(s) to the network controller, memory requests from the CPU stall. Similarly, if the local CPU accesses a word of local memory that is not in the cache, the cache controller sends a request message to the local directory controller to ensure that coherence is maintained (i.e., it does not read the data from the local DRAM until it is assured that a remote node is not caching a dirty copy of the data). While the directory controller is handling this request, it will not handle additional requests. In addition to these protocol processing delays, we also measured the contention between the cache controller and the directory controller for access to the DRAM bus, the former for processing local requests and the latter for processing remote requests. Between each pair of the controllers is a pair of FIFOs that are used to store requests for some action (e.g., invalidate a cache line, send a message, or update a directory entry). When requests are pending in both of a controller's input FIFOs, it handles them in a round robin fashion. The operations performed by each controller depend on which coherence protocol is being used, as briefly described in the following section. Table 2 lists the delay characteristics that we used in our model on a preliminary hardware design. We based these times on the existing PA-RISC 7100 implementation and our estimate of the time to perform operations within the CCU.

3.4 Protocols Investigated

We evaluated the performance of four basic coherence protocols: (i) a sequentially consistent multiple reader, singler writer, write invalidate protocol (`sc-wi`), (ii) a no-replicate migratory protocol (`mig`), (iii) a release consistent [13] implementation of a conventional multiple reader, single writer, write invalidate protocol (`rc-wi`), and (iv) a release consistent multiple reader, multiple writer, write update protocol (`rc-wu`). We selected these four protocols because they covered a wide spectrum

---

[3] However, the cache controller remains busy for a second cycle updating state information. Therefore, if the processor performs a second memory request immediately after the first write, it will be delayed an extra cycle.

[3] This one cycle only includes the time to copy the 8-byte message header into the receiving controller's input FIFO, after which time the sending controller is free to process another request. The data associated with the message, if any, is located in a shared (three-ported) data buffer array used by all three controllers (not shown in Figure 1). Data is read/written from/to this buffer at a rate of 8 bytes per cycle.

Operation	Delay
Local read hit	1 cycle
Local write hit	1 cycle[2]
DRAM read setup time	6 cycles
DRAM write setup time	2 cycles
Time to transfer each subsequent word to/from DRAM	1 cycle
DRAM refresh (time between DRAM requests)	3 cycles
Enqueue a message in a FIFO between controllers	1 cycle[3]
Dequeue a message from a controller's input FIFO	1 cycle
Update directory entry	4 cycles

Table 2 Delay Characteristics

of options available to system designers. In all of our experiments, we simulated an implementation that used a conventional directory-based management scheme, with a fixed home node per cache block based on a function of the block's address. Due to space constraints, we have not included a detailed description of the protocols in this paper, but instead refer the interested reader to a more detailed technical report [9]. For each application program, we explored the potential of allowing software to specify the coherence protocol to be used to maintain shared data for an application by evaluating the performance of each individual protocol on the application. In addition, we explored the implication of allowing software to specify the coherence protocol of individual pages or cache lines by using an off-line algorithm to determine the optimal protocol for each block of data. The `opt` pseudo-protocol represents the performance achievable if the optimal protocol is used for each data block (cache line or page, depending on the simulation).

The `sc-wi` protocol represents a direct extension of a conventional bus-based write-invalidate consistency protocol to a directory-based implementation. 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. If the data is dirty in a remote cache, the home node sends a message to the owner, and the owner sends the dirty cache line back to the home node, which in turn forwards a copy of the data to the requesting node. If the block is read shared, the home sends invalidate messages to all other nodes that still have cached copies of the block, collects the invalidations, and forwards a message to the requesting node indicating that all of the nodes that had a copy of the data have now invalidated it. 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. If not, the home node sends a message to the current owner requesting an up to date copy of the data, which it forwards to the requesting node.

Cache blocks being kept consistent using the `mig` 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, otherwise it requests the data and forwards it to the requester. 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.

For the two release consistent protocols (`rc-wi` and `rc-wu`), we assume the presence of a *write state* buffer that contains a small number of entries. Each entry is associated with a local dirty cache line and is used to keep track of which words are dirty in that line. Write state buffer entries are allocated on demand when the local cache writes to a shared cache line. Unlike a conventional write buffer[17], which contains the modified data as well as its address, the write state buffer contains only an indication of what words have been modified. The modified data itself is stored in the cache. This state information is used to improve the performance of writes to shared data, albeit in different ways for each protocol.

The `rc-wi` protocol performs identically to the `sc-wi` protocol on reads, but the write state buffer improves write performance. When a processor writes to shared memory, it may continue executing as soon as an entry has been allocated in the write state buffer, without waiting to receive ownership from the home node. The entry cannot be flushed until the local node has received ownership of the cache line. In the mean time, reads to the dirty words can be satisfied from the local cache, and reads to other words in a dirty cache line can be performed if the line was present in the cache before the write occurred. Only if the write state buffer becomes full, which is infrequent, or the processor reaches a "release" point and the controller has not received ownership of the cache lines in the write state buffer, does the processor need to stall. This can significantly reduce the overhead of handling shared writes. 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 - a detailed explanation can be found elsewhere [13].

The `rc-wu` protocol uses the write state buffer in a different way. When a node writes to a word of shared data, it allocates an entry in the write buffer for the associated cache line and marks that word as dirty. When the processor reaches a release point or the number of entries in the write buffer exceeds some threshold (in this case, four out of the eight entries), the local cache controller flushes the dirty words to the home node. Until that point, the processor delays the sending of the update. The home node forwards the update message to other nodes with a copy of that cache line, which incorporate the changes on a word-by-word basis. In this way, multiple processors can simultaneously modify a single cache line as long as they do not modify the same words, which would represent a race condition and likely a bug in the program. The `rc-wu` exploits release consistency's flexibility by *buffering* writes to shared data, thereby mitigating the normal problem of write update protocols and excessive bandwidth requirements. Furthermore, the use of a write update protocol can significantly reduce the number of read misses that a write-invalidate protocol induces as a side effect of maintaining coherence when the degree of sharing is high[5]. For example, if processors A and B are both reading and writing data from a particular cache line, a write invalidate protocol will result in a large number of invalidations and subsequent read misses when the invalidated processor reloads the data that it needs. The invalidations are relatively cheap, because they can be pipelined, but the read misses can seriously degrade performance, because while the data is being fetched, the processor must either stall or context switch. Both `rc-wu` and `rc-wi` must perform memory consistency operations when the program arrives at release points, which can degrade performance if the application synchronizes frequently.

Finally, we also measured what we will refer to as the `opt` or optimal pseudo-protocol. In the previous experiments, we assumed that the CCU could support multiple coherence protocols, but that only a single coherence protocol was used by any given program. In Section 3.6 we show that the choice of coherence protocol has a large effect on performance for the different applications. We also explored the potential additional benefit that could be derived by allowing software, e.g.,the compiler or programmer, to specify to the CCU the base protocol that should be used for individual blocks of data, rather than for the entire program. This experiment measures the value of adding two additional protocol state bits per page table and TLB entry (for page-grained specifications) or cache line (for cache line grained specifications). We measure the performance of the `opt` pseudo-protocol by determining off-line which protocol induced the least cache overhead per data block, and using this optimal protocol for that block when calculating total cache stall and execution times. `opt` represents a near best case measurement of the potential value of the adding protocol bits because it assumes that software is able to perfectly specify in advance how each block of memory should be handled, although it does not measure the potential value of changing the choice of protocol dynamically during runtime nor of reorganizing the data layout to exploit the particular features of a given protocol. While it is probably not reasonable to assume that this performance is achievable in general, it provides us with some insight into the value of allowing software to specify the coherence protocol at a small grain.

3.5 Benchmark Programs

We used five programs from the SPLASH benchmark suite [22] in our study, `mp3d`, `water`, `barnes`, `LocusRoute`, and `cholesky`. Table 3 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 migratory 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 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.

Program	Input parameters
`mp3d`	20,000 particles, 10 time steps, `test.geom`
`water`	`LWI12`, 128 molecules, 6 time steps
`cholesky`	`bcsstk14`
`barnes`	`sample.in`
`locus`	`bnrE.grin`

Table 3 Programs and Problem Sizes Used in Experiments

3.6 Experimental Results

We simulated the performance of the five application programs running on a detailed model of an eight-processor Avalanche system. Figures 2, 4, and 6 are for the Myrinet interconnect, while Figures 3, 5, and 7 are for the "fast Myrinet" interconnect. To avoid cluttering the graphs with irrelevant data, we factored out non-shared memory references, which add negligible overhead due to the large cache size relative to the working set size.

Figures 2 and 3 show the total cache stall times for each of the protocols as a percentage of the conventional `sc-wi` protocol. The height of each vertical bar represents the relative number of cycles that the processor spends stalled waiting for memory requests to be satisfied. Note that the `mig` protocol graphs have been scaled down for `barnes`, `locus`, and `water` so that `mig`'s poor performance on these program did not overwhelm the other results. The performance of the individual coherence protocols varied dramatically from application to application. For the Myrinet interconnect (Figure 2), the `rc-wu` protocol performed best for every application except `mp3d`, which is known to have mostly migratory data. `rc-wu` performed particularly well for `barnes` and `locus`, removing over 60% of the cache stall time compared to the conventional `sc-wi` protocol and over 40% compared to `rc-wi` protocol used as the base protocol in FLASH[18]. For the faster interconnect (Figure 3), the results were more varied. `rc-wu` continues to perform very well for `barnes` and `locus`, while `mig` continues to perform best for `mp3d`, but with the use of a faster interconnect, FLASH's `rc-wi` protocol performs best for `cholesky` and `water`. The large variance in each application between the most efficient protocol and the other protocols and the fact that the protocol that performs best differs from application to application is strong evidence that Avalanche's ability to support multiple coherence protocols will result in a significant performance payoff.

Each bar is subdivided into the individual components that account for the overall cache stall time. READ represents the overhead of read misses, which accounts for the majority of the cache stall time for the write-invalidate protocols (`sc-wi`, `mig`, and `rc-wi`). The large reduction in read miss penalties accounts for `rc-wu`'s significant performance benefits for `barnes` and `locus`, which contain a high degree of write sharing. WRITE represents the time spent stalled due to writes, which comes from a number of sources depending on the protocol, including the time to acquire ownership and the time to free up a write-state entry. The write-state buffer allows WRITE times to be largely masked for the release consistent protocols, except in `barnes` and `water`, where the WRITE time represents 20% of the cache stall time even for the release consistent protocols. The reason that the WRITE stall time is significant in these two applications is that they perform a large number of writes to shared data between synchronization points, which overwhelms the small (eight-entry) write-state buffer used in the simulations. Finally, SYNCH represents the time spent stalled at synchronization points while flushing the write-state buffer. This delay component was only significant for the two release consistent protocols, `rc-wi` and `rc-wu`, where it represents the time spent flushing the write-state buffer entries (acquiring ownership or propagating updates for `rc-wi` and `rc-wu` respectively).

Tables 4 and 5 present the average read, write, and synchronization times (measured in CPU cycles) for the various protocols on the different applications. These results include local reads and writes, which are almost always satisfied in a single cycle. Ideally the average read and write times would be one cycle, and the average synchronization time would be zero. However, the impact of coherence can dramatically increase the average memory access times. `mp3d`'s reputation as a poorly structured program is borne out by the fact that its average read cycle time varies from 7 to 23 cycles. The reason for `rc-wu`'s good performance in most of the applications is apparent - its average read cycle time is always the lowest of the four protocols measured. Since read

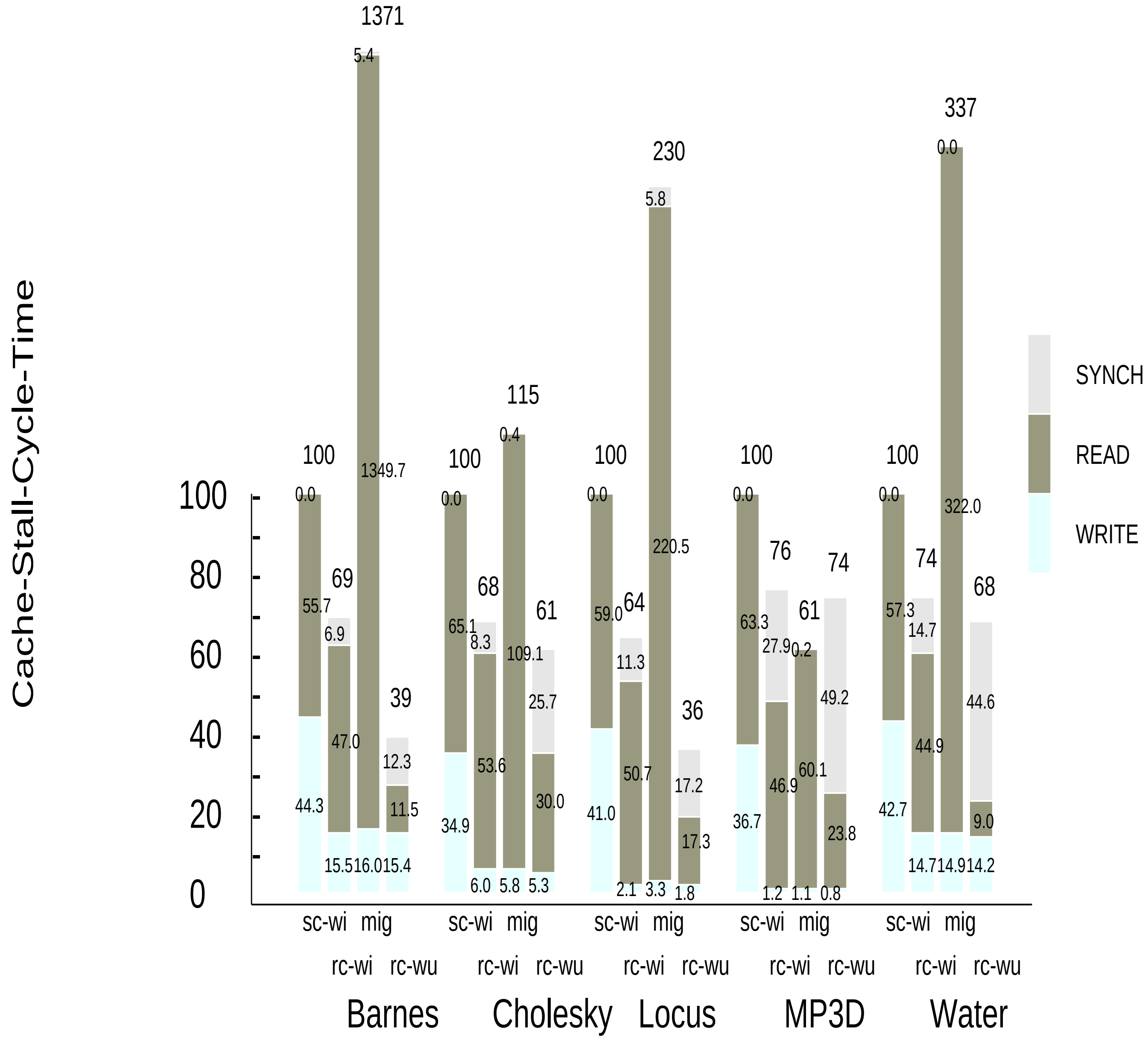

Figure 2 Cache stall time (Myrinet)

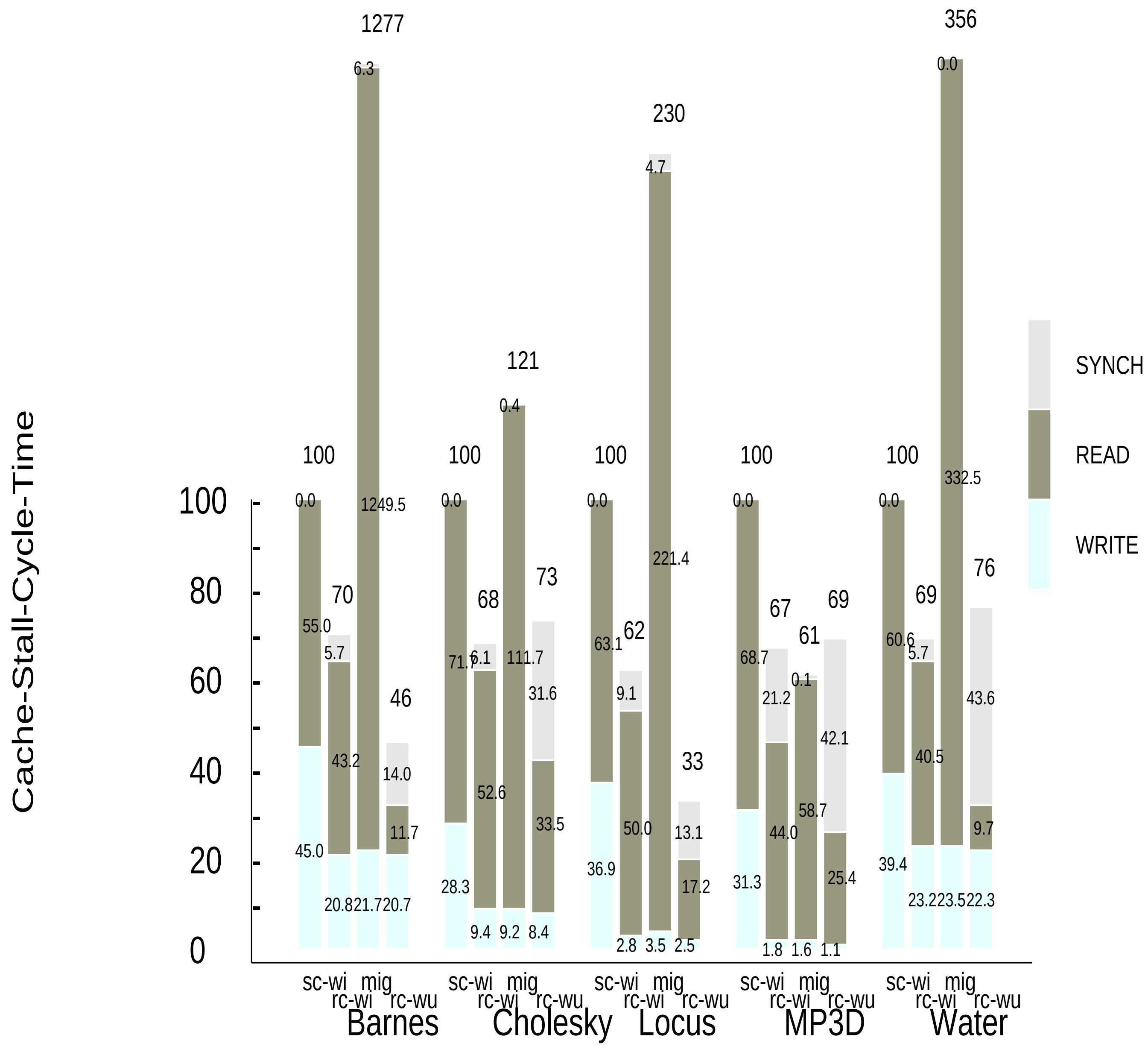

Figure 3 Cache stall time (10*Myrinet)

misses account for the largest component of the overall cache stall time for most applications, this is an important benefit. However, the tradeoff is `rc-wu`'s high synchronization time, when it is required to flush the write-state buffer by performing or completing pending update operations. Thus, for programs with very frequent synchronization, `rc-wu`'s good read miss performance can be overwhelmed by its high overhead at synchronization points.

Figures 4 and 5 show the overall execution times for each of the protocols as a percentage of the conventional `sc-wi` protocol, which follow the same trends observed above. Overall, Avalanche's support for multiple coherency protocols shows a clear improvement over conventional designs that employ the `sc-wi` protocol, reducing the overall execution times by as much as 30%. In addition, the use of a write state buffer improves Avalanche's performance compared to even a release-consistent write-invalidate protocol such as that employed in FLASH (`rc-wi`) - reducing execution time by as much as 15% (for `barnes`).

This improvement in performance did not come without some tradeoff. Figures 6 and 7 show the bandwidth consumed by each of the protocols as a percentage of the conventional `sc-wi` protocol. For the most part, they follow the same trends as before with the exception that the `rc-wu` protocol tends to consume more bandwidth than the other protocols despite its good performance in terms of stall cycles. For the programs that we examined, the bandwidth requirements were a small fraction of the bandwidth provided by the Myrinet interconnect, so it is not an issue. However, for applications with higher bandwidth requirements or lower bandwidth interconnects, this might become a problem.

The previous results assumed that the cache controller used the same consistency protocol for all of the shared data of a given program. Table 6 presents an approximation of the performance that can be obtained by adding extra state bit per page table entry to give software control over the coherency protocol used for each page of shared data and using the protocol best suited to the

Average read cycle time					
Protocol	`mp3d`	`water`	`barnes`	`cholesky`	`locusroute`
`rc-wi`	17.8	1.7	2.2	3.9	9.4
`rc-wu`	9.5	1.1	1.3	2.7	4.0
`mig`	22.5	6.1	34.2	6.7	23.1
`sc-wi`	23.6	1.9	2.4	4.5	10.8
Average write cycle time					
Protocol	`mp3d`	`water`	`barnes`	`cholesky`	`locusroute`
`rc-wi`	2.2	1.8	1.6	2.4	2.0
`rc-wu`	1.8	1.8	1.6	2.2	1.9
`mig`	2.1	1.8	1.6	2.3	2.1
`sc-wi`	30.3	3.1	2.7	8.4	16.0
Average synch cycle time					
Protocol	`mp3d`	`water`	`barnes`	`cholesky`	`locusroute`
`rc-wi`	370.3	243.7	464.8	425.5	369.5
`rc-wu`	652.5	738.2	831.7	1321.8	562.9
`mig`	2.0	0.1	368.0	20.5	189.0
`sc-wi`	n/a	n/a	n/a	n/a	n/a

Table 4 Average operation cycle time (Myrinet)

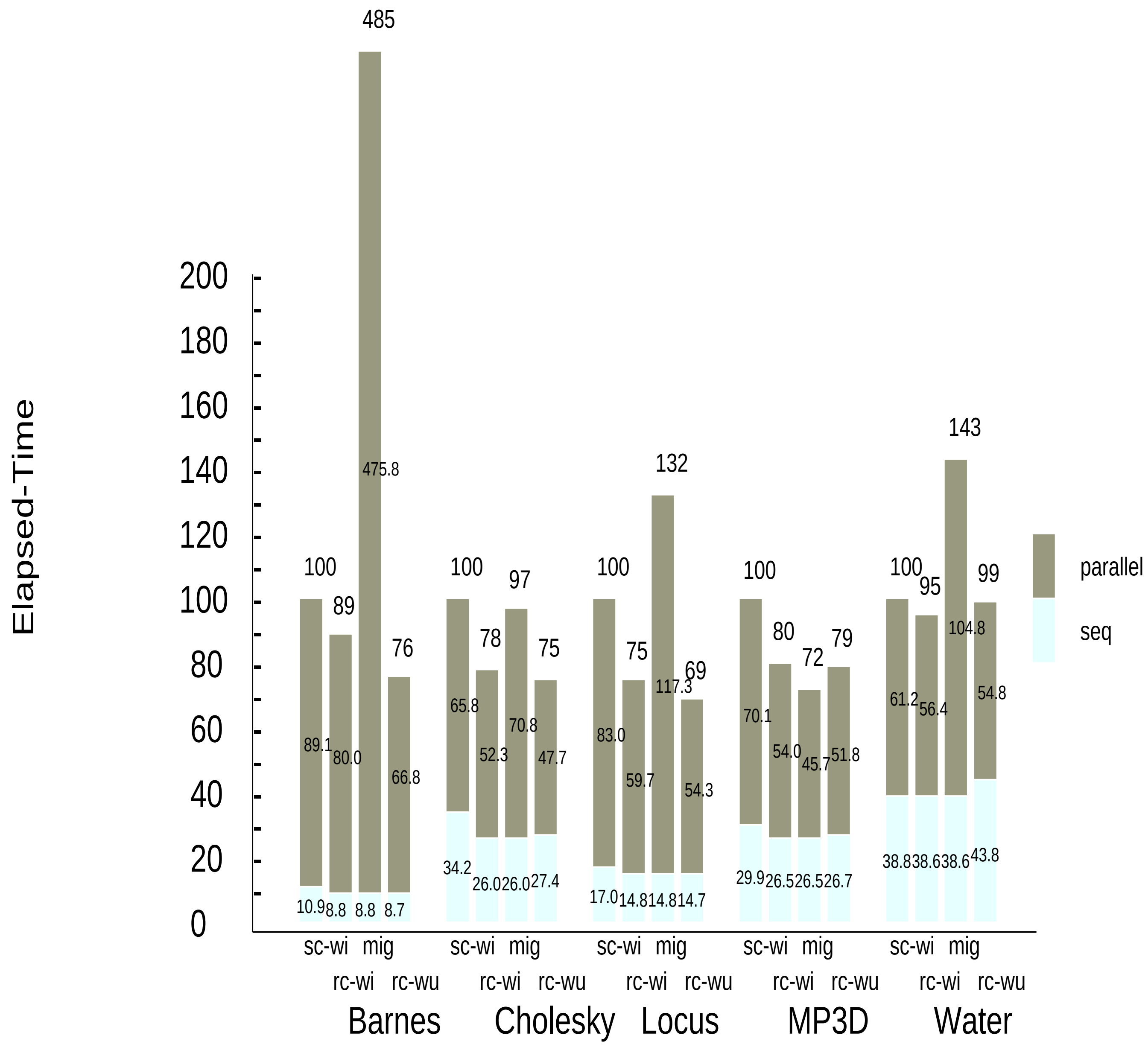

Figure 4 Execution time (Myrinet)

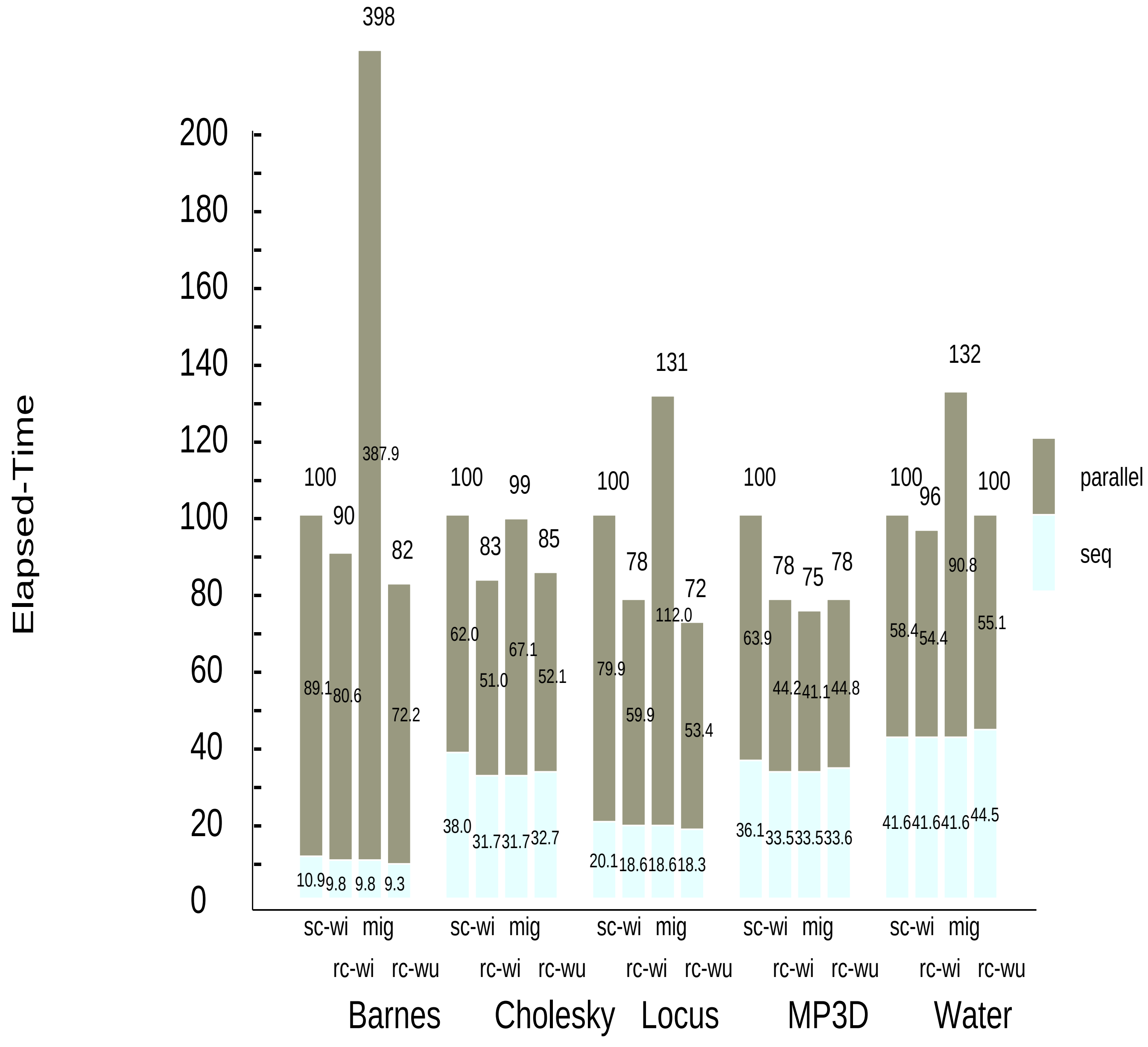

Figure 5 Execution time (10*Myrinet)

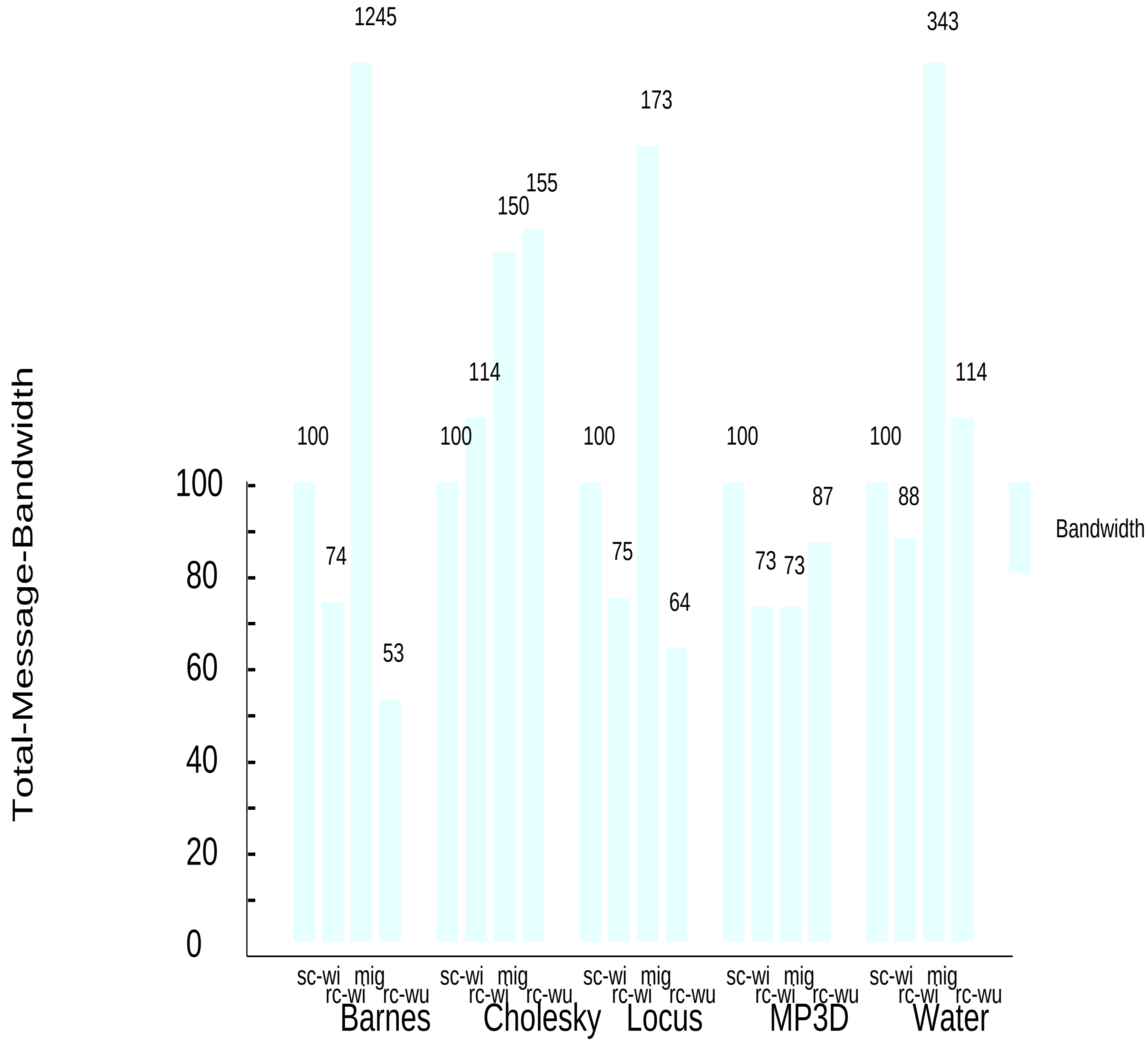

Figure 6 Bandwidth (Myrinet)

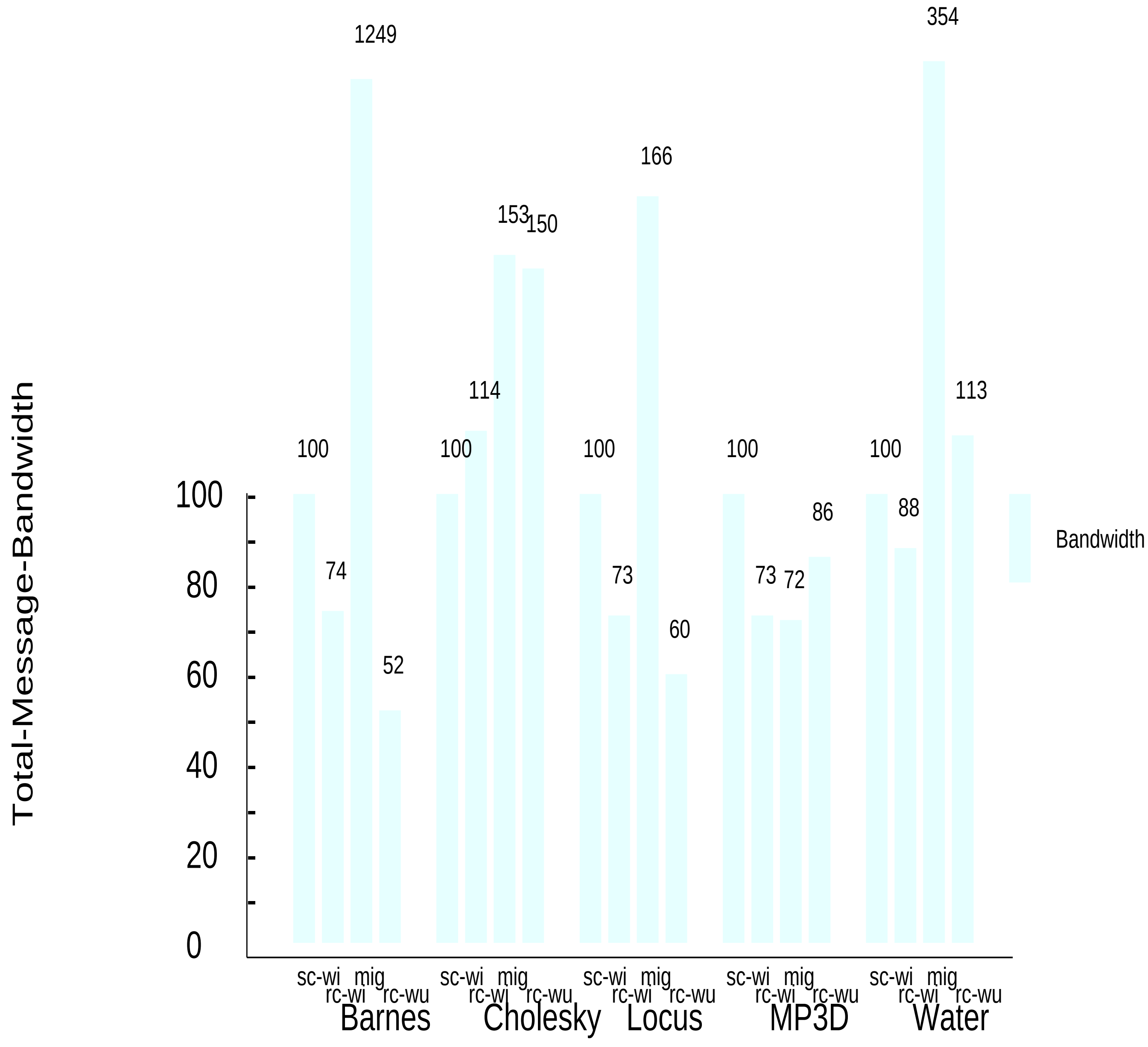

Figure 7 Bandwidth (10*Myrinet)

Average read cycle time					
Protocol	mp3d	water	barnes	cholesky	locusroute
rc-wi	12.1	1.4	1.8	2.8	7.1
rc-wu	7.4	1.1	1.2	2.2	3.2
mig	15.8	4.4	23.8	4.7	23.9
sc-wi	18.3	1.6	2.0	3.4	8.8
Average write cycle time					
Protocol	mp3d	water	barnes	cholesky	locusroute
rc-wi	2.2	1.8	1.6	2.4	2.0
rc-wu	1.8	1.8	1.6	2.2	1.9
mig	2.1	1.8	1.6	2.3	2.1
sc-wi	18.6	2.3	2.3	5.1	11.0
Average synch cycle time					
Protocol	mp3d	water	barnes	cholesky	locusroute
rc-wi	197.2	60.4	285.0	199.4	218.0
rc-wu	392.1	460.3	701.6	1032.7	315.7
mig	1.4	0.1	316.5	14.3	114.0
sc-wi	n/a	n/a	n/a	n/a	n/a

Table 5 Average operation cycle time (fast Myrinet)

way each page is used. These results are only approximate in that they do not accurately account for synchronization and secondary effects, but they are sufficient to provide an estimate of the value of providing this extra hardware. As expected, the impact is limited, but for `cholesky` and `locus` even simple page-level support can reduce the cache stall time by 4-6%. Table 7 presents the same information for the situation where the software is given control at the cache-line level through the use of extra state bits per cache line. The improvements are somewhat better (9-13%). However, these results are conservative in that the five programs studied each have a single dominant data structure that accounts for most of the shared memory accesses performed by the program. Thus, it is not surprising that fine tuning the cache behavior for different parts of the shared address space does not have a large impact on performance. Larger programs with more diverse uses of shared data should exhibit larger improvements. Overall, these results indicate that further study is worthwhile.

The results presented in this section provide strong evidence that the flexible memory controller being designed for Avalanche can lead to significant performance improvements, even for relatively fine-grained applications such as the ones that we studied. We are continuing to evaluate our

Protocol	mp3d	water	barnes	cholesky	locusroute
Myrinet	—	—	—	4%	6%
"Fast" Myrinet	—	—	—	5%	4%

Table 6 Reduction in memory overhead using "optimal" protocol (page granularity)

Protocol	mp3d	water	barnes	cholesky	locusroute
Myrinet	1%	—	—	10%	12%
"Fast" Myrinet	2%	—	1%	13%	9%

Table 7 Reduction in memory overhead using "optimal" protocol (cache line granularity)

design and are in the process of adding more applications to our application benchmark suite and modifying our simulation environment to allow larger working sets to be evaluated.

4 Related Work

There are a number of ongoing efforts whose goal is to design a scalable high-performance multiprocessor. Our approach differs from the approaches taken in these systems in a number of important aspects, as described below.

The Stanford DASH multiprocessor [19] uses a novel directory-based cache design to interconnect a collection of 4-processor SGI boards based on the MIPS 3000 RISC processor. The Convex Exemplar employs a similar design based around the HP7100 PA-RISC. Avalanche will employ a similar directory-based cache design. However, our cache controller will be tightly integrated with the communication controller, support a variety of consistency protocols and synchronization primitives, exploit a limited degree of context sensitivity, and allow software to tune the cache controller's behavior. A second generation DASH multiprocessor is being developed that introduces a limited amount of processing power and state at the distributed directories to add flexibility to the consistency implementation. This machine, called FLASH [18], is currently being designed to support both DASH-like shared memory and efficient message passing. However, their plans for exploiting the flexibility of their controller's operation have not been revealed.

The MIT Alewife machine [1, 10] also uses a directory-based cache design that supports both low latency message passing and shared memory based on an invalidation-based consistency protocol. Alewife incorporates a limited amount of flexibility by allowing the controller to invoke specialized low-level software trap handlers to handle uncommon consistency operations, but currently the Alewife designers are only planning to use this capability to support an arbitrary number of "replica" pointers.

The MIT M-Machine work [20] contains a context cache similar to previous designs such as the HP Mayfly system [12]. This context cache provides dynamic binding of variable names to register contents to permit rapid task switching and promote the interesting processor coupling mechanism of the M-machine. However, it does not provide the tight integration of communication fabric and protocol into a realistic memory hierarchy, nor does it exploit context sensitivity to tune its behavior.

The Motorola and MIT *T machine [4] has many interesting components that offer excellent support to exploit dataflow style parallelism. The *T architecture provides tight coupling between the processor registers and the interconnect fabric, but isolates the memory hierarchy by placing the CPU between the interconnect fabric and the memory. The result is that the CPU must mediate message and/or DSM communication events. The level of primary processor cycle stealing that this implies will seriously impede scalability on conventional style applications based on DSM or message passing that do not exploit the *T's powerful support for data flow languages.

Like Avalanche, the user level shared memory in the Tempest and Typhoon systems [21] will support cooperation between software and hardware to implement both scalable shared memory and message passing abstractions. Like the Alewife system, will support low level interaction between software and hardware to provide flexibility. As such, it currently requires extensive program modification or user effort to achieve scalable performance, although the designers are working on a number of compilation and performance debugging tools to help automate this process. The tradeoffs between the software and hardware approaches are being studied.

The SHRIMP Multicomputer [6] employs a custom designed network interface to provide both shared memory and low-latency message passing. A virtual memory-mapped interface provides a constrained form of shared memory in which a process can map in pages that are physically located on another node. Since the network controller is not tightly coupled with the processor, the cache must be put into write-through mode so that stores to memory can be snooped by the network interface, which results in added bus traffic between the cache and main memory. In addition, incoming messages are placed into main memory via a DMA engine, using invalidation to maintain consistency, which results in cache misses that would not occur if the network controller was more tightly coupled with the memory system.

The Thinking Machines CM-5 [23] did not directly support DSM or a multilevel external memory hierarchy, and as such the excellent communication fabric of the CM-5 is not well integrated into the memory architecture. Thus, the on-chip cache miss penalties discussed earlier have proven problematic in terms of achieving a reasonable percentage of the impressive peak performance of the CM-5 on real applications. Another commercial scalable supercomputer of interest is the Intel Paragon [16]. The interconnect is a high performance mesh routing device. The fabric does not support direct DMA into the Paragon's memory hierarchy but utilizes a second i860XP CPU for this purpose on each processing element. In addition, the interconnect is not tightly integrated into the memory hierarchy, so messages are only placed into main memory rather than the processor cache.

5 Conclusions

In the Avalanche project at the University of Utah, our goal is to build a 64-node multiprocessor designed to reduce the end-to-end communication latency of both shared memory and message passing programs. As part of our design efforts, we are evaluating the potential performance benefits and implementation complexity of providing a means for software to guide the cache controller's behavior. In this paper, we have discussed one particular aspect of this effort - support for multiple hardware cache coherency protocols. Using a detailed architecture simulation of Avalanche, we have found that support for multiple consistency protocols can reduce the time parallel applications spend stalled on memory operations by up to 66% and overall execution time by up to 31%. Most of this reduction in memory stall time is due to a novel release-consistent multiple-writer write-update protocol implemented using a *write state buffer*.

However, much work remains to be done before the Avalanche prototype is constructed. We are currently working with Hewlett-Packard to create a version of the PA-RISC 7100 which exports an interface for a new CCU which will be fabricated as a separate chip. We also are improving our simulation environment, testing the high-level CCU design on more and larger applications (both shared memory as reported upon in this paper and a variety of message passing programs), developing a set of protocol verification tools to reduce the debugging time needed to implement the CCU, and considering compiler-based techniques for fully exploiting Avalanche's flexibility. In summary, although the challenges that face us are considerable, we believe that the Avalanche design outlined here will result in the development of a memory architecture for commercial microprocessors that will significantly improve their performance utility in scalable multiprocessor configurations.

References

[1] A. Agarwal and D. Chaiken et al. The MIT Alewife Machine: A large-scale distributed-memory multiprocessor. Technical Report Technical Memp 454, MIT/LCS, 1991.

[2] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[3] David Beazley, 1994. Member of 1993 Gordon Bell Prize winning team, personal communication.

[4] M. J. Beckerle. An Overview of the START (*T) Computer System. MCRC Technical Report MCRC-TR-28, Motorola Cambridge Research Center, 1992.

[5] 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.

[6] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[7] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su. Myrinet – A gigabit-per-second local-area network. *IEEE MICRO*, 15(February):29–36, February 1995.

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

[9] J.B. Carter and R. Kuramkote. Avalanche: Cache and DSM protocol design. Technical report, University of Utah, April 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] Cray Research, Inc. *CRAY T3D System Architecture Overview*, hr-04033 edition, September 1993.

[12] A. L. Davis. Mayfly: A General-Purpose, Scalable, Parallel Processing Architecture. *Lisp and Symbolic Computation*, 5(1/2):7–47, May 1992.

[13] 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.

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

[15] M. Heinrich and J. Kuskin et al. The performance impact of flexibility in the Stanford FLASH multiprocessor. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[16] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[17] N.P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, May 1993.

[18] 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.

[19] 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.

[20] P. Nuth and W. J. Dally. A Mechanism for Efficient Context Switching. In *Proceedings of the IEEE International Conference on Computer Design*, pages 301–304, 1991.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.