Analysis of Avalanche's Shared Memory Architecture

Ravindra Kuramkote, John Carter, Alan Davis,
Chen-Chi Kuo, Leigh Stoller, Mark Swanson

UUCS-97-008

Computer Systems Laboratory
University of Utah

Abstract

In this paper, we describe the design of the Avalanchemultiprocessor's shared memory subsystem, evaluate its performance, and discuss problems associated with using commodity workstations and network interconnects as the building blocks of a scalable shared memory multiprocessor. Compared to other scalable shared memory architectures, Avalanchehas a number of novel features including its support for the Simple COMA memory architecture and its support for multiple coherency protocols (migratory, delayed write update, and (soon) write invalidate). We describe the performance implications of Avalanche's architecture, the impact of various novel low-level design options, and describe a number of interesting phenomena we encountered while developing a scalable multiprocessor built on the HP PA-RISC platform.

Analysis of Avalanche's Shared Memory Architecture

Ravindra Kuramkote, John Carter, Alan Davis,
Chen-Chi Kuo, Leigh Stoller, Mark Swanson

Computer Systems Laboratory
University of Utah

1 Introduction

The primary Avalanchedesign goal is to maximize the use of commercial components in the creation of a scalable parallel cluster of workstation multiprocessor that supports both high performance message passing and distributed shared memory. In the current prototype, Avalanchenodes are composed from Hewlett-Packard HP7200 or PA-8000 based symmetric multiprocessing workstations, a custom device called the **Widget**, and Myricom's Myrinet interconnect fabric [6]. Both workstations use a main memory bus known as the Runway [7], a split transaction bus supporting cache coherent transactions. In the Avalancheprototype a Widget board will plug into a processor slot in each node and Myrinet cables will connect each card to a Myrinet active crossbar switch providing connections to the rest of the cluster. The Avalancheprototype will contain 64 nodes, each containing between one and three processors. Sections 2 and 4 describe Avalanche's shared memory architecture in detail, while its performance implications are described in Section 6.

A unique aspect of Avalanche's architecture is that it is designed to support two scalable shared memory architectures: CC-NUMA and Simple COMA (S-COMA). Each architecture has significant advantages and disadvantages compared to the other depending on the memory access behavior of the applications. Supporting both models does not add significant design complexity to the Widget, but our current prototype design supports only the S-COMA model because the bus controller in HP workstations generates a bus error when a "remote" physical address is placed on the bus, even if the Widget signals its willingness to service the request. We are designing a solution to this problem, but in the interim, we have found that S-COMA's greater replication significantly improves performance compared to CC-NUMA in many circumstances due to Avalanche's relatively slow interconnect and the direct mapped nature of the HP chips' L1 cache.

We are also designing Avalancheto support multiple coherency protocols, which can be selected by software on a per-page basis. We currently support a *migratory* and *release consistent delayed write update* protocol, and are designing a *release consistent write invalidate protocol.* Simulation indicates that the relative performance of the three protocols varies dramatically depending on the way data is accessed[10], and it turns out that the Widget's design makes it relatively straightforward to support multiple protocols. The memory models and protocols are described in Section 3.

Finally, the decision to use primarily off the shelf hardware led to a more cost effective design than a fully custom solution, but doing so required a number of design compromises. In particular, using commodity hardware means living with mechanisms and protocols that were not designed with scalable multiprocessing in mind. In particular, Hewlett-Packard's Runway bus and memory controller design introduced a number of unexpected challenges that we are being forced to work around. The details of these quirks and their impact on performance are described in Section 7 as a lesson for future designers of scalable shared memory architectures.

2 Avalanche Design

The AvalancheWidget provides direct hardware support for low-latency message passing and distributed shared memory (DSM) functions. The board contains a Widget ASIC and 256 KB of SRAM called the Shared Buffer (**SB**). A block diagram of the Widget shown in Figure 1 contains seven distinct subsystems: the *Runway bus interface* (RIM), the *shared buffer manager* (SBM), the *shared memory cache controller* (SM-CC), the *directory controller* (DC), the *message passing cache controller* (MP-CC), the *protocol processing engine* (PPE), and the *network interface* (NI).

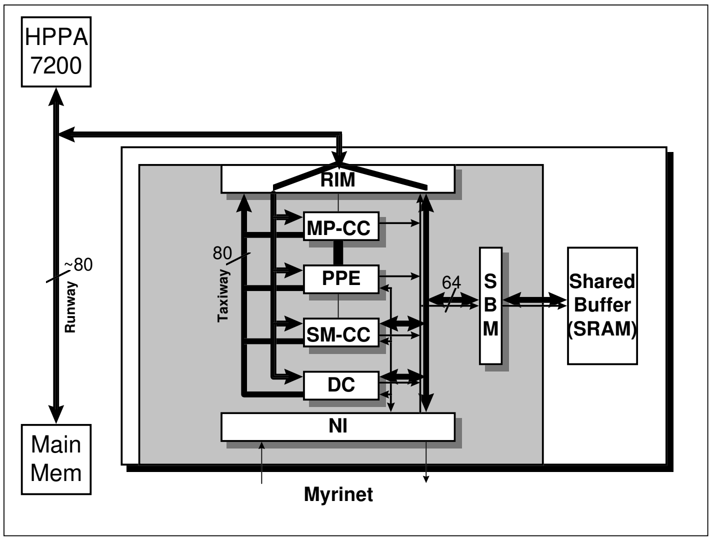

Figure 1 Widget Subsystem Block Diagram

The RIM's primary function is Runway bus mastering. In addition, it includes logic to determine what words of a shared cache line have changed by performing a *diff* of the dirty version of the line and a clean version of the line stored in the SB. It also has the logic to splice diffs into existing cache lines. This logic is used to improve the performance of our coherency protocols, as described in Section 4. The RIM also forwards a version of the Runway called the *Taxiway* to other Widget subsystems that must snoop the bus.

The SB is used as a staging area for communications and acts as a cache for incoming message data. The SB is organized as 2K 128-byte lines. SB line allocation and deallocation requests can be made by the SM-CC, DC, MP-CC, or PPE. These requests are served by the SBM.

DSM is supported by the SM-CC, which manages access to shared memory lines that are referenced locally, and the DC, which manages consistency for all shared memory lines for which the local node is *home*. The SM-CC and DC are described in detail in Section 4.

The PPE handles most message passing duties, providing direct hardware support for a set of low-latency message passing protocols known as Direct Deposit [26]. Incoming message data is stored in the SB; these lines are then managed by the MP-CC as an L2 communication cache. When local Runway clients reference this information, the MP-CC snoops resulting transactions on the Runway and supplies the data directly. This provides a significant reduction in the miss penalty that would be incurred if the message data were simply placed via the DMA interface into main memory.

The NI provides a direct interface to the Myrinet one-meter protocol, thereby eliminating the normal Myrinet interface card and its contribution to latency. In addition, like the FLASH Magic chip [17, 14], the NI splits and demultiplexes incoming messages, forwarding the data to the SB and the headers to the appropriate Widget subsystem (MP-CC, PPE, SM-CC, or DC).

The keys to scalability in cluster systems are minimizing communication latency and not degrading local node performance. Placing the Widget on the Runway bus rather than between the processor and its memory system avoids slowing down the local node. Operating the Widget subsystems in parallel avoids serialization of message passing and DSM operations. Finally, caching incoming message data on the Widget board reduces miss latencies seen by the processor.

Our decision to use commodity workstations and network components has several advantages but also carries some disadvantages. The upsides include commodity pricing, the ability to focus design time and effort on the Widget itself, and the opportunity to incorporate improved components as they become available. The downside is that certain pieces of the design space are closed off by these choices, as the behavior of commodity components must be viewed as beyond the Widget designers' control. Section 7 discusses the more serious impacts of this design choice.

3 Memory and Consistency Models in Avalanche

Placing the Widget on the Runway bus without interposing on the processor $\leftrightarrow$ cache interface or modifying HP's main memory controller (MMC) constrained our design options. We were free, however, to explore the performance implications of a broad set of memory architectures and coherency protocols. With the exception of the FLASH multiprocessor [17], existing scalable shared memory architectures employ a single memory model and coherency protocol to manage all shared data. For example, the DASH multiprocessor is a cache-coherence non-uniform access (CC-NUMA) machine with a release consistent write-invalidate protocol [18, 19], while the KSR-1 is a cache-only memory architecture (COMA) machine with a sequentially consistent write-invalidate protocol [8]. We are exploring the *potential benefits* and *implementation complexity* associated with supporting a suite of memory models and coherency protocols, and allowing software to specify the specific memory model and protocol to use to manage each page of shared data in the machine.

A unique aspect of Avalanche's architecture is that it is designed to support two fundamentally different scalable shared memory architectures: CC-NUMA and Simple COMA (S-COMA). Each architecture has significant advantages and disadvantages over the other, depending on the memory access behavior of the applications. Figure 2 illustrates both architectures.

Most large scale shared memory multiprocessor designs are CC-NUMA [18, 19, 17, 1, 11, 3]. In a CC-NUMA, the machine's global physical address space is statically distributed across the nodes in the machine, but a node can map any page of physical memory (local or remote) to its local virtual memory. To share data, nodes share a global virtual to global physical mapping table such that the same physical memory backs a particular global virtual address on all nodes. When a memory access misses in a node's local cache, the associated global physical address is used to fetch the data from either the local memory or a remote node's memory. CC-NUMA's primary advantage is its relative simplicity – a fixed "home" node is responsible for tracking the state of each block of data, which makes it easy to locate copies of data and keep them consistent. However, the amount of shared data that can be replicated locally, and thus accessed efficiently, is limited to the size of a node's cache. This leads to poor performance for applications with per-thread working sets larger than a processor cache, which exacerbates the need for large and expensive processor caches. Also, the home node for a particular block of data is often unrelated to the set of nodes that access the data, which turns potentially fast local accesses into slow remote accesses. Finally, changing the

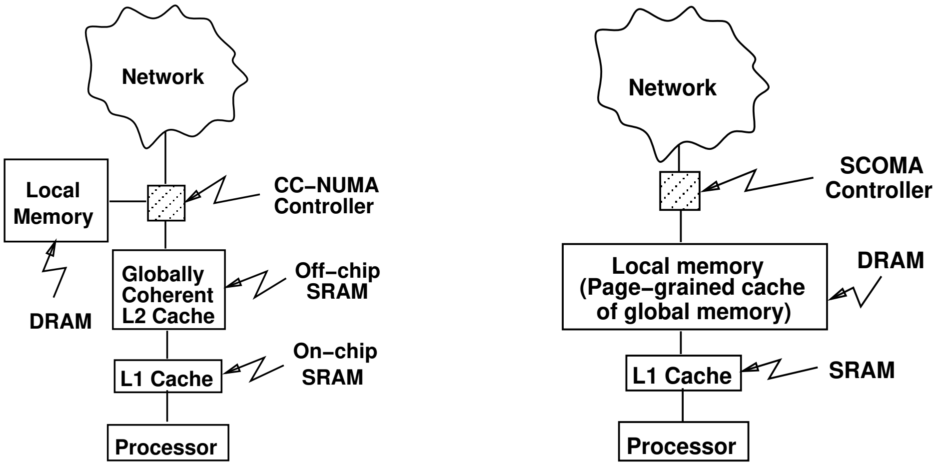

Figure 2 CC-NUMA and S-COMA Architectures

global physical to virtual address mapping, which is required before one can page memory to disk or migrate it, is very expensive since it requires a global TLB shootdown.

S-COMA divides the global address space into large chunks, typically *pages*, and uses a node's main memory as a cache for the global virtual address space. Space in this DRAM cache is allocated by the operating system in pages using the processor's memory management unit (MMU). External hardware maintains coherence at a cache line granularity within pages. When a process accesses a shared virtual address, a normal MMU check determines if the global virtual page is mapped to a local physical page. If it is not, the operating system maps a local page and resumes the access. If the MMU check succeeds, the physical address is placed on the memory bus where the S-COMA hardware determines if the access can be completed by the local main memory (e.g., the access is a `read` and the line is valid) or must be stalled while the S-COMA hardware performs some operation. Multiple nodes can have DRAM copies of shared pages. We refer to the primary DRAM copies of pages as *home pages* and replicas as *S-COMA pages*. As in a CC-NUMA, the directory controller manages the global state of locally homed pages. S-COMA addresses CC-NUMA's problems by using all of main memory to replicate shared data and by decoupling global virtual and local physical addresses. Unfortunately, because memory is allocated in page-sized units, it is possible to get very poor main memory utilization where only a few cache lines are active in any given page. As the ratio of an application's working set size to the size of main memory increases, this leads to high rates of paging and poor performance [23].

Ideally, we would like to support both CC-NUMA and S-COMA so that we can benefit from S-COMA's greater replication when memory utilization is high or paging rates are low, but fall back to CC-NUMA mode when page fault rates increase. Supporting both models does not add significant design complexity to the Widget. Applications and operating systems could tailor the behavior of the consistency management hardware based on how memory is typically accessed. Our current simulation model supports only an S-COMA model because of constraints imposed on us by the Runway bus. This performance benefits of S-COMA are discussed in detail in Section 6.

Three flavors of memory coherency protocols have been proposed and implemented over the years: *write-invalidate*, *write-update*, and *migratory* [2]. Most modern scalable shared memory machines employ a write-invalidate protocol because it requires the least management overhead, scales well, and handles read sharing efficiently. However, as with memory architectures, the performance of applications running under each of the different possible coherency protocols differs dramatically depending on the specific access behavior of the application [4].

Write-invalidate protocols perform well when threads frequently read shared data, but only infrequently modify it or the modifications are restricted to pieces of shared data that only the modifying node accesses frequently. They perform poorly when data is heavily shared and writes are frequent, because after each write data must be reloaded when next accessed by remote nodes. Migratory protocols slightly improve performance for applications where memory is concurrently shared infrequently [13, 24]. Write-update protocols work well when writes are frequent and written data is typically read by remote nodes prior to being overwritten, exactly those cases handled poorly by write-invalidate. They generate excessive communication overhead when modifications to shared data are not typically read by remote nodes. Depending on the application, the choice of coherency protocol can affect running time by 35% or more [4, 9].

Because the relative performance of the three protocols varies dramatically depending on the way data is accessed, the goal of Avalancheis to support all three protocols on a software-specifiable per-page basis. However, due to problems with the Runway bus described in Section 7, we currently support only a migratory protocol and a release consistent write-update protocol that exploits the *diff*ing mechanism provided by the RIM. We present performance using the migratory protocol in Section 6 because it better isolates the benefits and imperfections of our design.

4 Avalanche Controller Designs

Avalanche divides shared memory into 4-kilobyte pages, 128-byte Avalanche cache lines, and 32-byte PA-RISC cache lines. The operating system is responsible for allocating, deallocating, mapping, and unmapping pages. The SM-CC is responsible for managing the 128-byte Avalanche cache blocks, 32 per 4-kilobyte page. We chose a coherency unit larger than a native cache line to reduce buffer fragmentation, network fall-through latency, and the amount of DSM state information storage. A portion of main memory on each node stores shared data - the size of this region is determined by a boot parameter. The region is contiguous so that the SM-CC can respond to coherent bus requests to non-shared memory immediately.

Management of shared data is divided between the SM-CC and DC as follows. The SM-CC manages the state of the blocks accessed by local processors while the DC maintains global coherency information for locally homed shared data. For all the shared memory pages *mapped* on a node, the local SM-CC maintains two data structures: (i) *page state information*, including the coherency protocol, the identity of its home node, and the page address on that home node, and (ii) *block state information* for each block cached on the local node (*invalid*, *shared*, or *exclusive*). For all the shared memory pages *homed* on a node, which are all shared pages in CC-NUMA mode and primary copies of S-COMA pages, the DC maintains two data structures: (i) *page state information*, including the coherency protocol, home node, and virtual address of the page on the home node, (ii) *block state information* for each block containing its *global state* (*free*, *shared*, or *exclusive*) and a list of nodes that have a copy of the block. Rather than store all of this metadata on the Widget, which would require a large amount of on-chip memory and thus increase the Widget's cost and complexity significantly, the DC and SM-CC store their metadata in main memory. This metadata is initialized by the kernel when the shared memory page is allocated at or migrated to the node. As a performance optimization, the DC and SM-CC use the SB to cache recently accessed metadata. The performance implications of this method of managing DSM metadata are discussed in Section 6.

The SM-CC and DC are implemented as a collection of concurrent finite state machines. This design permits maximally concurrent operation and minimizes control induced latency and reduces resource conflicts that would be present under very high DSM traffic loads. Both the SM-CC and the DC can perform up to four concurrent operations to independent cache lines. For example, while the metadata for one shared memory operation is being loaded from the SB or the data requested in a `read` operation is being loaded from main memory, the SM-CC or DC can operate on another request. In both the SM-CC and DC, separate state machines (i) examine incoming requests to acquire the necessary metadata, (ii) snoop the internal Taxiway bus to detect the completion of requested memory operations, (iii) perform the necessary coherency protocol operations, and (iv) stage outgoing messages to the NI or other local DSM subsystems. The low level design of both the SM-CC and DC are beyond the scope of this paper, but can be found elsewhere [16].

The SM-CC includes a number of unique features designed to exploit the flexibility of release consistency. It maintains counts of pending *invalidate* and *update* acknowledgments similar to DASH [19, 18] and an acquire state buffer to delay updates similar to the way that coherence update buffers delay invalidates [20]. In addition, it maintains a *release state buffer* (RSB) to support the release consistent write update protocol. When a local processor acquires ownership of a shared block, the SM-CC stores a clean copy of the requested line in the RSB. When the RSB becomes full or the processor performs a release operation, the RSB uses the RIM's *diff*ing function to compute a mask of the words that have been modified. This mask and the dirty line are used to send a compressed update message containing the modified words and their positions to remote nodes caching the block, which use the RIM's splicing functionality to incorporate the changes. Thus, the RSB supports a *delayed write update* protocol similar to Munin, which significantly improves performance and scalability [9].

5 Experiment Setup

The simulation environment developed by the Avalancheproject is based on a simulator for the HP PA-RISC architecture, including an instruction set interpreter and detailed simulation modules for the first level cache, the system bus, the memory controller, the network interconnect, and the Widget. This simulator is called Paint (PA-interpreter)[25, 27] and is derived from the Mint simulator[28]. Paint is designed to model multiple nodes and the interactions between nodes, with emphasis on the effects of communication on the memory hierarchies. Paint provides a multiprogrammed processor model with support for operating system code so that the effects of OS/user code interactions can be modeled. The simulation environment includes a kernel based on 4.4BSD that provides scheduling, interrupt handling, memory management, and limited system call capabilities. The VM kernel mechanism was extended to provide the page translation support needed by distributed shared memory.

Figure 3 shows the parameters used for various components in the simulation. The processor, Widget, and Runway are all clocked at 120MHZ and all cycle counts shown are with respect to this. The cache model is based on the PA-8000, which can do aggressive out-of-order execution with 28 load/store slots. To simplify discussion of performance data, the model is configured, with one exception, as a blocking cache with one load/store slot. The SM-CC and DC caches are configured as 2-way and 4-way set associative, respectively. The 4 cycle hit time to the SM-CC and DC caches consists of two cycles of arbitration for the SB bus and two cycles for the off-chip read. The Main Memory Controller (MMC) is modeled on current HP workstations [15]. It contains 4 banks and returns the first doubleword of data to the Runway 26 cycles after a read appears on the bus. Due to its interleaving, the MMC can satisfy a 128 byte (four cache line) request from the Widget in

44 cycles. The simplified model of the Myrinet network only accounts for input contention; the latency for a message is computed simply from the distance in switches between the communicating nodes clocked at 160MHz.

We used five programs from the SPLASH-2 benchmark suite [29] in our study: `radix`, `fft`, `lu:contiguous`, `lu:non-contiguous`, and `barnes`. Figure 4 shows the inputs used for each test program. All the programs were run with the base problem size as suggested in the distribution. The total pages column in Figure 4 indicates the number of shared data pages each application touched, so the shared data space touched by the applications ranged from 2MB to 3.1MB.

6 Performance Analysis

Avalanche's shared memory architecture differs from other architectures in (i) its use of S-COMA as the primary memory model, (ii) the Widget's position as a peer to the CPU, with no control over the main memory controller, and (iii) the use of distributed state machines in its controllers. In this section we analyze issues resulting from these design decisions, including S-COMA's effectiveness with respect to reducing remote memory misses, the factors most responsible for the latency of various kinds of misses, and the effectiveness of overlapping multiple requests in the SM-CC and DC.

All of the benchmarks had high L1-cache hit rates, between 98.6% and 99.6%. Figure 5 shows the total number of misses on all nodes. The number in parentheses indicates the percentage of the misses that were to shared memory. The ratio of shared memory misses to local memory misses declines with increased node count because the problem sizes were kept constant.

To study the effect of Avalanche's large coherency unit and the S-COMA model, shared memory misses are classified as follows:

Cache Location	Characteristics
`L1 cache`	1 MB, direct-mapped, 32-byte lines, blocking, two write back buffers, 1 cycle hits, 32 cycle misses (best case), virtually indexed, physically tagged.
`SM-CC Cache`	16 KB, 2-way set associative, 32-byte lines, blocking, no writeback buffer, 4 cycle hits, 36 cycle misses (best case).
`DC Cache`	16 KB, 4-way set associative, 32-byte lines, blocking, no writeback buffer 4 cycles on hit, 36 cycles on miss (best case)
`Myrinet`	Propagation delay: 1 cycle, Fall through delay: 27 cycles, Topology: 4 Switches (2x2).

Figure 3 Simulation Parameters

Program	Input parameters	Total Pages
`radix`	256K Keys, Radix = 1024	528
`FFT`	64K Points, line size 32 and cache size 1MB	784
`LU Non-contiguous`	512x512 matrix, 16x16 blocks	519
`LU Contiguous`	512x512 matrix, 16x16 blocks	521
`barnes`	`input` file with 16K particles	779

Figure 4 Programs and Problem Sizes Used in Experiments

Application	Number of Nodes		
	4	8	16
`radix`	0.25M (64%)	0.34M (50%)	0.52M (38%)
`fft`	0.5M (80%)	0.54M (66%)	0.60M (45%)
`LU non-contiguous`	1.51M (93%)	1.62M (89%)	1.83M (82%)
`LU contiguous`	0.48M (85%)	0.62M (77%)	0.76M (72%)
`barnes`	2.0M (57%)	1.86M (53%)	4.29M (51%)

Figure 5 Total Number of Misses and Percentage of Shared Miss

- Any processor cache miss that causes the SM-CC to read a block from a remote node is classified as *coherency miss (COM)*. Note that due to the use of a migratory protocol, remote blocks are invalidated even on a read miss. These misses represent accesses that are inherently remote given the particular coherency protocol.
- A *spatial miss (SM)* is defined as any access to a cache line that misses in the L1 cache, but hits in local memory as a side effect of the SM-CC having recently loaded another 32-byte cache line within the same 128-byte Avalancheblock. These misses represent potentially remote accesses that are made local by Avalanche's large coherency unit.
- A *capacity/conflict miss (CCM)* is defined as an access to data with a remote home node that misses in the L1 cache due to a cache conflict or capacity problem, but hits in local memory. These misses represent potentially remote accesses that are made local by S-COMA page replication.
- All other misses to shared memory are classified as *non-remote misses (NRM)*. This category includes misses to shared data never accessed by a remote processor, cold misses, and capacity/conflict misses to shared memory whose home node happens to be the local node. These misses represent accesses that are inherently local.

Figure 6 shows the breakdown of misses across each of the above classes. Inherent *coherency misses* vary from 7% to 34% of all misses. Their number increases as the number of nodes increases due a combination of the low percentage of migratory data in the benchmarks and false sharing. The next two categories, *spatial misses* and *capacity/conflict* misses, represent potentially remote misses that become local misses in Avalanche. *Spatial misses* vary from 11% to 41% of all misses. The large number of spatial misses indicates that the large block size results in a high degree of effective prefetching that turns potentially remote SM misses into local memory accesses[1]. *Capacity/conflict misses* vary from 3% to 57% of all misses. The large number of capacity/conflict misses, caused primarily by the direct mapped nature of the PA-RISC's L1 cache, indicates that S-COMA's use of local memory as a backing store for cache lines forced out of the L1 cache is beneficial. The sum of SM and CCM misses varies from 40% to 70%, which indicates that 40% to 70% of remote misses can be made local by employing an S-COMA architecture with large coherency units. To be fair, one can do prefetching in CC-NUMA to handle the spatial misses efficiently, but most processors do not support injection directly into their L1 cache from an external unit.

[1]The ratio of spatial misses to coherency misses varies from approximately 1:1 to 3:1, which indicates that on the average one to three of the extra cache lines loaded by the local SM-CC are eventually accessed.

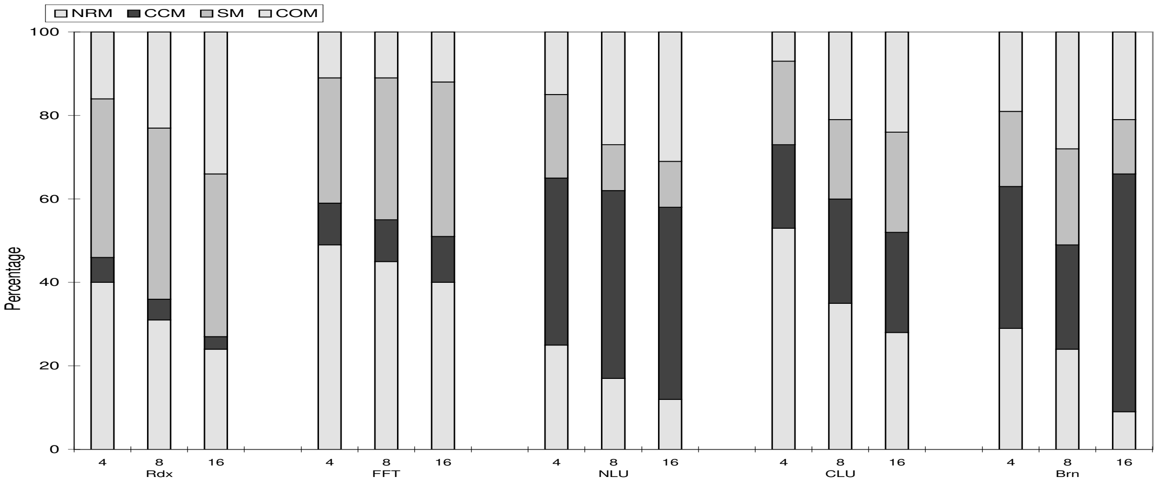

Figure 6 Miss Classifications

Recall that the SM-CC and DC store a small amount of their metadata in the SB and the rest in main memory. The Runway bus protocol requires that the SM-CC provide a "coherency response" signal for every coherent bus operation, whether or not the associated data is being managed by the SM-CC or a remote node. Thus, the latency of shared data accesses that miss in the L1 cache is affected by the SM-CC's metadata hit rate, as follows:

- If the SM-CC finds the pertinent state information in the SB and the state information indicates that the line is valid in local memory, we categorize this miss as a *local shared miss with state information hit (LSMSH).* In this case, the SM-CC can immediately respond and let main memory supply the data.

- If the pertinent metadata is not present in the SM-CC cache, the SM-CC must provide a coherency response that promises that it will supply the data to the requesting processor. After the SM-CC reads its metadata from main memory, if the line is found to be valid in local memory, we categorize this miss as a *local shared miss with state information miss (LSMSM).* In this case, the SM-CC must also read the data from memory and supply it to the processor through a *cache to cache copy.*

- If the line is not valid in main memory, the SM-CC sends a message to the DC at the home node requesting the data[2]. The DC invalidates the current owner, which forwards the block directly to the requesting SM-CC. When the home node is the local node, we categorize this as a *local DC miss (LDCM)*, which generates two network messages (the invalidation and the ack). When the home node is a remote node, we categorize this as a *Remote DC miss (RDCM)*, which generates either two or three messages depending on whether the home node is also the current owner.

Figure 7 shows the percentage of each type of miss. The minimum latency observed in our runs for each type of miss was 32, 154, 354, and 452 cycles respectively. This shows the importance of reducing the number of coherent misses in our architecture. The LSMSM category represent

[2] If the local node happens to be the home node, the SM-CC sends the message to the local DC.

misses that could be handled directly by the local memory if not for the cached nature of the Widget's metadata. The extra bus transactions add 50 to 59 cycles of latency to each of these misses. Happily, for all of the benchmarks except for `radix`, LSMSM misses represent less than 10% of all shared data misses, which limits the negative impact of this design. Section 7 provides more detail on the cause of this overhead.

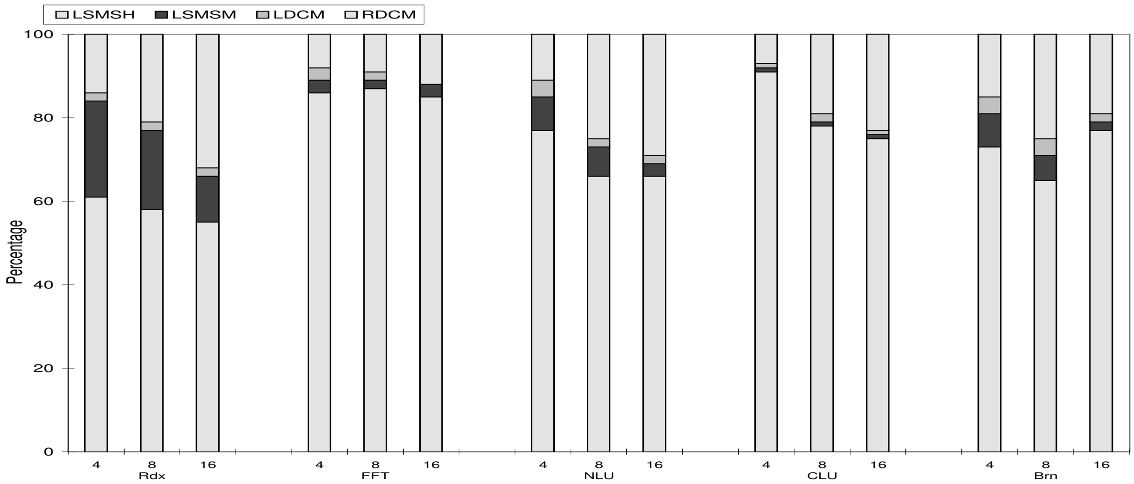

Figure 7 Miss Types Based on Latency

As described in Section 4, the SM-CC and DC are capable of handling up to four independent requests concurrently. Figure 8 illustrates how effective the SM-CC and the DC were in overlapping request processing. This study was done with an aggressive non-blocking cache similar to the PA-8000. The *active* column shows the total cycles during which the SM-CC/DC was handling at least one request. The *cumulative* column shows the sum of cycles spent by all requests. The *usage* column shows the time between when the SM-CC/DC first receives a request until it completes processing the last request. The *overlap* column shows the additional penalty that would have resulted if the controllers were totally sequential. This is significant considering the fact that the SM-CC was active only for 0.7% to 6% of the time and the DC for 0.4% to 15% of the time. We expect to see more overlap with larger problem sizes and multiple processors on the bus.

Application		Cumulative	Active	Usage	Overlap
`radix`	DC	495177	469982	1%	6%
	SM-CC	2707404	1537878	6%	44%
`fft`	DC	415033	334768	0.4%	20%
	SM-CC	1177557	981484	4%	17%
`LU non-contiguous`	DC	4628614	3336216	15%	28%
	SM-CC	9450756	8445506	4%	11%
`LU contiguous`	DC	1186936	896268	2%	25%
	SM-CC	2696289	2518012	0.7%	7%
`barnes`	DC	4034096	3828046	0.7%	6%
	SM-CC	10861286	10211076	2%	6%

Figure 8 DC and SM-CC Occupancy in Cycles

The *active* cycles were spent in one of three ways: (i) *busy* performing useful work, (ii) *waiting for the SB* to supply metadata, or (iii) *waiting for main memory*. The percentage of active cycles spent performing useful work varied from 37% to 46%, meaning that between 54% and 63% of the time when they were active, the SM-CC or DC were waiting for data or metadata. Most of the waiting time (between 32% to 50% of the total active cycles) was spent waiting for metadata to be supplied from the SB, which is primarily due to the latency of arbitrating for the shared SRAM buffer. This delay could be reduced significantly by providing a separate SRAM buffer for metadata, but this option is precluded by the limited pin count available without significantly increasing the cost and complexity of Widget fabrication. We are considering various other design options such as caching additional metadata in the Widget.

7 Penalty for Using Off the Shelf Hardware

Using a high degree of commercial hardware is more cost effective than custom solutions, and it allows our design to be incorporated into a commercial product line relatively easily. However, it also forces us to live with mechanisms and protocols that were not designed with scalable multiprocessing in mind. This section discusses some of the problems that have arisen due to the use of commodity hardware and their performance impact.

The backbone of a Hewlett-Packard workstation is the Runway bus [7] and the Main Memory Controller (MMC) [15]. The split transaction Runway bus allows data to be delivered out of order. A number of *clients* are connected to the bus, where a client can be a CPU or intelligent IO controller. When any client performs a coherent memory operation, all clients are required to issue a *coherency response* to the MMC indicating the state of the requested cache line in their respective caches. The MMC collects the coherency responses from all clients before allowing subsequent memory requests to be retired; in other words, a client *must* provide a coherency response to a pending request before any additional memory operations can be completed.

For this discussion, the relevant coherency responses are (i) COH_OK, which indicates that the responding client has no interest in the line, (ii) COH_SHR, which indicates that the responding client has a read-only copy of the line and wishes to retain it, and (iii) COH_CPY, which indicates that the responding client has a dirty copy of the line and will send the data directly to the requesting client. If all clients respond with *COH_OK*, then the MMC supplies data from main memory and the requesting client marks the state of the line as *private*. If any client responds with *COH_SHR*, the MMC supplies the data from main memory, but the requesting client marks the state of the line as *shared*. No client may write to a shared line without first performing a write request. If a client responds with *COH_CPY*, the MMC discards the request and the requesting client simply waits for the responding client to perform a cache to cache copy to supply the data, which, because it is a split transaction bus, may be deferred arbitrarily long. Upon receiving the data from the responding client, the requesting client marks the state of the line as *private*, and the responding client is required to invalidate its copy of the data.

Three problems arose in designing Avalanche: *inflexible memory operation ordering*, *Runway's migratory bus protocol*, and *out of address range bus exceptions.*

Inflexible memory operation ordering: For every coherent read to shared memory, the SM-CC must determine the state of the line before issuing a coherency response. If any of the nodes for which it is acting as a proxy have a dirty copy of the data, it must respond with *COH_CPY*. If any of the nodes have a shared copy of the data, it must respond with *COH_SHR*. If, however, the state information associated with the line is not present in the SM-CC's internal cache, the SM-CC cannot make this determination. Ideally, the SM-CC would simply perform a read operation to load the necessary state information, and respond appropriately. Unfortunately, such a read would be blocked by the outstanding coherent operation, so the SM-CC is forced to generate a coherency response without knowing the state of the line. Thus, it must assume the worst case, whereby a remote node has a dirty copy of the data, and respond with *COH_CPY*. It can then issue the read for the line's state information. Having once responded with *COH_CPY*, even if it determines that the line was valid in memory and that a *COH_OK* would have been appropriate, it is still required to supply the data to the requester. Thus, it must issue a read to main memory to obtain the data and supply it to the requesting client via a cache to cache write, wasting bus bandwidth and increasing latency. Also, because a node that performs a cache to cache write must invalidate its copy, if it turns out that a remote node (or nodes) had a clean copy of the data, the SM-CC must still invalidate those copies prior to sending the data to the requester. This is another source of latency, and can increase the miss rate at the remote nodes if the data is not being used in a migratory fashion. Note that this is due to the migratory nature of the Runway bus, and is required for any sequentially consistent Avalancheprotocol.

If the MMC supported *non-coherent, out-of-order* reads, the SM-CC could read its state information prior to generating a coherency response. In the case where the data was valid in local memory, the SM-CC would respond with *COH_OK* or *COH_SHR*, the line would be supplied directly from memory, and remote copies would not be invalidated. The LSMSM portion of misses in Figure 7 shows the percentage of shared memory misses where this problem arises, ranging from 1% to 23%. This problem is most noticeable when the SM-CC cache is cold or when the local processor is accessing many shared memory lines, overwhelming the capacity of the SM-CC cache. Reading the data from memory and supplying it to the local client adds between 50 and 59 cycles per miss.

Runway's migratory bus protocol: As described above, when a client responds with *COH_CPY* and later supplies the data with a cache to cache write, the data *migrates* to the requesting node. This design makes it impossible to support a sequentially consistent write invalidate protocol in a distributed shared memory environment! If a client performs a coherent read to data that is invalid in local memory, the SM-CC must fetch the line from the remote node and supply it to the local requester. However, the SM-CC cannot delay the coherency response while it is requesting the data from the remote node, because doing so can lead to deadlock if two nodes attempt to invalidate lines in each other's caches simultaneously. Thus, as above, the SM-CC must respond with *COH_CPY*, supply the data via a cache to cache write after fetching it, and invalidate all remote copies of the data. Thus, only a migratory protocol.can be sequentially consistent.

It would be easy to support a weak consistent write invalidate protocol that supports read sharing, by having the SM-CC read the data back from the processor cache immediately after supplying it. Should the local processor modify the data before we acquire a clean copy, we can request ownership of the line and use the RIM's diffing hardware to coalesce the local modifications with any remote modifications that may have occurred. Figure 9 shows that 8% to 28% of shared read misses would require the SM-CC to perform these extra bus transactions to change the state of the cache line in the processor cache back to shared mode. Although these transactions would not show up directly as latency to read misses, they would increase the Widget and MMC controller occupancies and bus bandwidth consumption and thus indirectly impacts performance.

Out of address range bus exceptions: Finally, when a processor generates an address that lies outside the range of physical addresses supported by the local main memory, the MMC generates a bus exception. This behavior makes it difficult to support the CC-NUMA model, because the processor cannot directly generate remote physical memory addresses without causing the system to crash. This problem led to the adoption of Simple-COMA as Avalanche's primary memory model, notwithstanding S-COMA's much better caching of remote data. Figure 10 illustrates the potential impact of using an Simple COMA-only design. It shows a snapshot of the page occupancy

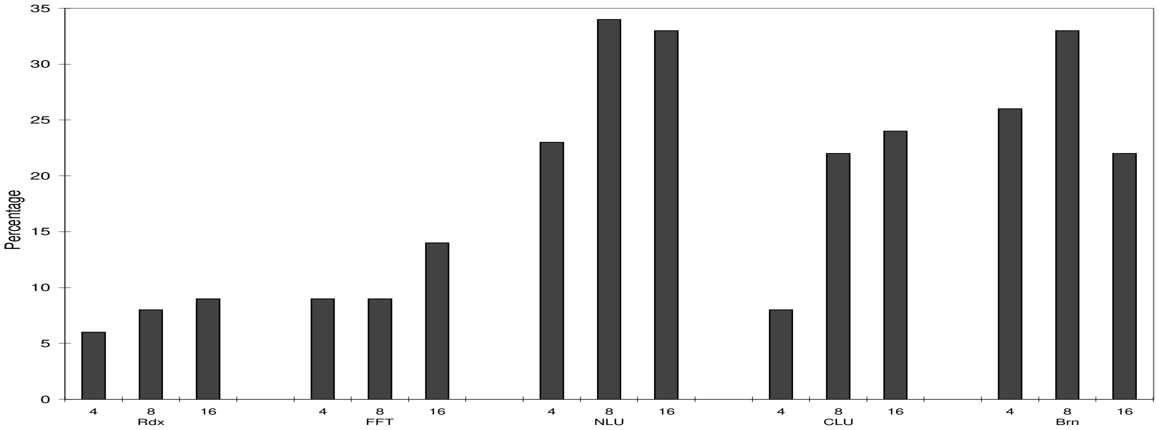

Figure 9 Write Invalidate Penalty

on all nodes. The shaded bands represent the percentage of pages for which there are less than or equal to 8, 16, or 32 valid blocks (out of a possible 32 blocks per page). The number of pages with less than 8 valid blocks, and thus less than 25% utilization, varies from 6% to 96%. When memory pressure is high it is better to use the CC-NUMA model on these underutilized pages, accepting increased remote misses to avoid page level "thrashing." To handle this problem we are attempting to circumvent the MMC's addressing restriction to support a mixed S-COMA/CC-NUMA architecture.

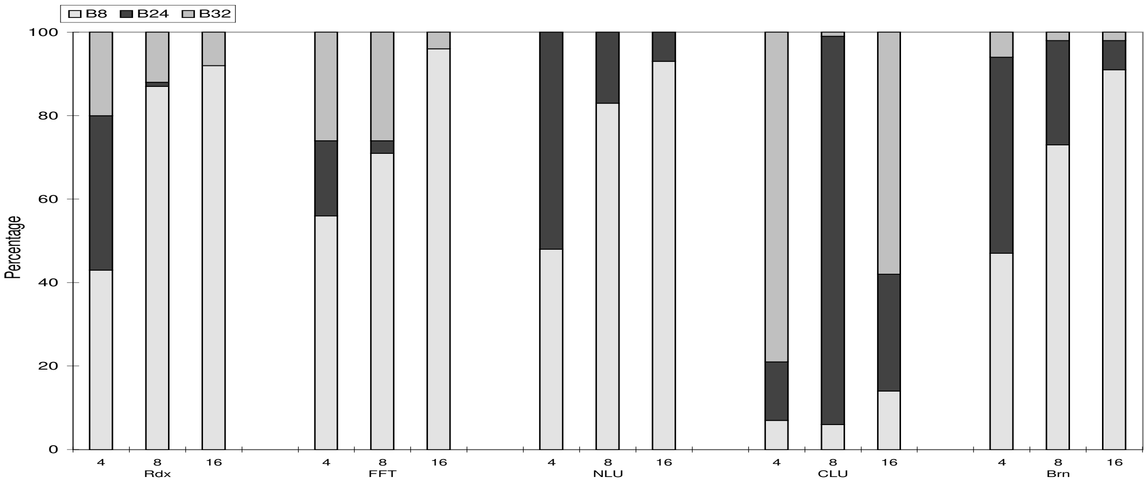

Figure 10 SCOMA: Page Occupancy

In summary, although the benefits of commercial components are significant, they are not without drawbacks. In particular, the lack of a facility for out-of-order reads and imposition of a migratory-only cache coherency protocol result in noticeable performance overhead when building a scalable shared memory multiprocessor. These constraints also make it impossible to support a sequentially consistent write-invalidate protocol. Causing a bus exception for out of range addresses makes it difficult to support a CC-NUMA architecture. These constraints represent the kinds of things that architects must overcome when using commercial components. We hope that our experience will help guide future commercial system developers who might impose similar constraints inadvertently if not made aware of the outcome of their decisions. As scalable SMPs become an increasingly important market segment, their design needs should be considered.

8 Related Work

The Stanford DASH multiprocessor [19, 18] used a 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 [3].

The MIT Alewife machine [11, 12] was one of the first machines to implement directory based shared memory. It was also the first hardware-based SM system to use software for protocol processing. Alewife used a directory-based cache design that supports invalidation-based consistency protocol. Alewife also had support for fast message passing.

The Stanford FLASH [17, 14] is a second generation DASH multiprocessor that offloads the protocol processing to a processor situated on the MAGIC chip. The MAGIC chip's processor possesses its own instruction and data caches for holding, respectively, the protocol code and the protocol metadata. By having a separate processor, the FLASH system is able to provide flexibility that can be used to support different protocols.

The user level shared memory in the Tempest and Typhoon systems [22] supports cooperation between software and hardware to implement both scalable shared memory and message passing abstractions. Like FLASH, the proposed system uses low level software handlers to provide flexibility including memory architecture similar to SCOMA called *stache* that uses the node's local memory to replicate remote data.

The SHRIMP Multicomputer [5] 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. A store to such a shared page is forwarded to the remote node where it is placed into main memory. Since the network controller is not tightly coupled with the processor, the cache must be put into write-through rather than write-back mode so that stores to memory can be snooped by the network interface; this results in an increase in bus traffic between the cache and main memory.

The S3.mp multiprocessor system [21] was developed with the goal of using a hardware supported DSM system in a spatially distributed system connected by a local area network. For the interconnect it used a new CMOS serial link which supported greater than 1Gbit/sec transfer rate. The shared memory hardware system was tightly coupled to the memory controller and, even used extra ECC bits to store state information.

9 Conclusions

The primary Avalanchedesign goal is to maximize the use of commercial components in the creation of a scalable parallel cluster of workstation multiprocessor that supports both high performance message passing and distributed shared memory. We have described a design that accomplishes this goal by combining a cluster of commercial multiprocessor workstations, a high speed commodity interconnect, and a small custom VLSI Widget. In our prototype, a Widget board plugs into a processor slot on each of 64 nodes and interfaces with the Runway bus and the Myrinet fabric to maintain data coherency.

A unique aspect of Avalanche's architecture is that it is designed to support two scalable shared memory architectures: CC-NUMA and Simple COMA (S-COMA). Supporting both models does not add significant design complexity to the Widget, but our current prototype design supports only the S-COMA model because of problems associated with the Runway bus controller. Notwithstanding our desire to support both CC-NUMA and S-COMA, we have found that S-COMA's greater replication significantly improves performance in many circumstances due to Avalanche's relatively slow interconnect and the direct mapped nature of the HP chips' L1 cache. We are also designing Avalancheto support multiple coherency protocols. We currently support a *migratory* and *release consistent delayed write update* protocol, and are designing a *release consistent write invalidate protocol.*

Finally, the decision to use primarily off the shelf hardware led to a number of design compromises. Hewlett-Packard's Runway bus and memory controller design introduced a number of unexpected challenges involving the bus protocols and memory controller requirements. Despite these problems, we believe that future scalable shared memory multiprocessors *must* be based on commodity components, so it is imperative that architects designing both commodity components and multiprocessor architectures consider scalability in their base designs. In the final analysis, the Widget is a very minor component of the system cost and can be viewed as a Myrinet to HP workstation interface card that minimizes latency while supporting DSM and message passing transactions in the resulting cluster.

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

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

[5] 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.

[6] 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(1):29–36, February 1995.

[7] W.R. Bryg, K.K. Chan, and N.S. Fiduccia. A high-performance, low-cost multiprocessor bus for workstations and midrange servers. *Hewlett-Packard Journal*, 47(1):18–24, February 1996.

[8] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the KSR-1 computer system. Technical Report KSR-TR-9002001, Kendall Square Research, February 1992.

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

[10] J.B. Carter, M. Hibler, and R.R. Kuramkote. Evaluating the potential of programmable multiprocessor cache controllers. Technical report, University of Utah, 1994.

[11] 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.

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

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

[14] 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.

[15] T.R. Hotchkiss, N.D. Marschke, and R.M. McClosky. A new memory system design for commercial and technical computing products. *Hewlett-Packard Journal*, 47(1):44–51, February 1996.

[16] R. Kuramkote, J. Carter, A. Davis, C. Kuo, L. Stoller, and M. Swanson. The design of shared memory cache and directory controller in avalanche. Technical report, University of Utah - Computer Science Department, November 1996.

[17] 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.

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

[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] J. Wang M. Dubois, L. Barroso and Y. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proceedings of Supercomputing'91*, pages 197–206, 1991.

[21] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp scalable shared memory multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, 1995.

[22] 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.

[23] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for Simple COMA. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 276–285, January 1995.

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

[25] L.B. Stoller, R. Kuramkote, and M.R. Swanson. PAINT- PA instruction set interpreter. Technical Report UUCS-96-009, University of Utah - Computer Science Department, September 1996. Also available via WWW under http://www.cs.utah.edu/projects/avalanche.

[26] L.B. Stoller and M.R. Swanson. Direct deposit: A basic user-level protocol for carpet clusters. Technical Report UUCS-95-003, University of Utah - Computer Science Department, March 1995. Also available via WWW under http://www.cs.utah.edu/projects/avalanche.

[27] M. Swanson and L. Stoller. Shared memory as a basis for conservative distributed architectural simulation. In *Parallel and Distributed Simulation (PADS '97)*, 1997. Submitted for publication.

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

[29] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.