Design and Evaluation of a Distributed Shared Memory Controller with Multiple Access Protocols

Ravindra Kuramkote

Computer Systems Laboratory
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

1 Introduction

Hardware Distributed Shared Memory (DSM) systems are primarily targeted for parallel applications that have fine grained communication requirements. Such systems have two driving goals: (i) minimizing the number of misses to remote memory and (ii) minimizing the latency of such misses. There have been a plethora of distributed shared memory architectures designed both in industry and academia to address the above goals. These architectures fall into two general classifications: those that require modifications to the processor, memory controller, and network to integrate the DSM controller into the system [45, 11, 2, 37] and those that use strictly commodity workstations and networks [8, 39, 49, 17, 35]. The design of the DSM controller for these architectures has varied from using custom hardwired state machines to using programmable protocol processors.

The use of commodity workstations and networks has several advantages over designs that require proprietary components, including faster prototyping and improved portability of the design across already existing hardware platforms. However, the latency of remote memory access in DSM systems built using commodity workstations and networks tends to be higher since the use of commodity components prevents designers from tuning any of these components to provide faster and specialized access for DSM transactions.

Remote memory misses fall into two types: *essential* and *non-essential* [18]. Essential misses are inherent in the parallel application and cannot be avoided. For example, the cache miss that occurs when nodes try to increment a global counter falls into this category. Non-essential misses are caused by inefficient use of the cache and memory on each node or by employing non-optimal coherency protocols for consistency management. Reducing the number of non-essential misses is particularly important in architectures that use commodity components because of their higher remote access latency.

Software DSM systems generally employ large coherence units (pages or objects) and suffer from high communication and OS overheads due to their software implementations. This has led researchers to propose multiple protocols to match the applications communication requirement. Hardware DSM systems, with their ability detect access faults at cache line granularity, have support for small coherence unit and, thus, suffer

fewer remote access faults compared to software DSM systems and have relatively lower access latency for the faults that remain. However, non-essential misses still remain in hardware DSM systems. One question that I will investigate is whether it will be beneficial to support multiple access protocols in hardware DSM systems? If so, what features should be present in a DSM controller designed to be used with commodity workstations and networks? This thesis proposal addesses these questions.

I expect that the proposed research will show that with useful hints from the user or compiler, a DSM controller supporting multiple access protocols can significantly reduce the number of non-essential remote memory misses and thereby increase the speedup of parallel applications. Also, the proposed research will show that to reap this performance benefit, careful consideration has to be given to the design of DSM controller. The controller will have to have enough parallelism within it, known as *intra-operation parallelism*, through distributed state machines to reduce the waiting time for incoming requests and parallelism between it and other components, known as *inter-operation parallelism*, to avoid locking the entire controller while waiting for long latency operations.

The rest of this proposal is organized as follows. Section 2 gives background information related to distributed shared memory. Readers who are familiar with the research issues in DSM can skip this section. Section 3 details the factors that contribute to the problems above. Section 4 discusses the proposed solutions to these problems that will be explored as part of this research effort. Section 5 describes how the proposed solutions will be evaluated. Section 6 lists chronologically the events that will lead to the completion of the thesis. Section 7 compares and contrasts the research in industry and academia that have addressed the same problem.

2 Basic concepts

The main virtue of distributed shared memory is the abstraction of a global shared address space that it presents to parallel programs, which makes it easy for programmers to develop and reason about the parallel program. The parallel processes running across the machine exchange information by reading and writing to the global address space. The underlying distributed shared memory system is responsible for maintaining the consistency of the shared address space as different processes across the system access and modify it. In particular, the parallel programmer need not concern themselves with specifically identifying all communication that must be performed to solve the problem at hand.

2.1 Software DSM Systems

The concept of distributed shared memory system was first introduced by Kai Li [41]. Li's Ivy DSM system, a software *distributed virtual memory* (DVM) system, uses the

operating system's virtual memory support to implement consistency. Each node in the system has a *consistency manager* that maintains the global coherency with help of the *directory server*, which can be centralized or distributed. The directory server maintains two data structures to manage global coherency: the *state* of the pages in shared address space (e.g., *free*, *shared* or *exclusive*) and a *copyset* of the nodes that have cached the shared page in their physical memory. When a node accesses data in the shared address space that it does not currently have cached locally, it results in page fault. In response, the *consistency manager* on the node gets a *shared* or an *exclusive copy* of the page, depending on whether it was a *read* or *write* fault, after consulting the directory server.

The initial software DSM system implementations provided a *sequentially consistent* [36] view of the shared address space. Informally, in a sequentially consistent system, any write to an address by a process will be propagated to all the nodes holding a copy of the page before the writing process is allowed to continue. In a DVM, maintaining such a strict degree of consistency leads to stall of the processes writing to a page until the node can get an exclusive copy. Given the high latency of communication and virtual memory operations, this can lead to poor performance. Also, writes to a page result in invalidation of the page at the other nodes even though the other nodes are reading different parts of the page. Later DVM systems [6, 33], provided a weakly consistent view of data [29, 1, 21] to improve the performance of properly labeled programs. One such weak model, known as *release consistency*, requires the programmer to label the program with release and acquire points. These labels define the points at which the writes done by a process have to be visible at other nodes having copy of the page. In practice, this labeling requirement is trivially satisfied by exposing synchronization operations (e.g., lock and barrier operations). Providing a weaker consistency model improves performance for three reasons: (i) instead of getting stalled at every write fault, the process is allowed to continue until the release point, (ii) for update protocols, the DVM system is able to coalesce and send all the writes at the release point, and (iii) pages are not invalidated unnecessarily at other nodes or invalidation can be postponed until the next acquire point.

The directory server in conjunction with the consistency manager uses coherency protocol to maintain consistency of the global shared space. A number of coherency protocols [4] have been proposed by the research community. The coherency protocol of choice has been *write invalidate*. In a write invalidate protocol, a node wanting to write to a page requests for an exclusive copy. The directory server asks the nodes that have a *read only* copy to invalidate their copy and then supplies the *exclusive* copy to the requesting node. If a node wants a *read only* copy and another node has an *exclusive* copy, the directory server asks the node with *exclusive* copy to change it to *read only* mode and send back the page. Though this simple coherency protocol is able to achieve global consistency, it does not match the communication pattern of all the applications. For example if different parts of a page are being read and written by different processes, the *write invalidate* protocol results in unnecessary invalidations and read page faults. A variety of protocols have been developed to address this problem. In a *migratory protocol* [15, 51], a special case of write invalidate, a node treats a read page fault as

though it is a write page fault and gets an *exclusive* copy of the page instead of just a *read only* copy. In cases where reads are followed by writes, this eliminates a subsequent write fault. The advent of release consistency that was discussed in the previous paragraph resulted in a suite of coherency protocols [46, 12, 32] targeted for applications using the weak consistency model. All these protocols are derived from write invalidate and *write update* protocols. A write update protocol allows a node to modify its copy of the page without getting exclusive ownership. However, at the release point, all the modifications are forwarded to nodes that have copy of the page. In software DSM systems the use of release consistent write update protocols can achieve 25% to 100% speedup [12].

Software DSM systems are easy to port across various hardware and operating system platforms. However, they have shortcomings. The page grained coherency control limits the performance for parallel applications with fine grained communication needs. This problem is exacerbated when the application needs sequential consistency. Also, page level replication leads to waste of physical memory due to internal fragmentation. These shortcomings led to the development of hardware DSM systems, which are discussed in the next section.

2.2 Hardware DSM Systems

Each node in a hardware DSM system has a DSM controller that is responsible for maintaining global consistency. Being in hardware, the granularity of consistency maintained by the DSM controller can be as small as a processor cache line. The function of the DSM controller is similar to that of the software DVM except that access control is triggered by misses to the processor cache instead of read and write page faults. Section 2.2.1 discusses the various memory architectures that are supported by hardware DSM systems (Figure 1) and Section 2.2.2 details the different approaches used in the implementation of DSM controllers.

2.2.1 Memory Architectures

The *cache coherent non-uniform memory access* (CC-NUMA) architecture [40, 2] has been the most popular amongst the hardware DSM systems that have been proposed or built. In a CC-NUMA system, the DSM controller at a node is known as the *home* for global memory stored in the physical memory at that node. When there is a cache miss, the DSM controller looks at the global physical address to determine whether it is to local memory or to remote memory. For misses to local memory, the DSM controller takes the required actions depending on the state of the line and copyset, e.g., supplying the data directly or invoking the coherency protocol if the data is cached remotely in a mode inconsistent with the type of access. Remote memory requests are forwarded to the home DSM controller. The cache line returned by the home node is supplied to the SRAM cache. The SRAM cache, being small, limits the amount of memory available at each node for replication of global memory. This is one of the main drawbacks of a

CC-NUMA system.

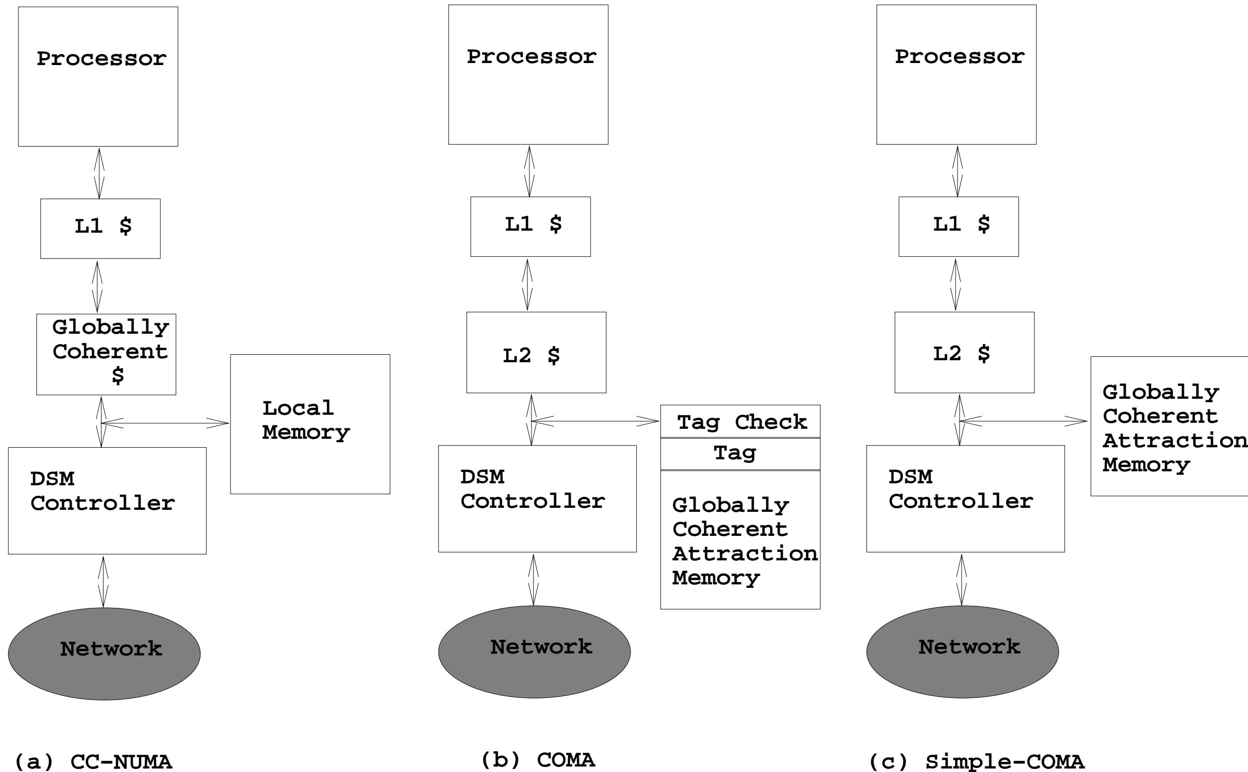

Figure 1: Memory Architectures

In *cache only memory architectures* (COMA) systems [24], each node's DRAM is divided into blocks and used as a cache of the global memory. Since the amount of DRAM in a node typically dwarfs the amount of SRAM, far more global memory can be cached on a COMA node than a CC-NUMA node. When a node accesses a block in shared address space, the DSM controller does a lookup in a memory tag. On a miss in the COMA memory tag, the block is replicated or migrated to the node's physical memory depending on whether it is a read or write miss. Since there is no fixed home node for each block, the DSM controllers manage the coherency using a distributed coherency protocol. The complex tag lookup and coherency management hardware and their related performance hits are the main drawback of the COMA architecture. Also, the replacement traffic in the COMA systems can lead to poor performance when little or no DRAM is available for replication.

Simple-COMA (SCOMA) systems [50, 49] combine the best of the DVM and the COMA systems to use DRAM for remote memory replication. When a node accesses a page, an SCOMA machine allocates pages in local physical memory for replication, as is done by a DVM system. However, instead of fetching the whole page, the SCOMA DSM controller fetches each individual block from the home node on a cache miss. In essence, page grained memory allocation is done in software, as in a DVM, while line grained replication and coherency is done in hardware, as in a CC-NUMA machine.

SCOMA provides most of the benefits of COMA without incurring as much complexity or performance hit at the memory controller. However, page grained allocation can lead to inefficient use of memory and poor performance when memory becomes scarce.

2.2.2 Controller Implementation

The rigidity of a DSM controller designed using custom state machines has lead researchers to design flexible DSM controllers. Such a design has three advantages: (i) the protocols supported by the controller are not fixed at implementation time, (ii) it allows applications to program the controller to use coherency protocols well suited for the application, and (iii) it allows for workarounds for bugs encountered in hardware after implementation.

The design of flexible controllers has taken three different approaches. In the first approach, the processor is interrupted by the directory server part of the DSM controller and the interrupt handler is used to achieve the flexibility. This, however, requires tight integration with the processor and fast interrupt path to keep the overhead low [13]. The next two approaches use a separate generic processor to do protocol processing. One such approach interrupts the protocol processor to carry out fine grained access control [48]. The other approach assigns identifiers to each message, including access faults, and dispatches them to a special handler based on the identifier [25].

3 Problem Statement

The use of commodity components in building hardware DSM machines is being espoused by various researchers since it reduces the cost of building a scalable parallel cluster of workstation multiprocessor [40, 35, 49]. However, the use of commodity components can lead to increased latency. Thus, reducing the number of misses to remote memory to be only those that are *essential* becomes paramount in such systems [23, 18, 53, 19]. To this end, the DSM controller has to support multiple access protocols to match the applications communication behavior. In order to reap the benefit of multiple access protocols, the implementation of DSM controller should keep the side-effect of increased occupancy to a minimum to avoid increasing remote access latency. The following sections address these problems in more detail.

3.1 Reducing the Number of Remote Memory Misses

Misses to remote memory can be classified into three categories: *cold*, *capacity* and *coherency*. A miss while accessing the block for the first time is termed a *cold* miss. Subsequent misses to the same block fall into the other two categories. If the miss occured because the block was evicted from the node due to lack of space or a conflict, it

is known as a *capacity* miss. A miss after the block was invalidated by some other node is known as a *coherency* miss. Cold and capacity misses that are caused by inefficient use of the memory available at a node and coherency misses that are caused by using the wrong coherency protocol are said to be *avoidable* [23, 18, 53, 19]. The next two sections discuss these factors in more detail.

3.1.1 Cold and Capacity Misses

A capacity miss occurs when all the remote data accessed as part of the working set of the application cannot be accommodated locally. This happens because of a lack of memory for caching remote data. The problem is more serious in CC-NUMA systems because the size of the expensive SRAM cache that is used to cache remote data is limited. COMA systems, by using cheap and large DRAM memory to cache remote data, are able to overcome this problem. However, COMA systems have not been popular because of the complexity they introduce to the memory controller. SCOMA systems reduce the cost of capacity misses in the SRAM cache by allocating pages in the DRAM to cache remote data. But, the capacity problem can quickly shift to DRAM memory if the application access pattern leads to waste of DRAM memory with internal fragmentation. When that happens, the system can run out of DRAM memory resulting in *expensive* capacity misses because of the overhead of evicting pages from the DRAM.

3.1.2 Coherency Misses

Data objects accessed and manipulated by the parallel processes can be classified into the following four categories: *read only*, *migratory*, *mostly read and occasionally written* (MROW), and *simultaneously read and written* (SRW) [55]. *Migratory* objects migrate from one node to another and at each node they are read and written. For example, objects that are read and written inside a critical section fall into this category. MROW objects are read by one or more processes most of the time, but at times are written and manipulated. After the write, the object might be read by the same or a different set of processes. SRW objects are ones to which two or more processes read and write simultaneously. An instance of this is when different parts of an array are read and written by different processes.

The mismatch between the usage of a data object and the protocol employed by the hardware DSM to keep it consistent can lead to an increase in the number of coherency misses. For example, employing a *write invalidate* protocol for *migratory* object results in unnecessary write misses. Using a *write invalidate* protocol for SRW objects leads to an increase in read misses. Thus, the *write invalidate* protocol is appropriate for *read only* and *MROW* objects. The *migratory* protocol, which fetches an exclusive copy of the block on a read miss is better for *migratory* objects. The *write update* protocol works well for SRW objects. In contrast, using a *write invalidate* protocol for SRW objects can lead to an increase in reads and invalidations for such objects because of *false sharing* [6, 19].

Similarly, the number of update messages increases when a *write update* protocol is used for MROW objects.

The problems arising out of coherency mismatch have been addressed in software DSM systems and solutions have been proposed [5]. However, designers of hardware DSM systems have not looked into ways of adopting these solutions. One of the reasons they have not is that the adverse effect due to protocol mismatch is small when the granularity of the coherent block is small (e.g., a 32 byte cache line). Lately, however, the coherent block size employed by hardware DSM systems has increased to 128 bytes for three reasons: (i) to keep up with the second level cache line size, (ii) to keep down the memory overhead of directory structure, and (iii) to amortize the latency overhead in commodity networks. With this increase in block size, the problems faced by the software DSM systems in terms of coherency misses become relevant to the hardware DSM systems too.

3.2 Reducing Remote Access Latency

The design of the DSM controller plays a crucial role in the remote memory access latency. A custom hardwired state machine implementation reduces the time spent in local and remote controllers while satisfying a miss. However, if the design is rigid, it leads to increased misses as discussed in Section 3.1.1 and Section 3.1.2. There are other approaches used to reducing remote access latency such as developing special purpose processors [3], networks [37], memory controllers [45], or prefetching [57]. However, they are beyond the scope of the proposed research.

A controller built around a protocol processor provides a level of adaptability that software can use to reduce the number of misses. However, protocol processors based on generic processors take more cycles than custom state machines to do the same task. This results in an increase in the remote memory access latency for two reasons [26, 44]: (i) the time spent in local and remote DSM controller to handle the miss increases, and (ii) during high contention, the increased occupancy in the controller leads to an increase in the waiting time at the local and remote DSM controller. Protocol processors are able to reduce this effect to some extent by overlapping protocol processing with other operations such as memory read [35]. However, not all coherency messages have operations that can be overlapped.

4 Proposed Research

This section discusses the approach that is being investigated to solve the problems that were discussed in Section 3 and the rationale behind the approach. An architecture that supports multiple access protocols and whose implementation has reasonable parallelism is proposed.

Figure 2 shows the configuration at each node in the proposed implementation. The system consists of commodity SMP nodes connected using a commodity network. The DSM controller plugs into one of the processor slots on the coherent bus. It is connected to a small SRAM that acts as an inclusive CCNUMA cache and is used for staging data between memory and the network. The DSM controller DRAM is used to hold directory and other state information.

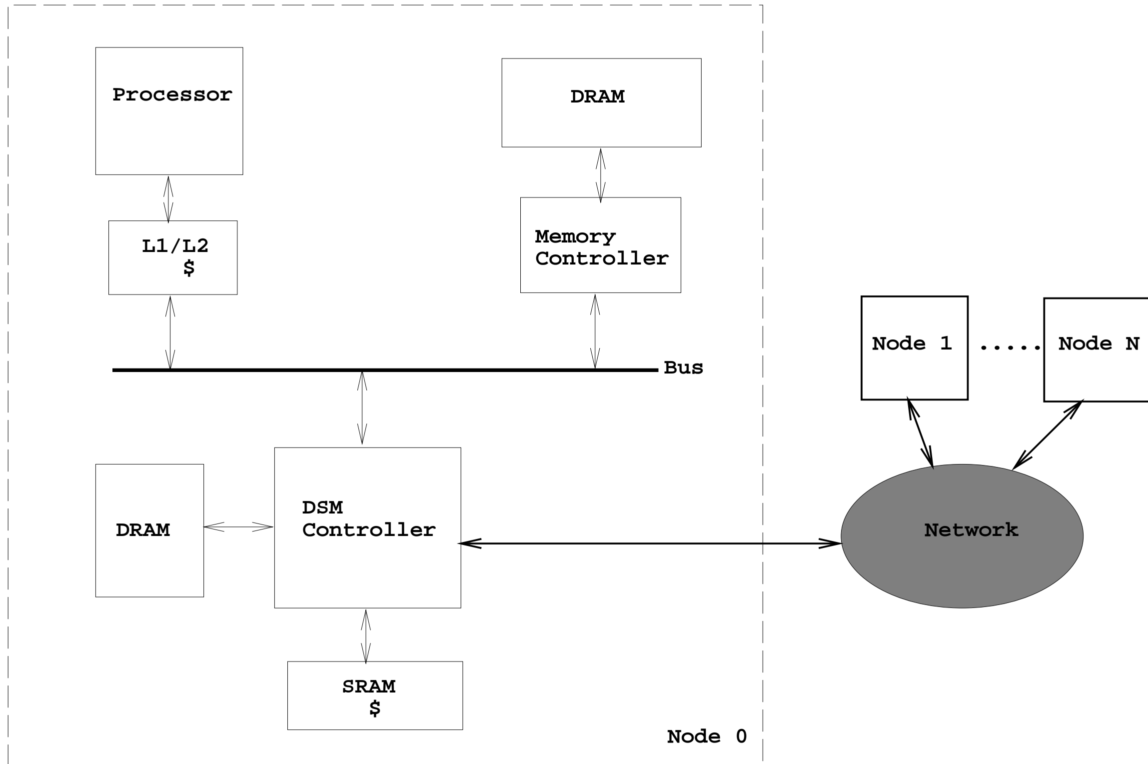

Figure 2: Node Configuration

One of the unique features of the proposed system is that it enables the compiler or application, to choose an appropriate access protocol for each shared memory page in the application to keep the remote memory misses to minimum. For example, one page might be mapped in CC-NUMA mode using a sequentially consistent write invalidate protocol, while another might be mapped in SCOMA mode and employ a release consistent write update protocol. Section 4.1 details the proposed solution to reduce the number of misses to remote memory and Section 4.2 describes a proposed solution targeted to reduce remote memory access latency.

4.1 Remote Memory Miss

Section 4.1.1 discusses the architectural features that reduce the number of non-essential *cold* and *capacity* misses. Section 4.1.2 details how the architecture can be made to

adapt to application needs and use different coherency protocols to reduce the number of non-essential coherency misses.

4.1.1 Combined Memory Model

The SCOMA model, with its huge DRAM cache, performs well when the memory pressure [1] is low because it eliminates most of the capacity misses that would have occured in a CCNUMA model. However, as discussed in Section 3.1.1, the SCOMA model can lead to page thrashing at high memory pressure. By supporting a combined CCNUMA and SCOMA memory architecture, the relative advantages and disadvantages of these two models can be used to complement each other to achieve a balanced system that fares well under both high and low memory pressure.

4.1.2 Multiple Coherency Protocols

The number of coherency misses that result when the protocol supported by a machine does not match the application's access pattern will be reduced in the proposed system by supporting variants of three protocols: *migratory*, *write invalidate*, and *write update*. The application or the compiler can specify the protocol to use on a per page basis and the DSM controller will use that protocol. Though software DSM systems have employed a suite of special purpose protocols, only three protocols are considered in the proposed architecture for the following reasons: (i) most of the other protocols are extensions of *write update* [7], and (ii) supporting more than three in hardware controller would be expensive.

One of the unique features of the proposed architecture is its support for a write update protocol using commodity components. The usefulness of an update protocol in hardware DSM has been shown by several researchers [22, 30, 47, 31, 38], but, these systems did not use commodity components. An update protocol can lead to high latency during synchronization and increase network congestion. A unique feature of the proposed implementation will be its support for *release state buffer* (RSB) to cull synchronization overhead and network congestion. The RSB is a table and each entry in the table contains the physical address and clean copy of the block that is being modified. The number of entries in the table limits the number of blocks that can be modified simultaneously at a node. When the number of blocks being modified is above a threshold, a victim block is selected in *least recently modified* order. The modifications done to the block by the local processor are sent to the home node. The RSB is expected to improve the performance of the write update protocol: (i) by avoiding having to walk all the blocks looking for modified blocks, (ii) by reducing the latency during synchronization

[1] *Memory pressure* is the inverse of the percentage of physical pages in free page pool. When there are few pages in the free page pool, the memory pressure is high. It is low when there are many free pages. For example, if a node has sixteen physical pages, four of which are free, memory pressure is said to be 75%.

by limiting the number of blocks that needs to be flushed, (iii) by coalescing modifications, (iv) by maintaining a *least recently modified* order on the updates sent, and (v) by leveling the update traffic originating from a node between two synchronization points. Another problem with a write update protocol is the propagation of useless updates [22]. They increase network traffic and unnecessarily invalidate the updated blocks from processor's cache. To mitigate these two problems the proposed controller has two features that are derived from previous research [42, 22]: (i) an *acquire state buffer* (ASB), and (ii) *write update counter* per block. The ASB holds the updates received and delays the posting of the updates. This helps in coalescing of updates to the same block from different remote nodes and also reduces the number of invalidations on the bus. The update counter helps to keep track of the number of updates that have been received without the local processors accessing the block. When the counter reaches a threshold, the block is invalidated and the DSM controller at the home node is informed not to send any more updates. This is expected to reduce the number of useless updates.

4.2 Remote Access Latency

The controller occupancy problem that aggravates remote memory latency will be addressed in the proposed research by designing a DSM controller with the following features:

- Divide the controller to two parts to handle local and remote requests separately.
- Design each part using distributed state machines.
- Make each part non-blocking while waiting for long latency operations.

The division of the controller into two parts and building them with distributed state machines provides parallelism and pipelining to reduce the delays during high contention. The non-blocking feature avoids stalling of the controller due to a long latency external event such as memory read. Figure 3 shows the internals of the DSM controller.

The controller is divided into two components: an *access controller* (AC) and a *directory controller* (DC). The AC handles misses and maintains the coherency of blocks accessed by the local processor. The DC on a node handles the global coherency of blocks *homed* at the node. The functionality of the AC and the DC are similar to the consistency manager and the directory server in software DVM systems respectively. Apart from reducing occupancy, the split avoids penalizing local memory accesses when the DSM controller is heavily loaded. The AC and the DC consist of multiple state machines, as shown in Figure 3, to increase the throughput of the controller and reduce the latency of transaction processing during contention. Any action that requires a large number of cycles to complete, such as a read from memory is performed using a split transaction protocol. This avoids blocking any state machine in the controller while handling longstanding actions and allows the controller to handle more than one coherent

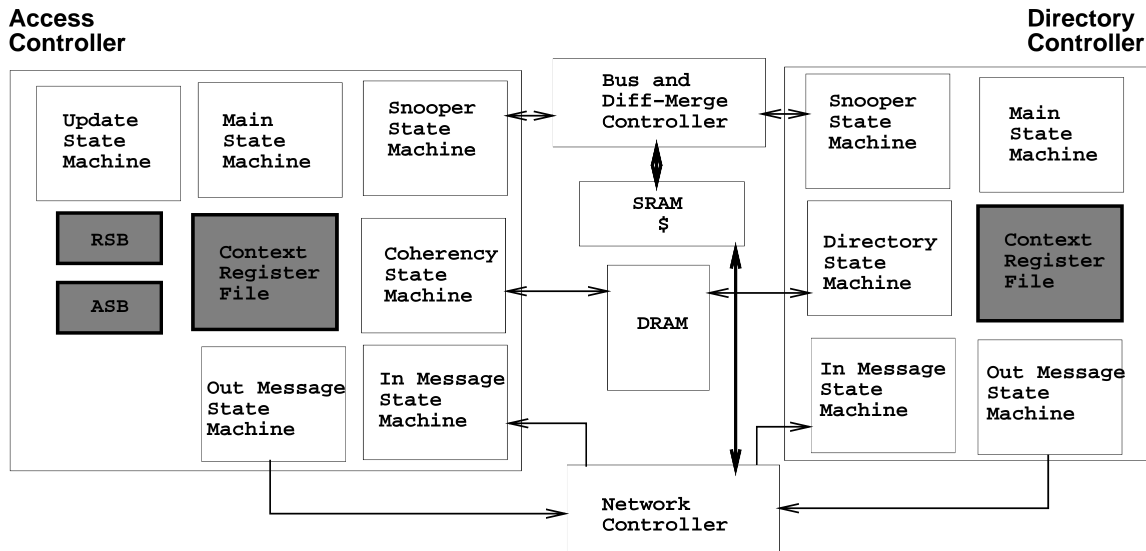

Figure 3: DSM Controller

transaction simultaneously. Each register in the *the context register file* (CRF) holds information about a in-process transaction. For example, when the DC receives two read requests for different blocks, two slots in the CRF are allocated to store information related to request while the DC is waiting for memory reads to complete.

5 Evaluation Method

Evaluation of the proposed system will be done via detailed simulation of the HP PA-RISC architecture, including an instruction set interpreter and detailed cycle accurate simulation modules for the first level cache, the system bus, the memory controller, the network interconnect, and the DSM controller. The simulation will be based on the following commodity components:

- *Processor*: Hewlett-Packard's (HP) PA-8000 processor [28] is modeled by the execution driven PAINT [52] simulator. PAINT does not model the deep pipeline, aggressive instruction lookahead or multiple-issue features of the processor. However, it does model the stall-on-use property of the PA-8000 processor and the aggressive non-blocking first level cache.
- *Coherent Bus*: HP's split-transaction Runway bus [10], which supports a restricted MOSEI protocol, will be used.
- *Main Memory Controller* (MMC): The MMC is modeled on the current HP workstations [27].
- *Network*: The network is modeled on the Myrinet network [9].

A simulation of all the above components has already been developed to evaluate the *Avalanche* multiprocessor prototype [17]. However, the protocol restrictions imposed by HP's Runway bus, MMC, and processor cache make it difficult to support an efficient combined memory model, multiple coherency protocols, and multiple processors per node [34]. Hence, the simulation models of these components will be suitably extended to support the proposed system.

The SPLASH-2 benchmark programs [56] will be ported and used in the evaluation. Other benchmarks, such as EM3D [16], will be ported and evaluated if found suitable. Also, to provide an insight into the *best case* performance, synthetic benchmarks will be developed and analyzed.

One of the primary metrics used to evaluate the performance of the proposed hardware DSM system is the speedup achieved during the parallel phase of the benchmark programs. Speedup of the system will be evaluated with 2 to 16 nodes. In addition to speedup, following other metrics will be analyzed to evaluate the novel features supported in the proposed system and their effect on the speedup:

- *memory stall*: The total stall time of each application while accessing memory.
- *synchronization stall*: The total stall time of each application due to synchronization.
- *remote miss*: The number of cold, capacity, conflict, and coherency misses to remote memory suffered by the program.
- *controller occupancy and contention*: The occupancy will be measured by the number of transactions handled by the controller and the total number of cycles the controller is busy. The contention will be measured by the number of cycles each transaction is stalled at the input port of the controller.
- *network occupancy and contention*: The occupancy will be measured by the number of bytes transferred by the network and the total number of cycles the network is busy. The contention will be measured by the number of cycles each transaction is stalled at the input port of the network.

The proposed system incorporates various novel features, such as the combined memory model, and multiple coherency protocols, to reduce the number of remote memory misses. The distributed, non-blocking implementation of the DSM controller will reduce the remote memory latency. To allow a fair analysis of the cost-benefit trade-off, the performance evaluation will be done in following steps. In each step, a new feature will be introduced into the system and the system will be compared and contrasted against the system in the previous steps. In all these steps, the commodity components described above will be used and all of the above performance metrics will be reported.

1. The performance will be evaluated by restricting the DSM controller to operate as a controller built using a single blocking state machine. The controller will support

only the CC-NUMA memory model and a sequential consistent write invalidate coherency protocol. This model will serve as the baseline against which all other system variants will be compared.

2. The DSM controller will be split into two parts: the AC and the DC. However, the AC and the DC each will continue to operate as a single blocking state machine. The performance gain achieved by separating the DSM controller into two components will be evaluated.

3. Distributed state machines and support for non-blocking operation will be added to both the AC and the DC. The performance gain will be determined.

4. The controller will be extended to support a combined memory model. The gain in performance by the combined memory model will be evaluated.

5. The DSM controller will be extended to support migratory and write update protocol in addition to the write invalidate protocol. The performance gain by adapting the memory controller to use the correct protocol for each page will be evaluated.

The performance of the system in step 3 approximates to some extent the performance of a DSM controller built using custom state machines that supports a single protocol. The performance of the final system in step 5 will be re-evaluated by restricting the DSM controller to a single state machine. The performance of such a system will approximate the performance of a system built using a protocol processor with support for multiple protocols and memory models [2].

6 Timetable

Table 1 gives the tentative completion dates of various tasks of the proposed research. Most of the simulation code required to model the processor, bus, MMC, and network is complete. However, some of it must be extended. The DSM controller developed for the Avalanche prototype supports only an SCOMA memory model. This model will be extended to support a combined memory model and an efficient implementation of three coherency protocols. The implementation of the DSM controller will be completed by the end of spring quarter. In the summer quarter, the evaluation of the system will be completed. The writing of thesis will be completed by the end of fall semester. The thesis defense will be held in the end of the fall semester.

[2]Something possible but not currently implemented in systems with protocol processors.

Task	Finish date
SCOMA implementation	done
CC-NUMA implementation	June 30, 1998
Baseline model evaluation	August 30, 1998
Split DSM controller evaluation	September 15, 1998
Distributed and non-blocking controller evaluation	September 30, 1998
Combined memory model evaluation	October 15, 1998
Multiple coherency protocol evaluation	October 30, 1998
Thesis writing	November 30, 1998
Thesis Defense	December 16, 1998

Table 1: Completion dates of various tasks of the research

7 Related work

There are a number of past as well as ongoing efforts in the area of hardware DSM architectures. This section details these research efforts. In order to compare and contrast these efforts with proposed research, emphasis is placed on how the DSM controller is designed, whether they use commodity components and what protocols and memory models are supported.

The MIT Alewife machine [13, 14] was one of the first machines to implement directory based shared memory. It was also the first hardware-based DSM system to use software for protocol processing. Alewife supports a sequentially consistent write invalidate protocol and the CC-NUMA memory model. The DSM controller is built using custom state machines and is tightly integrated with the processor core. The processor core is also modified to support special instructions and fast interrupt path. This ability to trap into software could also be used to support multiple consistency protocols, but the Alewife designers limit its use to extending directory pointers and to providing a limited number of specialized synchronization operations.

The Stanford DASH multiprocessor [40, 39] 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. DASH used a cluster of processors connected through a mesh interconnect. DASH was the first DSM system to use a commodity processor, bus, memory controller and network. The DSM controller was designed using custom state logic and supported both sequential and release consistent write invalidate protocols and a CC-NUMA memory model.

The Stanford FLASH machine is a second generation DASH multiprocessor that offloads protocol processing to a processor situated on the MAGIC chip. The MAGIC chip resides between the processor and the memory controller. It must mediate every memory reference on the system bus, as well as handling message passing traffic and distributed memory coherency maintenance. 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. Except for the special interface to the memory controller, the FLASH system uses commodity components. However, how the FLASH model can be adopted to SMP nodes is not known. Even if it can be adopted, the occupancy of the controller becomes a bottleneck [44]. What protocols and memory models are supported using the flexible feature of the controller and how they perform is not yet known.

User level shared memory in the Tempest and Typhoon systems [49] support cooperation between software and hardware to implement both scalable shared memory and message passing abstractions. Tempest provides an interface that allows user level code to efficiently utilize the underlying low-level communication and shared memory mechanisms. Typhoon is a proposed hardware implementation for this interface that contains a fully programmable, user-level processor. Like FLASH, the proposed system uses low level software handlers to provide flexibility. The system uses commodity components. This is one of the systems that supports a memory architecture similar to SCOMA called *stache*. Recently the Tempest and Typhoon system was extended to support a combined memory model called *Reactive NUMA* [20]. The system maintains a *refetch counter* per page to determine hot CC-NUMA pages. The hot pages are mapped to SCOMA mode after allocating a page in the DRAM. However, the remapping is supported only from CC-NUMA to SCOMA mode. Once there are no more pages in the free pool, the system stops adopting. Also, like FLASH, the system suffers from the same occupancy problem by using a protocol processor for flexibility.

The SHRIMP Multicomputer [8] 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 packaged into a message and forwarded to the remote node, where it is placed into main memory. SHRIMP relies on a write through cache to implement the write update protocol.

The KSR-1 [11] was the first commercial "scalable" shared memory multiprocessor based on the COMA architecture. It employs a hierarchical directory scheme with each level of the hierarchy implemented using a ring interconnect. In order to avoid the problem of finding a node when an exclusive line is replaced, the KSR-1 has a fixed *home page* for every cache line that acts as a place holder. This differentiates it from traditional COMA architectures. The KSR-1 was built using a custom processor and all the protocols are in hardware.

The S3.mp multiprocessor system [45] was developed with a 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 that 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. The S3.mp divided the DSM controller into two engines one for local requests and another for remote

requests. The engines contained programmable micro-controllers. This programmable feature was used to support both CC-NUMA and SCOMA memory models. However, the system has not provided a transparent combined model to the programmer as is done by the proposed system and Reactive-NUMA systems.

8 Conclusion

The use of commodity components in building hardware DSM systems is becoming popular since it promotes faster design cycles and makes the system less expensive. However, remote memory latency in such systems is high due to the inherent latency of the commodity components. So, reducing the number of remote memory misses becomes critically important in such systems. The proposed research will prove that a hardware distributed shared memory architecture that supports multiple access protocols will significantly reduce the number of remote misses and increase the performance of scalable DSM machines based on commodity components. This research will also show that it is important provide enough parallelism and keep the occupancy to minimum in the DSM controller for such an architecture to retain the performance benefits achieved with reduced number of misses.

References

[1] S. Adve and M. Hill. Weak ordering: A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] 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.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, September 1990.

[4] 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.

[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] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 168–176, March 1990.

[7] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory using multi-protocol release consistency. In A.I. Karshner and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] J.B. Carter. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, September 1995.

[13] 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.

[14] 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.

[15] 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.

[16] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in spit-c. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[17] A. Davis, M. Swanson, E. Brunvand, J. Carter, L. Stoller, and R. Kuramkote. Avalanche: Communication and memory architectures for scalable parallel computing. In Preparation, 1996.

[18] M. Dubois, J. Skeppstedt, L. Ricciulli, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 88–97, May 1993.

[19] S.J. Eggars and T.E. Jeremiasses. Eliminating false sharing. In *1991 International Conference on Parallel Processing*, pages 377–381, August 1991.

[20] B. Falsafi and D.A. Wood. Reactive NUMA: A design for unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

[21] 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.

[22] H. Grahn, P. Stenström, and M. Dubois. Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, 11(3):247–271, June 1995.

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

[24] E. Hagersten, A. Landin, and S. Haridi. DDM – A cache-only memory architecture. *IEEE Computer*, 25(9):241–248, September 1992.

[25] 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.

[26] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Stanford University, 1995.

[27] 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.

[28] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *COMPCON '95*, pages 123–128, 1995.

[29] P.W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 302–311, May 1990.

[30] L. Iftode, C. Dubnicki, E. Feltin, and K. Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the Second Annual Symposium on High Performance Computer Architecture*, pages 26–37, February 1996.

[31] A. Karp and V. Sarkar. Data merging for shared-memory multiprocessors. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, January 1993.

[32] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency.* PhD thesis, Rice University, 1994.

[33] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[34] R. Kuramkote, J. Carter, A. Davis, C. Kuo, L. Stoller, and M. Swanson. Analysis of avalanche's shared memory architecture. Technical Report UUCS-97-008, University of Utah - Computer Science Department, July 1997.

[35] 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.

[36] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[37] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *SIGARCH97*, pages 241–251, June 1997.

[38] J. Lee and U. Ramachandran. Architectural primitives for a scalable shared memory multiprocessor. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 103–114, Hilton Head, South Carolina, July 1991.

[39] 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.

[40] 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.

[41] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[42] 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.

[43] M. Marchetti, L. Kontothonassis, R. Bianchini, and M.L. Scott. Using simple page placement policies to reduce the code of cache fills in coherent shared-memory systems. In *Proceedings of the Ninth ACM/IEEE International Parallel Processing Symposium (IPPS)*, April 1995.

[44] M. M. Michael, A. K. Nanda, B.H. Lim, and M. L. Scott. Coherence controller architectures for SMP-based CC-NUMA multiprocessors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 219–228, June 1997.

[45] 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.

[46] U. Ramachandran and M. Ahamad. Programming with distributed shared memory. In *Proceedings of COMPSAC '89*, September 1989.

[47] A. Raynaud, Z. Zhang, and J. Torrellas. Distance-adaptive update protocols for scalable shared-memory multiprocessors. In *Proceedings of the Second Annual Symposium on High Performance Computer Architecture*, pages 323–334, February 1996.

[48] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 44–55, May 1996.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] J. Torrellas, M.S. Lam, and J.L. Hennessy. Shared data placement optimizations to reduce mutliprocessor cache misses. In *1990 International Conference on Parallel Processing*, pages 266–270, August 1990.

[54] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[55] W.-D. Weber and A. Gupta. Analysis of cache invalidation patterns in multiprocessors. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[56] 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.

[57] Z. Zhang and J. Torellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 188–199, June 1995.