Avalanche: A Communication and Memory Architecture for Scalable Parallel Computing

John B. Carter, Al Davis, Ravindra Kuramkote,
Chen-Chi Kuo, Leigh B. Stoller, Mark Swanson

UUCS-95-022

Computer Systems Laboratory
University of Utah

Abstract

As the gap between processor and memory speeds widens, system designers will inevitably incorporate increasingly deep memory hierarchies to maintain the balance between processor and memory system performance. At the same time, most communication subsystems are permitted access only to main memory and not a processor's top level cache. As memory latencies increase, this lack of integration between the memory and communication systems will seriously impede interprocessor communication performance and limit effective scalability. In the Avalanche project we are redesigning the memory architecture of a commercial RISC multiprocessor, the HP PA-RISC 7100, to include a new multi-level context sensitive cache that is tightly coupled to the communication fabric. The primary goal of Avalanche's integrated *cache and communication controller* is attacking end to end communication latency in all of its forms. This includes cache misses induced by excessive invalidations and reloading of shared data by write-invalidate coherence protocols and cache misses induced by depositing incoming message data in main memory and faulting it into the cache. An execution-driven simulation study of Avalanche's architecture indicates that it can reduce cache stalls by 5-60% and overall execution times by 10-28%.

Avalanche: A Communication and Memory Architecture for Scalable Parallel Computing

John B. Carter, Al Davis, Ravindra Kuramkote,
Chen-Chi Kuo, Leigh B. Stoller, Mark Swanson

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 would seem to be attractive options for assembly into cost-effective parallel machines. However, the communication delay between tasks on different processors, regardless of whether they are using messages or shared memory, rapidly becomes the bottleneck. At best, most I/O and communication subsystems are only permitted access to main memory. Thus, effective end to end message passing latencies are large and getting larger due to the cache misses required to load received data into the highest level of the cache where it can be used. Similarly, existing shared memory implementations use hardwired coherence protocols that induce a large number of cache misses as a side effect of keeping data coherent. As deepening memory hierarchies cause main memory latencies to increase from 10's to 100's of cycles, the avoidable cache misses caused by a lack of integration between the communication and memory systems and inflexible cache controllers will seriously impede communication performance and scalability.

To achieve effective scalability, all sources of interprocess communication latency must be attacked. In doing so, it is important to measure the latency of communication events for the *total* latency path. For message passing programs, latency includes software protocol overheads, the time required to inject a message into the communication fabric, the propagation delay, and the time spent handling the cache misses required to load the data from the recipient's main memory to its highest level cache. In shared memory programs, 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 the time spent servicing cache misses for data that has been invalidated as part of the coherence mechanism. 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[27] and Cray T3D[12] despite their powerful communication fabrics[3]. Even when the interconnection fabric is capable of very high speed communication, the effects of the memory hierarchy and the parasitic influence of other overheads become the dominant latency components.

For an architecture to scale effectively into the tera- and peta-op range, high latency cache misses must be avoided. This implies the need for a communication architecture that is consistently

This paper is a draft that was submitted to ISCA '95. Do not distribute without permission.

integrated into the highest levels of the memory hierarchy and that can adapt to the current state of the processor. However, always injecting data directly into the highest level cache can potentially reduce performance by displacing useful cache lines of the currently active task on the receiving node or contending with the receiving processor for access to the cache. It is therefore necessary to recognize the task activity state of the receiving processing element in order to decide the proper place for the message. For applications with a significant amount of shared data, 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. Thus, consistency protocols and cache controller designs that reduce the frequency of cache misses must be developed.

In the Avalanche project, we are designing and evaluating a novel memory and communication architecture that addresses these problems. 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). Specifically, we are splitting the processing and memory management subsystems into two components to create a memory-less version of the 7100 with the proper control signals and state information exported off chip. At the core of our system, we are designing a flexible communication and cache controller unit (CCU) so that system components outside the processor, in particular the communication fabric, are given first-class access to the complete memory system. A centerpiece of the CCU will be its ability to exploit processor context information to support multiple cache consistency protocols, avoid *conflict* misses between active tasks and incoming data, and dynamically prefetch data to the appropriate level of the memory hierarchy depending on the *speculation level* of the prefetch. An execution-driven simulation study of Avalanche's architecture indicates that it can reduce cache stalls by 5-60% and overall execution times by 10-28%.

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.

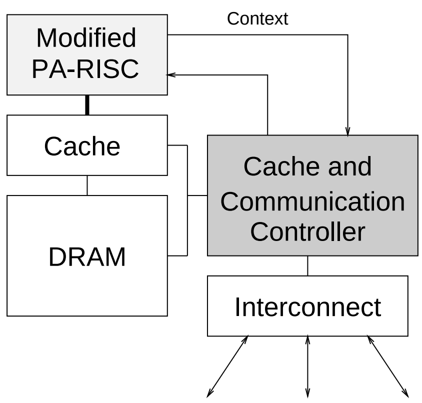

Figure 1 Overview of Avalanche Memory and Communication Architecture

2 Avalanche Design

2.1 Basic Architecture

The goal of the Avalanche project is to develop a communication and memory architecture that supports significantly higher *effective* scalability than existing multiprocessors. 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 provisions for exploiting processor context information or software guidance.

Figure 1 illustrates the high level view of the Avalanche architecture. Our design efforts will be centered on the two shaded regions, the cache and communication controller and the CPU itself. The majority of our design effort will involve the design of the cache and communication controller (CCU). The goals of the design CCU include:

- supporting the ability for data to be transmitted from and received by any level of the memory hierarchy, from L1 cache to main memory,
- allowing the processor to guide the CCU's behavior through the use of context information and tag bits, and
- minimizing the impact of conflict misses induced by interference between the local processor's memory accesses, speculative data prefetching, and the memory coherence mechanism.

A major issue that the CCU must be able to address is deciding into what level of the memory hierarchy incoming data should be placed. In the case where the local processor is blocked awaiting data, e.g., a cache line loaded as a result of a read miss in the absence of multithreading, the data obviously should be placed in the L1 cache. However, in all other cases, a decision must be made whether to place the data high in the memory hierarchy (meaning close to the processor), which reduces the latency of access to the received data but increases the latency of access to the displaced data, or low in the memory hierarchy, which avoids the displacement cost but increases the latency of accessing the received data whenever it is eventually accessed. We call this problem the *injection problem*, and discuss several techniques that we are exploring to address the problem throughout the remainder of this paper.

One significant difference between Avalanche and related research projects [21, 20, 1, 24] is that we are not treating the CPU as an unmodifiable black box. The HP PA-RISC 7100 contains no on-chip cache but does contain the cache controller logic. The interface between the cache controller and the rest of the CPU is relatively straightforward. Hence, in conjunction with Hewlett-Packard, we are designing a version of the HP PA-RISC 7100 chip with the cache controller moved off chip. This approach is being taken for experimental expediency since our focus is on a redesign of the cache controller rather than the CPU core. A minor change to the control path of the CPU will be required to export the necessary context state information. Additionally, a minor modification to the existing pipeline stall blocks will be necessary to maintain proper pipeline synchronization with the new requirements of the CPU. The functionality of the 7100's cache control logic will be subsumed by the new Avalanche CCU.

We do not anticipate modifying the core logic of the 7100. Although modifying a complex chip such as the 7100 is not without its risks, the intent of this effort is to permit us to both explore a wide set of design options and determine what small set of modifications are cost effective for commodity microprocessor vendors should they desire to make their memory architectures support highly scalable multiprocessing as well as their existing core uniprocessor markets.

The design of high performance parallel systems has been stylistically diverse. Many designers have touted the benefits of the message passing model while others prefer the shared memory abstraction. Our approach is to accept the application community's requirements and pursue both message passing and distributed shared memory (DSM). At the architecture level the difference may not be as significant as it is at the applications level. Both require efficient low-latency communication. The difference is in how a communication event is initiated. Because the architectural requirements of efficient message passing and efficient DSM are so similar, we believe that it is possible to support both communication models with the proper memory architecture. Although many of Avalanche's performance optimizations will improve the performance of both message passing and shared memory programs, some aspects of the CCU design are specific to one or the other type of program. The following two subsections describe these features.

2.2 Support for Message Passing

Our previous work on high speed networking [26] made it clear that for high bandwidth interconnects the software protocol overhead and the time spent handling the cache misses required to load the data from the recipient's main memory to its highest level cache were major sources of communication latency. We are attacking these two problems (i) by developing a very low overhead sender-based communication protocol and (ii) by allowing the CCU to transmit data from and inject data into any level of the memory hierarchy.

Two important characteristics of sender-based protocols are that they are connection oriented and that both the sender and the receiver reserve portions of their address space as buffers for a given connection[26]. Based on its knowledge of the state of that buffer space, a sender can transmit a message to the receiver with the *certainty* that the message will be received into a known location in the receiver's memory. Maintenance of this "knowledge" of buffer state is the responsibility of higher levels of the protocols. This overhead is only incurred on initial send-receive connection pair setup and is therefore amortized over subsequent communications for that connection. For example, in an RPC configuration, the reply to the client could be defined to mean that the request buffer in the server is again available for the sender to use. A major benefit is that a message can always be copied directly from the sender's address space into the network interface, and from the network interface directly into the receiver's address space. This mechanism avoids intermediate copies of the data, which are a major source of inefficiency in many existing protocols[13, 11], and is a prerequisite for successfully attacking the injection problem.

On a 100 MHz HP7100 processor with an external I/O controller, our current protocol implementation takes 111 CPU cycles to write a DMA descriptor block using programmed I/O. It then takes 300 cycles to DMA the 32 word packet body into the network controller. Another 400 cycles of DMA penalty is incurred on the receiving side when the packet body is written to main memory. These high DMA penalties are an artifact of the location of the I/O controller on a low-bandwidth, high latency I/O bus. In addition, the receiving side incurs a cache-miss penalty of 40 cycles per cache line for each of the 4 lines in the packet body. Thus, the total per packet latency is 971 cycles plus interconnect fabric delay. While this number is a huge improvement over standard protocols such as UDP and TCP/IP, it is also best case and can easily expand to three or four times this level under realistic processor loads. Even this best case of 971 cycles dominates the propagation delay of the high-speed interconnects in current multiprocessors [18, 27], which is a strong indication that as much effort needs to be placed in improving the performance of the memory system and the network controller's access to the memory as is being spent developing higher bandwidth and lower latency interconnects. This imbalance becomes an even more dominant factor as message size grows. Specifically, with few exceptions[27], conventional memory architectures require that messages always be transferred to or from main memory and that the cache be flushed as appropriate to maintain consistency[18]. Unfortunately this restriction guarantees that the receiver will incur cache misses for the entire message body. Substantial improvement can be made by more tightly coupling the communication fabric and protocol with the memory system and the context of the processor. This improvement will increase as miss penalties grow due to the deepening memory hierarchies required to balance CPU performance improvements.

The Avalanche CCU will provide us with the ability to place incoming data in any level of the memory hierarchy, as illustrated in Figure 2[1]. The CCU will incorporate a protocol processing element (PPE) to support the DMA requirements and some of the protocol duties. Unlike the protocol processor in the FLASH multiprocessor[20], Avalanche's protocol processor will only perform a very limited number of built-in operations – it is not a general purpose processor. To improve performance, the key problem that will need to be solved is how to dynamically determine which level of the memory hierarchy should receive the incoming data. When the processing element is lightly loaded or is waiting for a particular message, direct delivery into the highest level cache is the proper strategy. In cases where the processing element is heavily loaded with multiple runnable threads on the run list, then it is unlikely that delivery to the highest level cache will be the proper strategy since this delivery will cause conflict induced misses in the cache by the active or soon to be active contexts[2]. A study by Pakin et al. showed that the choice of where to inject incoming data can have a tremendous impact on overall performance of an application[23]. Always injecting data to main memory, as in the Intel Paragon[18], results in a message latency too high for very fine grained applications, and the effect is getting worse as processor speeds increase. On the other hand, always injecting data to the cache, as in the CM-5[27], displaced so much active data that the overall cache miss rate of the applications increased 85-290%. Hence the appropriate level in the memory hierarchy for message placement will critically depend on the current context of the processing element.

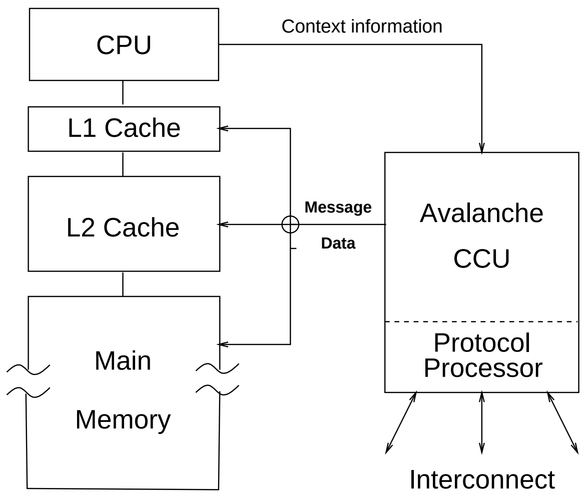

Figure 2 Message Passing Support in Avalanche

[1]The number of levels of cache in Avalanche has not been determined. Two levels are shown for purposes of illustration.

[2]We envision the need to support multiple tasks on each node to permit communication and computation to be overlapped and to reduce the probability of an idle processing element.

We are employing a number of techniques to make this decision. Our software protocol will be extended "upward" to the user level to allow the program to communicate compiler or programmer-generated knowledge of data use patterns to the lower levels of the protocol. The protocol will be extended "downward" to the network interface to provide information to the interface to enable it to make more intelligent decisions about placement of data into the memory hierarchy. We currently envision providing incoming connections with context information such as the process identifier of the receiving process, and limited scheduling information such as context identifier of the currently running process, newly scheduled process identifiers, etc. The PPE will be informed by the CPU on a context switch, which can be used to determine if the incoming data is destined for the executing process. Similarly, state will be added to the CCU to determine the "heat" of the cache, similar to the way in which the CAML buffer detects the "heat" of a particular set of pages in a direct-mapped cache[6]. We also plan to support more explicit software control of data placement in the form of directives in the incoming connection descriptors specifying message placement within the hierarchy.

2.3 Support for Shared Memory

Spurred by scalable shared memory architectures developed in academia [1, 21], 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[3] 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, 16, 28]. 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 *programmable multiprocessor cache controllers* that can execute a variety of caching protocols [9, 30], support multiple communication models [10, 17], or accept guidance from software [20, 24]. Programmable controllers would seem at first glance to be an ideal combination of software's greater flexibility and hardware's greater speed. As such, we are investigating CCU options which will implement a variety of caching protocols, support both shared memory and message passing efficiently, accept guidance from software to tune its behavior, and 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 individual data items (at page or cache line granularity) to be maintained using the consistency or synchronization protocol best-suited to the way the data is being used. 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 could be handled using conventional distributed locking protocols, while more complex synchronization operations

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

like barriers and reduction operators for vector sums could be handled using specialized protocols. By handling data with a flexible protocol that can be customized for its expected use, we expect the number of cache misses and messages required to maintain consistency to drop dramatically.

By reducing the amount of communication required to maintain coherence, multiprocessor designers can either use a commodity interconnect[8] and achieve performance equal to that of a static controller and a fast special purpose interconnect like that found in the CM-5, or use the faster interconnect to support more processors. 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. We performed a detailed execution-driven simulation study of this question, and present the results in Section 3.6.

2.4 Potential Pitfalls

The Avalanche project involves a number of risks that result from the inherent complexity in the approach. Modifying a modern commercial microprocessor that is tuned for performance to integrate a new subsystem as critical as the cache controller may result in reduced performance. However, the absolute speed of our prototype design is not the central issue since our desire is to investigate memory organizations that will impact the multiprocessor performance capability of future designs. These future designs clearly will integrate the design of the CPU and the CCU on the same chip. Hence our intent is to create a CCU design that is consistent with the commercial microprocessor core strategy but in its initial implementation may not be perfectly balanced for performance.

Providing flexibility in hardware often incurs a performance penalty. For mainline microprocessors, performance is everything. Our focus is to determine which protocol and message injection options are cost-effective. While this paper is an early status report of work in progress, our preliminary results indicate that for a relatively small increase in the complexity of the CCU, the performance of certain applications can be enhanced significantly. Further study is required to quantify both the scope of this advantage and the exact cost increment of the CCU.

While there are risks in this approach, we feel that the payoff is significant. Multiprocessor performance scalability utilizing the highly cost-effective workstation processor technology is an important enough goal to justify these risks. Fortunately Hewlett-Packard is an active partner in this effort. We would have little chance of succeeding if this were not the case.

3 Performance Evaluation

Avalanche is a large ongoing project with many aspects, as outlined in the previous section. Thus, for the purposes of this paper we will concentrate on one aspect of how Avalanche's novel communication and memory architecture impacts performance. Specifically, we report the results of a detailed architecture simulation study in which we explored the impact of using a number of coherence protocols, both individually and in an "optimal" combination.

3.1 MINT Multiprocessor Simulator

We used the MINT memory hierarchy simulator [29] 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 two system simulator models, one for message passing programs and one for shared memory programs. 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[8]. 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. 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. The parameters that we use are presented in Table 3.2 in terms of 10ns CPU clock cycles.

With this model, we were able to perform very detailed measurements of the amount of contention in the interconnect. Unfortunately, this network model was so detailed that the network simulation code represented approximately 90% of the execution cost of simulation, which restricted the size of the problems that we could simulate in a reasonable amount of time. Thus, we chose to implement a simplified version of the network model in which the end-to-end latency for a message was determined strictly by the distance between the communicating nodes. We used the results of our detailed simulation model to determine the average amount of contention delay induced by the different workloads, and combined this with the minimum end-to-end latency to derive the end-to-end latencies used in the simplified model, shown in Table 3.2. Due to the accuracy of the model from which we derived the simple model's parameters, the impact on simulated performance of our simple model was small, although we will reexamine this problem as we scale up the problem and machine sizes that we simulate.

3.3 Memory Model

Figure 3 illustrates the high-level organization of Avalanche's CCU. The three major components of the CCU are the cache controller, the directory controller, and the network 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)

Simplified Model		
Parameter	Myrinet	Fast Myrinet
One hop	62	5
Two hops	112	11
Three hops	162	16

Table 2 End-to-end Latency Used In Network Approximation

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 controller is responsible for handling incoming and outgoing interconnect traffic, including the DMA and offloaded protocol processing operations described in Section 2.2. In particular, for incoming coherence messages, it is responsible for forwarding them to the directory controller or the cache controller, as appropriate.

We used the following model in our architecture simulations. We assume that each of the three control units can handle only one request at a time, which introduces contention. 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, 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

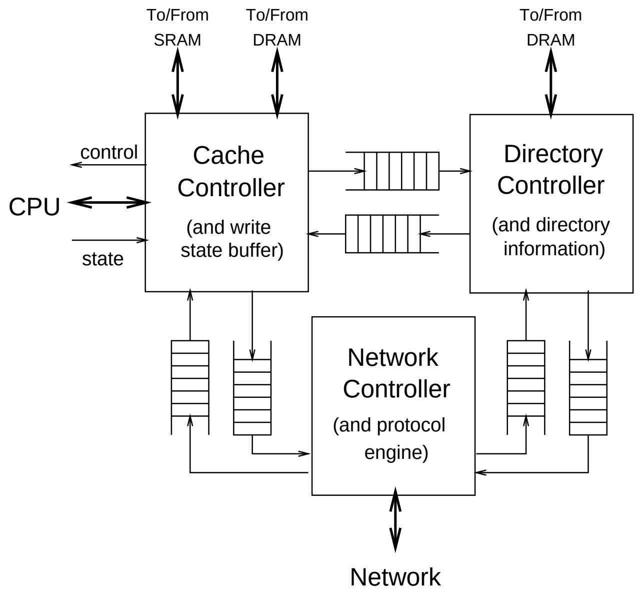

Figure 3 High-Level Cache Controller Design

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 queueing 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 3.3 lists the delay characteristics that we used in our model. 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 [15] 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 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 an accompanying tech report[5]. 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

Operation	Delay
Local read hit	1 cycle
Local write hit	1 cycle[4]
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
Dequeue a message from a controller's input FIFO	1 cycle
Update directory entry	4 cycles

Table 3 Delay Characteristics

[4]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.

[5]Although we cannot do so now without eliminating any pretext of anonymity

`opt` pseudo-protocol represents the performance achievable if the optimal protocol is used for each data block (cache line or page).

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[19], 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 [15].

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 pseduo-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. 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 [25] in our study, `mp3d`, `water`, `barnes`, `LocusRoute`, and `cholesky`. Table 4 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.

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 4, 6, and 8 are for the Myrinet interconnect, while Figures 5, 7, and 9 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 4 and 5 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 4), 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 in FLASH. For the faster interconnect (Figure 5), 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

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 4 Programs and Problem Sizes Used in Experiments

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 5 and 6 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 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.

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 5 Average operation cycle time (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 6 Average operation cycle time (fast Myrinet)

Figures 6 and 7 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.

Figures 8 and 9 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 be an important issue.

Figures 10 and 11 present an approximation of the performance that can be obtained by adding hardware support, in the form of extra state bits per page table entry or cache line, and handling each block of data with the protocol best suited to the way it is being 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 `locus` and `cholesky` even the simple page-level support can reduce the cache stall time by over 5%. Thus, while it is not clear that the performance gains are worth the added hardware complexity, minor though it may be, 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 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 [21] 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 [20], 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 [22] contains a context cache similar to previous designs such as the HP Mayfly system [14]. 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 [24] 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 [7] 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 [27] 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 [18]. 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

We have motivated the need for a multiprocessor architecture that supports higher effective scalability than existing architectures by tightly integrating the communication and memory systems. Our approach towards achieving this goal is embodied in the flexible cache and communication controller being designed at the core of the Avalanche project. The goal of Avalanche is to develop a communication and memory architecture that attacks the problem of the high effective end-to-end communication latency present in conventional designs, both for message passing and shared memory programs. Among the techniques that we are employing are the use of very streamlined sender-based message passing protocols, the ability for the communication controller to inject incoming data to any level of the memory hierarchy, and support for multiple coherence protocols. We showed via a detailed simulation study that even supporting four coherence protocols can dramatically improve application performance. Thus, we are encouraged by the preliminary results of our evaluation of Avalanche.

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] B. Bershad, D. Lee, T. Romer, and J.B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.

[7] 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.

[8] 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.

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

[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. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

[12] Cray Research, Inc. *CRAY T3D System Architecture Overview*, hr-04033 edition, September 1993.

[13] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, pages 36–43, July 1993.

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

[15] 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.

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

[17] 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.

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

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

[22] 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.

[23] S. Pakin and A. Chien. The impact of message traffic on multicomputer memory hierarchy performance. Technical Report Concurrent Systems Architecture Group Memo, University of Illinois at Urbana-Champaign, 1994. available from http://www-csag.cs.uiuc.edu/.

[24] 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.

[25] 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.

[26] M.R. Swanson and L.B. Stoller. PPE-level protocols for carpet clusters. Technical Report UUCS-94-013, University of Utah - Computer Science Department, April 1994.

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

[28] 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.

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

[30] 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.

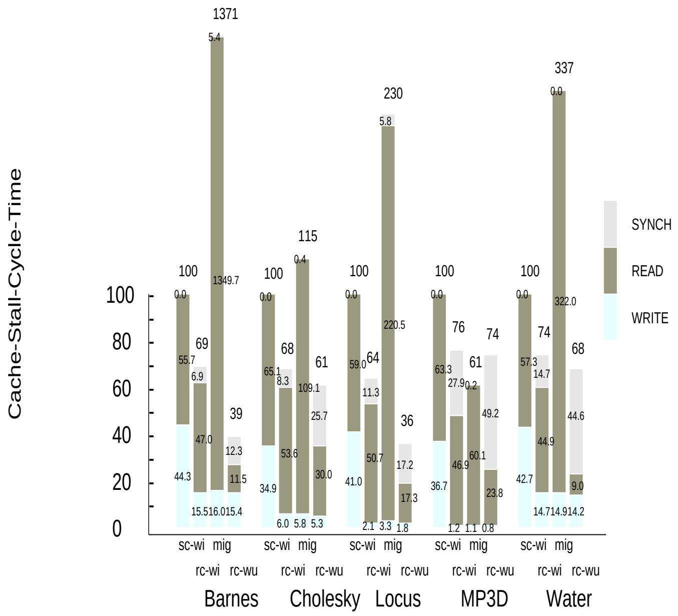

Figure 4 Cache stall time (Myrinet)

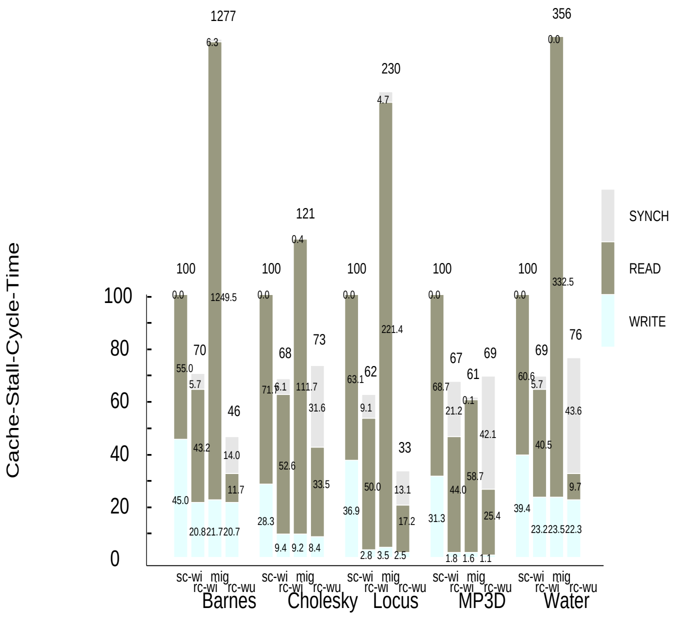

Figure 5 Cache stall time (10*Myrinet)

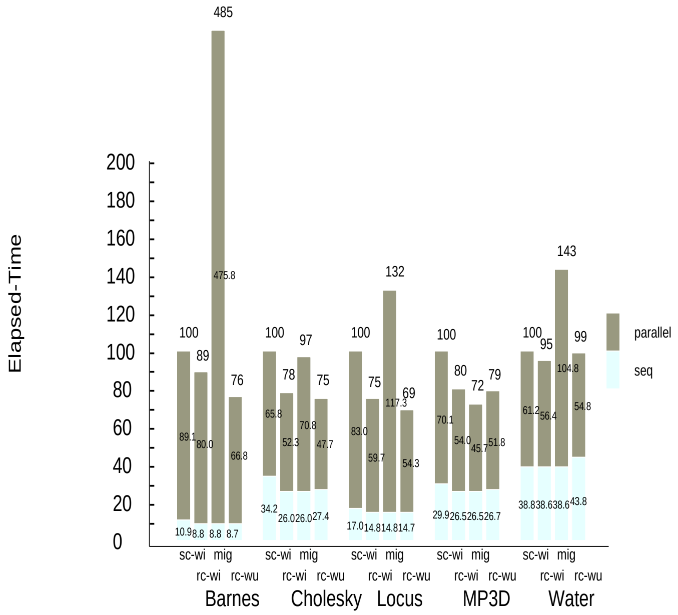

Figure 6 Execution time (Myrinet)

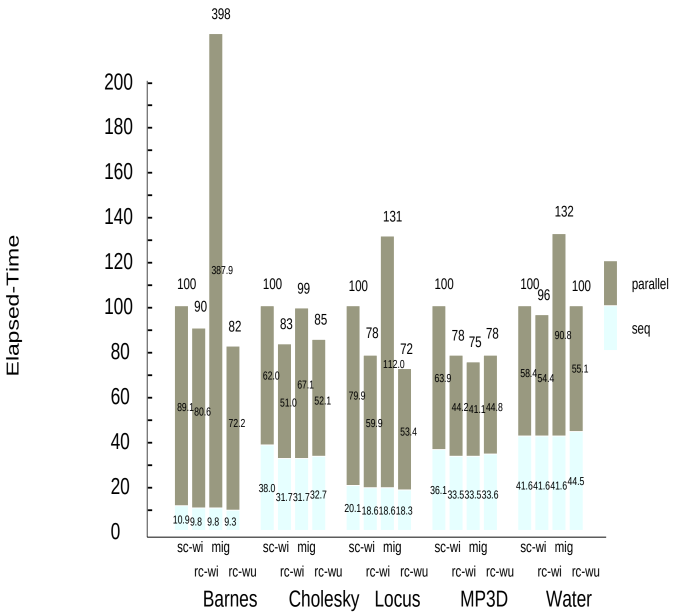

Figure 7 Execution time (10*Myrinet)

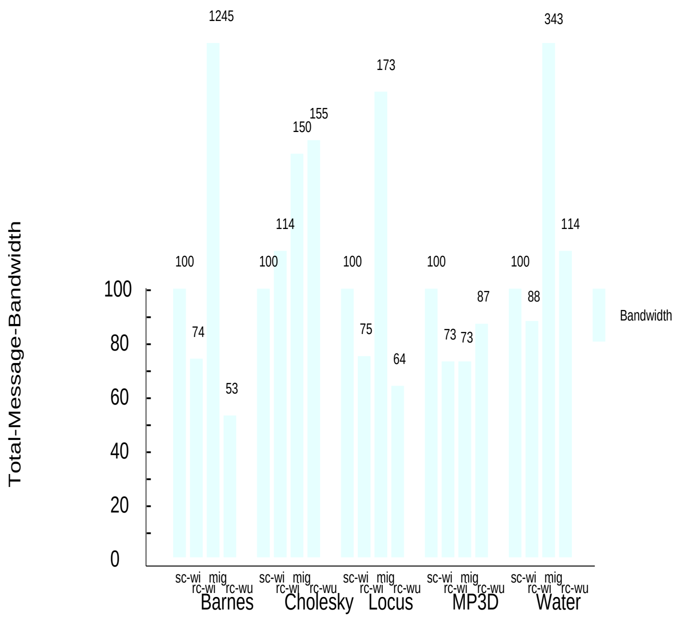

Figure 8 Bandwidth (Myrinet)

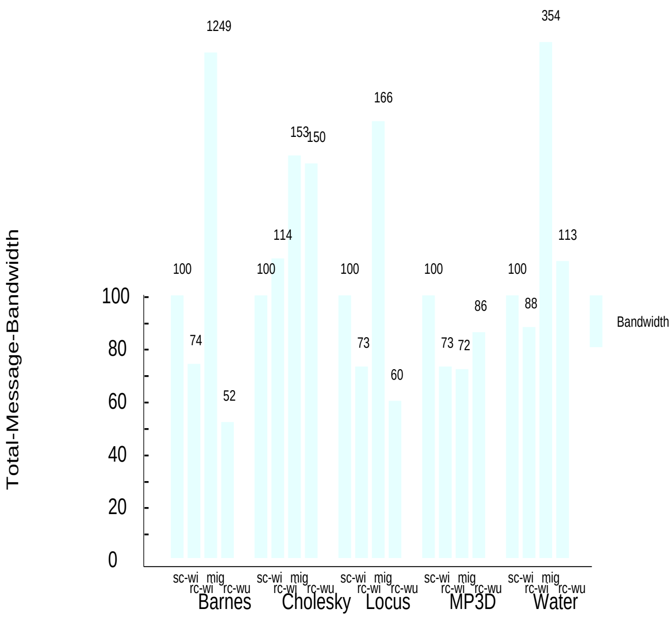

Figure 9 Bandwidth (10*Myrinet)

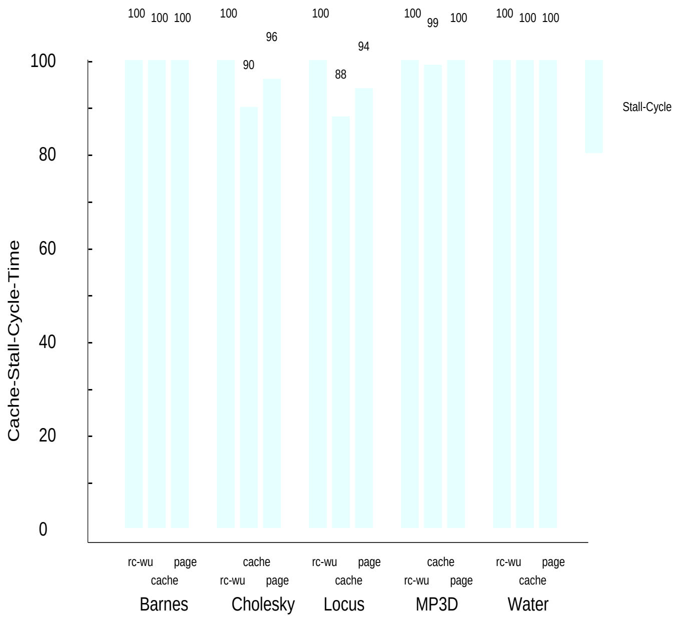

Figure 10 "Optimal" protocol performance (Myrinet)

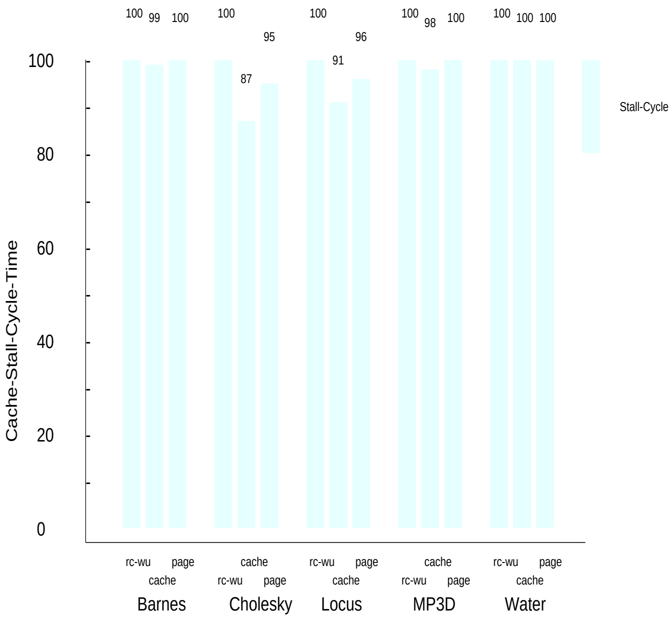

Figure 11 "Optimal" protocol performance (10*Myrinet)