Computer Systems Laboratory
Cornell University, Ithaca, NY 14853

Providing Hardware DSM Performance at Software DSM Cost

M. Heinrich and E. Speight

Technical Report No. CSL-TR-2000-1008

November 2000

Providing Hardware DSM Performance at Software DSM Cost

Mark Heinrich and Evan Speight
Computer Systems Laboratory, Cornell University, Ithaca, NY 14853

Abstract

Emerging trends in commodity network technology coupled with key insights from academic research in active memory systems are leading toward the realization of hardware DSM on commodity clusters. We call the result of this convergence active memory clusters. After discussing the current state of the art in hardware DSM, clusters, and software DSM architectures, we highlight the key differences between hardware and software DSM systems and show how these differences are rapidly disappearing in commodity systems—with the notable exception of the specialized memory controller present in hardware DSM systems. We then discuss recent research results in active memory systems that show that uniprocessor performance can be improved with the inclusion of an active memory controller, indicating that it may become part of forthcoming commodity workstations. We make the observation that active memory support can be treated as an extension of the cache coherence protocol, and that an active memory controller contains the necessary functionality for building a hardware DSM machine. Coupled with enhancements in network technology and a small amount of software support, active memory clusters can achieve hardware DSM performance at software DSM cost.

1. Introduction

With the advent of low-cost, high-performance commodity components, networks of industry-standard workstations have captured the interest of both industry and research institutions over the past several years. Referred to as NOWs (network of workstations), COWs (collection of workstations), COPs (collection of PCs), etc., these *clusters* are typically comprised of symmetric multiprocessor (SMP) nodes containing two or four processors, large amounts of local memory (on the order of 1-4GB of RAM), and a network such as Myrinet, cLAN, or ServerNet that provides zero-byte latencies less than $10\mu s$.

When taken as a whole, the machines comprising a cluster make up a distributed memory parallel machine, indicating that message passing should be the natural choice for parallel programming on clusters. However, each individual node already ensures that data exchanged between co-located processors is cache-coherent. Thus, on a single node, an application developer may use the shared-memory programming model to achieve higher performance than sending messages between processors located on the same node.

An ideal situation is to provide shared-memory parallel programming on the cluster as a whole, not just on the individual nodes. This combines the ease of shared-memory programming with the cost advantage of a cluster-based distributed memory architecture. Many such *Distributed Shared Memory* (DSM) systems have been designed. Some of these systems provide the shared-memory abstraction in software, adding little or no cost to the cluster price but providing poor performance in many cases. Other systems provide shared memory in hardware, which can add substantial cost to each individual node in exchange for higher performance on parallel applications. The inclusion of the hardware required to support DSM typically provides no improvement in uniprocessor/single SMP performance, and therefore has never been included in commodity cluster components. In this paper, we argue that emerging technology trends in industry coupled with key insights from academic research are leading toward the realization of hardware DSM on commodity clusters at software DSM cost, with a concomitant improvement in single-node performance. We call the result of this convergence *active memory clusters* (AMC).

In Section 2, we describe background material relating to the design of both hardware and software DSM systems, as well as the rise in popularity of clusters of commodity workstations. Section 3 explains the key differences between software and hardware DSM systems, and how recent developments and research results can have narrowed the gap between the two. Section 4 presents the design of active memory clusters and discusses the architectural and operating systems issues that must be addressed for AMC to become a reality. Section 5 discusses the performance of AMC relative to that of hardware DSM and software DSM systems, and Section 6 concludes the paper.

2. Background

We begin by summarizing the development of hardware cache-coherent shared-memory systems, clusters of workstations, and the rise of software DSM techniques to attempt to bridge the gap between the two. In Section 3 we will then take a closer look at the differences between hardware and software DSM machines.

2.1. Hardware DSM

Scalable cache-coherent distributed shared-memory (DSM) machines have received much attention in the literature since the late 1980s. To demonstrate their effectiveness, several cache-coherent non-uniform memory access (CC-NUMA) *hardware DSM* machines were built in the research community (e.g. DASH [26], Alewife [2], FLASH [22], Typhoon [36]) and commercial machines followed (e.g. SGI Origin 2000 [23], Sun S3.mp [33], Sequent NUMA-Q [29], HP Exemplar [1], Data General Aviion [7]). At the same time, a large research effort produced a set of scientific benchmarks with which to evaluate DSM machines [48].

Most high-performance hardware DSM machines have tightly-integrated node or memory controllers that connect the microprocessor both to the memory system and to a proprietary high-speed switching network. The scalable coherence protocols (e.g., [6,14,27,38,41,42]) used in such machines are implemented either in hardware finite-state machines or in software running on an embedded programmable device in the controller. Despite the resulting high performance of these systems, and efforts to show that the necessary additional hardware to support hardware DSM in commodity workstations and servers is small [25], high-end PC servers and engineering workstations have yet to integrate the additional functionality needed to build seamless hardware DSM from COTS (commodity off-the-shelf) components.

2.2. Clusters

The ability of current microprocessors to directly support symmetric multiprocessing for two or four processors has created a “sweet spot” of low-priced SMP workstations that are rapidly becoming the commodity PCs of the enterprise environment. The low cost of these machines has led to their use in clustered compute farms consisting of high-end servers or workstations interconnected by a high-speed network such as Myrinet or the Virtual Interface Architecture that deliver latencies on the order of $10\mu s$ for inter-node communication. A “virtual parallel machine” constructed from such a cluster provides a distributed memory machine, with each node containing a complete version of the operating system, its own memory hierarchy, and I/O subsystem independent of other nodes in the cluster. Thus, message passing APIs such as MPI [31] or PVM [8] are the natural choice for cluster-based parallel programming.

However, current clusters based on SMP nodes actually present the application programmer with two distinct memory interfaces: shared memory between the processors local to each machine, and distributed memory between processors that are not co-located. The programmer is thus faced with a dilemma. There are clear performance advantages to using the native SMP load/store interface for communication between co-located processors, but accesses to remote memory require an entirely different communication model. Utilizing two different communication models in the same program is problematic, particularly in the case where the number of threads and processors per machine is not known at compile time.

2.3. Software DSM

In an attempt to remove the cost limitations associated with scalable DSM computing and address the programming concerns of cluster-based parallel computing, several *software DSM* systems have been built that do not rely on specialized hardware to provide programmers with shared memory. These systems include Ivy [28], TreadMarks [21], Munin [4], Brazos [43], CRL [18], MGS [49], CVM [20], Blizzard-S [40], Shasta [39], Cashmere-2L [46], and SoftFLASH [9]. The underlying principle in these machines is to leverage commodity parts—particularly the use of commodity processors, node boards, networks, and operating systems—to build a scalable DSM machine. Most software DSM systems rely on trapping protection violations and executing the coherence protocol in software running on the main microprocessor. These systems must use page-level granularity to enforce coherence, which also allows the high cost of communication to be amortized over the larger coherence unit. Much work into reducing the amount of communication necessary to maintain coherence has been reported, the most important being the use of multiple-writer protocols [4] and relaxed consistency models [4,21]. Other software DSM systems instrument application code to check for coherence actions that need to be performed before each access to shared memory (e.g., the Shasta system [39]). These systems can then implement coherence at any granularity desired, but the high handler overhead and the fact that the network is typically integrated on the I/O bus rather than the memory bus still results in the choice of pages for data transfer. Subsequently, such systems still incur large software overheads compared to hardware DSM systems.

Several software DSM systems [16,44] have attempted to address the programming problems associated with the mixed-architecture presented by a cluster of SMP machines by extending the OpenMP programming model used in SMP programming to encompass entire clusters. These software DSM systems use the available cache-line coherence provided in each SMP node to maintain coherence between co-located threads, while only invoking the software DSM handlers when inter-node communication is necessary. Although alleviating the amount of expensive communication to some degree, high synchronization rates, frequent sharing, or large amounts of false sharing severely hinder the performance of software DSM systems. As a result, their performance remains poor compared to their hardware DSM counterparts [3,9]. Still, the cost advantages of software DSM clusters make them a viable alternative for certain applications.

3. Differences Between Hardware DSM and Software DSM

To the naïve eye, a physical comparison of a hardware DSM machine like the FLASH multiprocessor with a modern software DSM system based on clusters reveals few differences. Both machines are constructed out of individual commercial boxes (FLASH uses Origin 200's as its nodes with a modified motherboard to hold its specialized node controller, while Brazos uses high-end SMP PCs such as Compaq Proliant 6400's) connected together with proprietary high-speed networks (FLASH uses SGI Craylink [10] and Brazos makes use of the cLAN architecture produced by Giganet). In fact, clusters may appear even more tightly-integrated than the FLASH machine. Closer examination reveals three main differences between the two systems:

- hardware DSM networks are faster and more tightly-integrated
- nodes in software DSM systems run separate versions of the operating system
- hardware DSM requires a specialized node controller

The first difference is the speed and integration level of the network. Typically the communication latency in software DSM networks is about an order of magnitude more than in hardware DSM networks ($\sim 10\mu s$ versus under $1\mu s$). In addition, commodity motherboards integrate the network on the I/O bus versus the tighter integration on the node or memory controller in hardware DSMs. These are real differences, but they are rapidly disappearing. The computing industry's new InfiniBand network (discussed further in Section 4.2) has latencies on the order of $1\mu s$ (similar to hardware DSM latencies) and will be connected directly to the memory controller on future commodity motherboards [17].

The second difference is that unlike hardware DSM systems in which every node is under the control of a single operating system, software DSM systems run in an environment where each node executes its own version of the operating system. The critical aspect of this distinction with respect to hardware DSM is the lack of a central page table accessible by all nodes in the system. Instead, each operating system maintains its own set of virtual-to-physical mappings. As we address in Section 4.3, we can make this distinction disappear, if necessary, by making a "distributed page table" that acts like the centralized page table in a hardware DSM-capable operating system.

With a software DSM system constructed on InfiniBand-based clusters, the only architectural difference that remains is the last one listed above: the specialized node controller. In hardware DSM machines, this node controller implements the directory-based coherence protocol at a fine-grain and offloads the overhead from the main microprocessor. If these functions were integrated into the memory controller of a commodity box, the construction of a hardware DSM machine becomes as simple as the construction of a cluster today because there is essentially *no* remaining difference between the two architectures.

While the integration of the specialized hardware DSM functions into commodity servers and workstations is possible, the economic arguments have not been compelling enough to include this functionality for three reasons: The size of the high-performance hardware DSM market has never been large, this additional functionality does not improve uniprocessor performance, and it may not be obvious that adding this functionality will eventually be the only thing standing between achieving hardware DSM performance at software DSM cost. So the debate about whether the necessary controller functionality will ever become commodity would be left at that, except for two factors. First, there is a trend toward placing more CPUs per machine in today's SMP boxes. This is naturally accompanied by a higher cost per box. The desire to keep the cost of individual boxes low, while retaining the ability to program an entire cluster as if it were a single SMP and achieve similar performance, will be a powerful economic argument for including hardware shared-memory support in commodity cluster components. Second, the recent research into the single-machine performance benefits associated with *active memory systems* could further tilt the scales in the debate of whether to include such support in the high-end servers of the future. The next section describes how active memory systems improve the performance of single machines, which lends strong arguments for their inclusion in COTS components.

3.1. Active Memory Systems

One of the biggest challenges facing modern computer architects is overcoming the *memory wall* [37]. Technology trends dictate that the gap between processor and memory performance is widening. Even though good cache behavior mitigates this problem to some extent, memory latency remains a critical performance bottleneck in modern high-performance processors. Heavily pipelined clocked architectures have improved memory bandwidth, but this does nothing to address memory latency or reduce the number of cache misses incurred by the processor.

One approach to reducing the gap between processor and memory performance is to move processing into the memory system by using active memories [5,11,12,30,34,35,37]. Schemes vary, but either parts of a program that have poor cache behavior are executed in the memory system, thereby reducing cache misses and memory bandwidth requirements; or address remapping techniques are used to re-structure data (like linked lists or non-unit-stride accesses) so that the processor can access them in a more cache-efficient manner. Recently, we made the observation that active memory techniques can be treated as an extension of the cache coherence protocol, and proposed two-level active memory systems [30]. The key components of such a system are an active memory controller that implements the coherence protocol and the extensions necessary to support active memory, and active memory elements that contain both memory arrays and processing capability and provide the ability to process large amounts of data in parallel. It is possible to build active memory systems with only an active controller or only active memory elements. In fact, previously proposed active memory systems have either active controllers or active memory elements but not both. However, we believe that the active memory controller is the key part of an active memory system since it allows the transparent use of shadow address techniques. Our particular two-level active memory approach is described in more detail in Section 4.1.

Although active memory research is still in its infancy, initial results from all researchers are

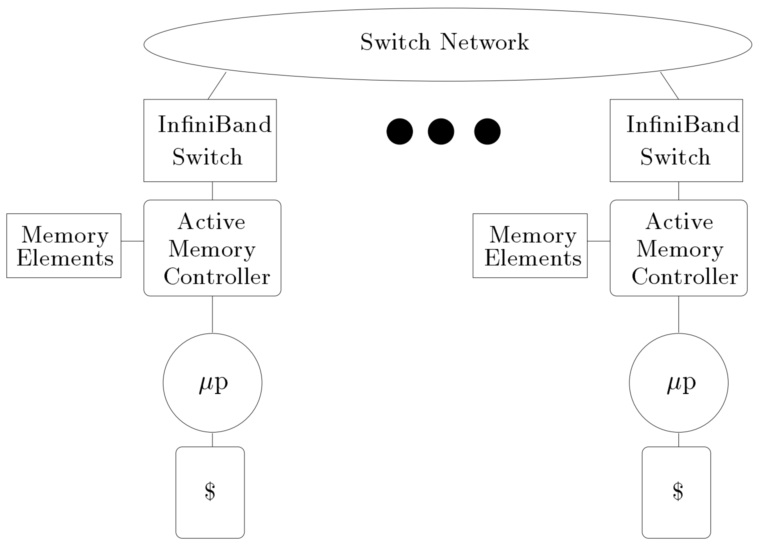

Figure 1: An active memory cluster.

promising. The severity of the memory wall problem dictates that something needs to be done, and given the progress of technology it seems some type of an active memory system is inevitable. It is important to note that this new architectural feature benefits *uniprocessors*, and therefore a strong argument can be made that commodity cluster nodes of the future will have active memory support, at least as an option, much as the high-end cluster components of today come with advanced options not found on average workstations.

3.2. Convergence

We return now to our discussion of hardware versus software DSM systems. In Section 3 we distilled the architectural differences down to a single issue: whether the fine-grained coherence functionality needed for hardware DSM would ever be integrated into a commodity node. If this argument could be convincingly made, the need for page-based software DSM systems would disappear, and clusters could be used as hardware DSM machines with the performance advantages that come with it. Unfortunately the economic argument for adding this functionality has not been strong enough to bring the idea to fruition.

We believe that the active memory controllers being proposed to help resolve the memory wall problem contain the same functionality needed for cluster-based hardware DSM systems to become a reality. In fact, we have recently implemented active memory extensions on the specialized node controller in the FLASH hardware DSM machine [30]. This observation—that active memory controllers and hardware DSM controllers share much of the same functionality—is the central idea in this paper. It strengthens the cases of both the active memory and hardware DSM advocates. Even if the individual arguments for including specialized controller functionality fall short, their combined benefits may be enough to finally produce commodity nodes with active controllers.

In the next sections we will show how our active memory clusters (see Figure 1) can use the active controller and continue to run individual operating systems while still achieving hardware DSM performance. Alternatively, if the operating system has native DSM support, it could run on

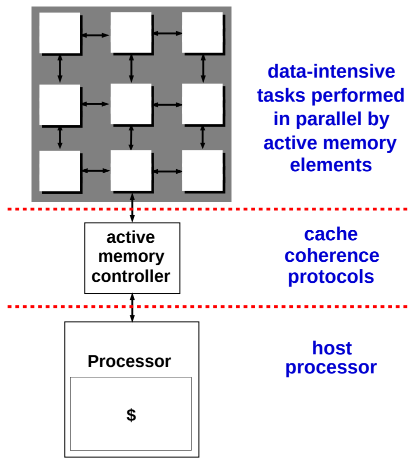
data-intensive tasks performed in parallel by active memory elements
active memory controller
cache coherence protocols
Processor
$
host processor

Figure 2: Two-level Active Memory System.

active memory clusters in the same fashion it would run on a traditional hardware DSM machine. We will see that either operating system model will work.

In summary, we believe that the combination of a single enhanced memory controller and evolutionary improvements in the integration of network technology will result in the architectural convergence and the potential for commodity implementations of active memory, hardware DSM, and software DSM systems. The result will be hardware DSM performance at software DSM cost.

4. Active Memory Clusters Implementation

This section discusses the implementation details of an active memory cluster system. We focus on the three architectural differences between current hardware and software DSM machines that we presented in Section 3. We first describe what controller support is needed for active memory systems and how this support is similar to that needed in hardware DSM machines. We then discuss the ramifications of upcoming tightly-integrated commodity networks. Finally we end with a discussion of the issues that need to be solved to run commodity operating systems on an active memory cluster.

4.1. Controller Functionality

In our approach to active memory systems, we make the distinction between active memory controllers and active memory elements. Our initial work with active memory controllers using the FLASH prototype indicates that the occupancy of an active memory controller would be significantly reduced by the introduction of active memory elements, thereby improving overall system performance [15]. We introduced a *two-level* approach to active memory systems that focuses on designing active memory elements that can assist an active memory controller in performing data-intensive operations in the memory system itself. While the data-intensive calculations are best performed in the active memory element, the cache coherence problem (described below) is best solved in the active memory controller. This novel two-level approach to active memory systems is depicted in Figure 2.

Recent active memory proposals have advocated the technique of remapping the address space of a process in an application-specific manner. Accesses to this space are then used as a signal to the memory controller to perform "active" operations rather than satisfying this access from physical memory [5,13,30]. For example, when performing matrix operations that require row and column traversals, one traversal uses the cache effectively whereas the other does not. We can provide multiple memory viewpoints of the same matrix using shadow address spaces. Row traversals are unchanged, whereas column traversals are treated as row traversals of a matrix at a different (shadow) address. The memory controller issues scatter/gather commands to the active memory elements, which, in turn, fetch individual double-words from a column and return them in a single cache line. Data is therefore provided in blocks that can be cached efficiently by the main processor. The result is good cache behavior for both row and column traversals of the matrix. Such an approach could potentially speed up many scientific applications by using the processing capability in the memory system. Similar remapping techniques can be used to speedup linked-list-intensive programs, and other active memory improvements are the topics of our current research. Uniprocessor speedups as large as a factor of 3 have been reported by researchers building active memory controllers [50].

The key challenge with this active memory approach is solving the cache coherence problem it creates. For example, if columns of a matrix are being written via a different address space during column traversals, the next row traversal via the normal address space will return incorrect or stale data unless care is taken or costly cache flushes are performed. The key insight into solving the coherence problem in active memory systems is that the active memory controller controls both the coherence protocol *and* the fetching of the requested data by the processor. In architectures like the Stanford FLASH multiprocessor [22] and the S3.mp [33] the coherence protocol itself is programmable or extensible. Thus, it is possible to treat active memory support as an extension of the cache coherence protocol. In this case, the active memory controller can enforce coherence between the original and shadow address spaces.

Although active memory research is in its early stages, initial results from many researchers show that the technique can be quite effective for improving uniprocessor performance [24,34,50]. Because active memory techniques can improve processor performance in the face of the memory wall problem, we believe it will become part of the commodity servers of the future. If we assume that a commodity controller has the active memory functionality we describe above, what else does it need to support hardware DSM? Below we list the features of our active memory controller, and then describe the relatively minor additions required for hardware DSM.

Active Memory Controller Features. Our proposed commodity active memory controller manages the cache coherence protocol. Note that even a uniprocessor requires cache-coherent I/O independent of the coherence extensions that we propose for active memory functionality. The consequence of this is that the memory controller already has the ability to invalidate cache lines from the processor and retrieve dirty data from its cache. It also must maintain the ability to track sharing information in the system. In addition, the processor must be able to communicate with the memory controller through uncached writes to a portion of the address space. These writes are interpreted as commands by the memory controller. The final features of the active memory controller are twofold. First the controller must have the ability to dispatch both normal and active coherence handlers by looking at certain bits in the request and address on the processor or I/O bus. Second, in addition to the processor and I/O interface, the controller must have a network interface as shown in Figure 1. As we discuss in Section 4.2, the direct connection of the memory controller to a network such as InfiniBand will be present in commodity nodes regardless of whether or not active memory is supported.

Additional support required for hardware DSM. The key additional features needed for hardware DSM beyond those present in an active memory controller are the ability to dispatch handlers from the network interface (including deadlock avoidance in the scheduling mechanism) and the existence of the corresponding coherence handlers for network messages. It is important to realize that in flexible or extensible active memory controllers, the additional coherence handlers that are needed can be thought of as nothing more than a larger "code size", and do not necessitate changes to the active memory controller architecture.

In summary, if active memory support is included in commodity memory controllers, very little needs to be done to expand that functionality to support hardware DSM—especially if the controller has a mechanism to flexibly extend the cache coherence protocol. The next section describes how a hardware DSM-style network will be available in a commodity system.

4.2. Network Integration

Commodity architectures are witnessing evolutionary changes in network integration as the network connection moves from a plug-in card on the distant I/O bus to a routing chip directly connected to the memory controller (see Figure 1). The InfiniBand network is one example. Although the network is designed for use in storage networks and supports user-level heavyweight protocols, an active memory cluster only needs to use the physical routing capabilities of the network switches. The network switch is directly connected to the memory controller. Since the memory controller in our design is our active memory controller, it can send low-level coherence messages over this commodity switch network. The messages travel between memory controllers only and are not forwarded up to the processor for handling via interrupts. Instead, an active memory cluster can handle the messages entirely in the memory controller, similar to a hardware DSM machine. The only requirement is that the memory controller format its data payloads (coherence messages) in the physical packet format understood by the routers. Early InfiniBand specifications [17] show that this format has $\sim$10 bytes of overhead on top of the data payload of the packet (if you ignore other fields associated with user-level messaging), similar to typical hardware DSM machines.

Early reports on the latency and bandwidth characteristics of InfiniBand as well as its physical routing capabilities are comparable to (and often better than) networks used in today's hardware DSM machines. The "hop time" through a switch is on the order of 10ns, versus 50ns for the routers used in the SGI Origin 2000 and the FLASH multiprocessor. A 10ns hop time in the FLASH machine would reduce the nearest neighbor remote read latency from 860ns to 620ns. The bandwidth of the network is 2.5Gb/s (625MB/s). While this is slightly less bandwidth than the network in an Origin 2000 (800 MB/s), it is 2.5 times higher than that used in the most advanced software DSM machines. Coupled with the massive reduction in latency, this network will provide an excellent base on which to build a high-performance DSM machine.

4.3. Operating System Issues

At this point, we plan on using Linux to implement AMC because the source code of the virtual memory manager is readily available and free of licensing issues should we choose to modify it. The only functionality that AMC requires from the host OS is the ability to request that a specific virtual address map to a physical address within a certain range of the physical memory address space, a feature commonly used to map I/O devices into a process' virtual space. Therefore, we believe that our solutions are general enough to be implemented on any modern operating system, and we are currently writing the necessary device drivers to port AMC to Windows 2000.

As mentioned in Section 3, software DSM runtime layers are typically used on clusters of workstations, each running their own version of the host operating system. The majority of these clusters consist of the same types of machines running the same version of the same operating system. The reasons for this include the cost advantage of buying in bulk, ease of cluster-wide system management, and the avoidance of the overhead associated with translating between different data representations (as is done in the Amoeba and Emerald systems [19,47]). Clusters providing high-performance parallel programming environments are not groups of machines sitting on desks in an office environment, but rather sets of rack-mounted machines sequestered in machine rooms. Thus, while companies such as *Entropia* seek to use heterogenous, widely-spread machines for large-scale independent computations, realistically we do not expect high-performance cluster machines to be constructed of dis-similar components.

Although the use of an operating system with native DSM support and NUMA-aware policies such as Irix 6.5 would result in an AMC achieving hardware DSM performance at software DSM cost, for completeness we still address the case of a cluster comprised of a group of nodes where each runs its own operating system. The fact that the operating system on each node operates without awareness of other nodes presents several problems for active memory clusters:

- virtual–to–physical page mappings will most likely not be the same on all machines
- pages swapped out by the OS may cause problems when swapped back in
- pointers to shared data must be distributed to all participating processes

These three issues arise during the initialization portion of an AMC application. Therefore, in operating systems without DSM support, AMC application startup closely follows that used in software DSM systems. Here we present a complete description of the AMC initialization procedure, which will address the three points above.

When the application starts, it calls an initialization routine provided by the AMC library. This initialization routine has four main stages:

Remote process startup. A shared-memory application is started on a single node of the AMC, and this process specifies other nodes that will participate in the computation. The initialization library uses a standard remote execution mechanism to initiate a copy of the AMC process on all other nodes, with the only difference being the node ID. As in software DSM, this presupposes the existence of a shared file system and the permission of the initiator to access it on each participating node.

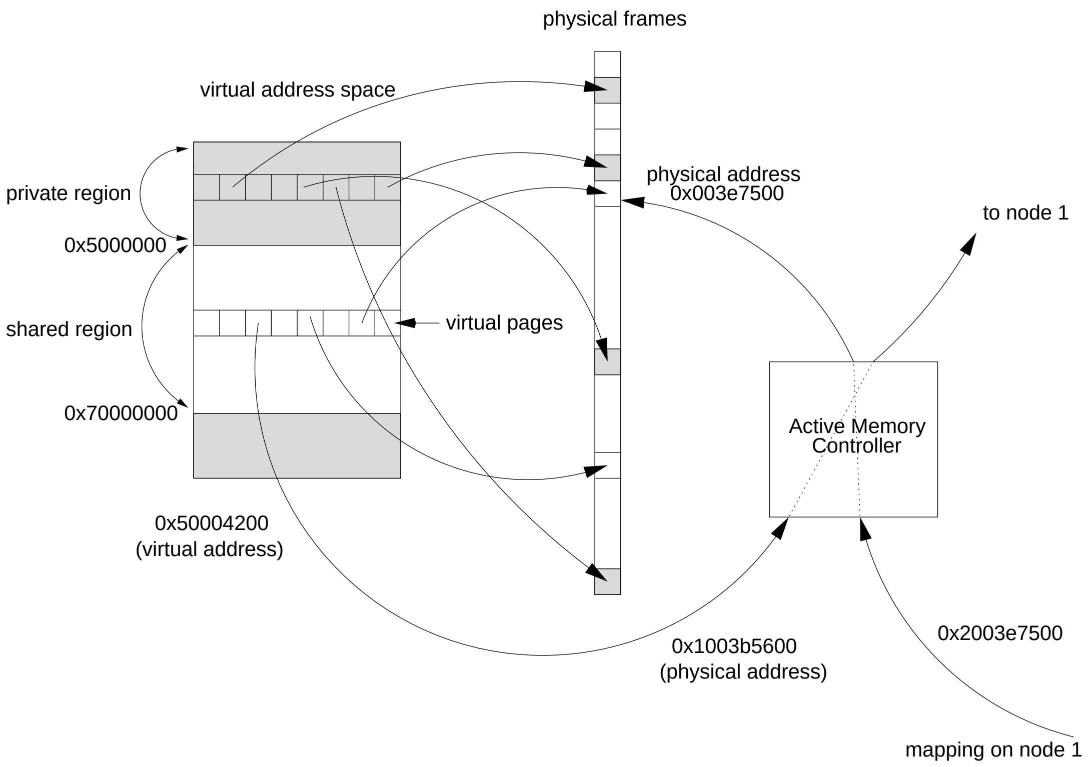

Figure 3: Virtual to Physical Mapping for Example Node 2.

Allocating correct virtual–to–physical mappings. The lack of a centralized page table accessible by any AMC process requires AMC to ensure that the same virtual–to–physical mappings exist on each node. The easy part of this process is ensuring that the shared-memory region exists in the same virtual address range on all nodes. This is commonly done in software DSM systems and poses no problem to AMC. However, a more difficult problem is making sure that all virtual–to–physical mappings are the same on each node. This is essential because hardware DSM memory controllers initiate coherence actions based on the physical address presented to them. If these mappings do not match from node to node, a translation table must be maintained by each memory controller for every physical frame for every remote node in the system. For space and performance reasons, this is clearly not an attractive alternative.

AMC addresses this problem through the use of a specialized device driver that maps virtual pages to the correct set of physical frames. Figure 3 depicts the memory translations present in a single node of the AMC cluster (the translation performed by the page table is omitted for clarity). The virtual address space contains a region agreed upon by all nodes to represent "shared memory" (the virtual address region between 0x50000000 and 0x70000000 in Figure 3), and this region resides at the same place in the virtual address space of each AMC process. Non-shared data regions are allocated by normal allocation methods (e.g., `malloc`), while shared-memory regions are allocated through a special allocation routine provided by the AMC library.

We assume at least a 40-bit physical address space, which for machine sizes up to 256 nodes provides us with 32-bits of physical memory and 8-bits of *node identifier*. The node identifier is used to indicate the home node for a particular page to the AMC memory controller. During initialization, each node allocates pages for which it is the home node (dependent on some application-specific page placement policy) through the use of the specialized memory mapping device driver. For local allocations, the upper 8 bits are set to 0 to allow the memory controller to respond to local requests as normal. The lower 32 bits specify the actual frame residing in physical memory on the home node.

As shown in Figure 3, the physical frames allocated at each home node do not have to be contiguous, their placement being decided by the the unmodified virtual memory manager running on the host OS. Shared frames and private frames may be intermixed, as indicated by the shaded and unshaded regions of the physical address space. When each node has completed this mapping process, messages are sent between every pair of nodes to inform all nodes of the correct mappings for their portion of the shared address space.

Distributed page table dissemination. After a node has created all local virtual–to–physical mappings, this information must be transmitted to other nodes running the AMC application. We use standard user-level messaging to accomplish this. When Node Y receives the mappings from Node X, Node Y will utilize the AMC Mapping Driver to establish mappings between the correct virtual page addresses and physical frames. Note that the physical frames do *not* have to be allocated on the remote node. The only two requirements are the following:

1). The virtual–to–physical mapping must indicate to the active memory controller that this page resides on a remote node. This is accomplished by using the upper 8 bits of the physical address as a node identifier, as shown in Figure 3 by the translation of virtual address 0x50004200 to physical address 0x1003b5600. The presence of a non-zero node identifier will cause the active memory controller to request the page from the indicated remote node instead of attempting to fill the request from local RAM.

2). The lower 32 bits of the physical address must be the same as on the home node. This ensures that requests from remote nodes will access the correct physical frame. The arrow in Figure 3 labeled “mapping on node 1” displays the situation in which a remote mapping points to a physical frame on this local node. Note that the node identifier is simply stripped off before the physical memory is accessed.

Because physical memory is only allocated on the home nodes, a substantial memory savings is realized over software DSM systems that must allocate the entire shared address space on every node. Thus, we will have mappings in each local page table that actually refer to physical frames on other machines. The memory controller will ensure that the local memory is not accessed on local cache misses to “remotely allocated” pages. Pages for which the local node is the home node will be allocated from the available physical RAM on the local system, which will respond to accesses from the local processor(s) normally. Our “distributed page table” therefore consists of *real* mappings for pages for which the local node is the home, and *shadow* mappings for pages whose home node is located across the network.

Pinning shared pages. We can easily provide initial virtual–to–physical mappings through the use of a device driver as described above without modification to the kernel. However, two problems arise if we allow the operating system to swap out pages currently in use by AMC. First, the AMC memory controller may receive a request for the swapped-out page. Since the memory controller does not know the page has been swapped out by the operating system, the AMC controller will respond with the current, incorrect contents of the physical frame referenced. Second, the operating system may choose to swap the page back into physical memory at a different location than the original one, causing the memory controller to again fetch incorrect data in response to remote requests.

For these reasons, and for performance reasons, we plan to pin shared pages in memory at the home to prevent the OS from swapping out the page during the course of the program's execution. Because only those shared pages for which the local node is the home node must be pinned, we do not see this as an unreasonable requirement. However, a few minor OS modifications could eliminate the necessity of pinning pages in AMC. For example, if we modified the virtual memory manager to inform the AMC memory controller when a page is swapped out, we could initiate the necessary TLB shoot-down to resolve the first problem indicated above. To address the second problem, a re-map of a shared page by the operating system could invoke our mapping device driver, and the new mapping would then be sent to all AMC memory controllers as is done during initialization. Implementing these options would remove the pinning requirement, but would involve kernel modifications that may or may not be possible depending on the operating system in use on the cluster.

Distributing the contents of shared pointers. Similar to software DSM systems, only data allocated with the AMC library `shmalloc` routine will be shared across the cluster. Variables that are declared globally (outside of any procedure body) will be global only to threads residing in the local process. Pointers to shared memory reside in this *process-specific global space*, and are located at the same address across AMC processes because we are executing a copy of the same executable on each node. However, when a process allocates shared memory and assigns it to such a pointer, the new contents of the pointer are only visible from the calling process. Therefore, we must *distribute* the value contained in the pointer to the other AMC processes on the cluster to ensure that the pointer holds the same value in all address spaces. This is accomplished via standard user-level messaging as is commonly done in software DSM systems, in contrast to a typical hardware DSM system in which remote processes would be forked after global memory has been allocated, ensuring that global pointers "pointed" to the same region of shared memory.

5. Expected Performance Results

The following are a set of interesting performance comparisons involving active memory clusters:

AMC performance vs. hardware DSM. After AMC has completed the initialization described in Section 4.3, execution proceeds without any additional overhead beyond that normally experienced by hardware DSM systems with flexible memory controllers. For this reason, we expect the parallel performance of AMC to be identical to a hardware DSM system built from custom hardware with the same network characteristics as AMC. Thus the simulation results showing the parallel execution time of applications running on both systems would be the same. This is not surprising, since this is the insight of the paper. Active memory support plus the DSM-style InfiniBand network *is* a hardware DSM machine modulo the system software issues.

AMC performance vs. traditional software DSM on an InfiniBand cluster. We expect the performance of traditional software DSM on InfiniBand to perform better than current software DSM systems because of three features of InfiniBand and other emerging network architectures:

- a significant reduction in memory latency
- the capability of remote DMA operations
- reliable communication mechanisms

Nevertheless, for most applications we would still expect AMC to outperform software DSM due to software DSM's remaining high kernel overhead involved with page exception handling, the large granularity of sharing required, and the processing overhead for runtime structures such as `diffs` and `twins`. These are the same well-studied reasons that hardware DSM machines generally outperform software DSM machines. Regardless of the performance difference, however, an AMC-enabled cluster could run *either* hardware or software DSM, so the question of which performs better on which applications is inconsequential.

Most importantly, however, and central to the theme of this paper, the parallel performance of AMC does little toward strengthening the argument for putting the required functionality in commodity boxes. The only way the active memory controllers necessary to make AMC a reality will be placed in commodity servers is by the industry embracing the research results showing active memory techniques will significantly improve the performance of a standalone machine, as discussed in Section 4.1. When this happens, designing a *flexible* active memory controller will enable us to realize hardware DSM performance at software DSM cost.

6. Conclusions

In this paper, we have shown that building clustered–compute farms that deliver hardware DSM performance at software DSM cost is becoming a real possibility. We show that the only necessary changes are utilizing a more tightly-integrated network technology, adding new functionality in the memory controller, and writing a small amount of supporting software.

Of these three enabling mechanisms, the most problematic is the additional functionality required of the memory controller. Until recently, the additional cost of changing the memory controller to support hardware DSM was not justified by the expected performance gain for two main reasons. First, the required support did nothing to help uniprocessor performance, which is the metric by which the computer industry measures any technology for inclusion in "commodity" boxes. Second, the number of users that would benefit from such a costly change to the memory controller architecture was relatively small. Meanwhile, recent research in active memory systems has argued for enhanced memory controller functionality to improve *uniprocessor* performance. By observing that the active memory support can be treated as an extension of the cache coherence protocol, we realized that the controller support needed for active memory and hardware DSM is almost identical, provided the active memory mechanisms are implemented in a flexible manner.

Industry has already begun addressing the issue of network–system integration to support the low communication latency required by the cluster–based systems in use today. We used InfiniBand as an example of such a network architecture, but others exist (e.g., RapidIO [32]) that would work equally well with active memory clusters.

The only remaining issue is one of system software. Software DSM machines have had a distinct advantage in this area, because the use of commodity operating systems is an important factor in keeping both initial system cost and subsequent upgrade costs low. The active memory cluster architecture can certainly be used with an operating system that natively supports hardware DSM. However, we have shown that simple functionality provided by device drivers can be used with commodity operating systems to achieve hardware DSM performance, even in the absence of a specialized DSM operating system.

In summary, current research in the area of active memory systems shows that active memory techniques can improve uniprocessor performance in cases where caching behavior is poor. The desire for including active memory support in commodity machines now creates two communities clamoring for the same controller functionality. As active memory research matures, this may be enough to warrant including this functionality in commodity memory controllers. When that happens, our claim of hardware DSM performance at software DSM cost will become a reality, effectively merging the two fields and combining the advantages of both.

References

[1] G. Abandah and E. Davidson. Effects of Architectural and Technological Advances on the HP/Convex Exemplar's Memory and Communication Performance. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 318–329, June 1998.

[2] A. Agarwal, R. Bianchini, D. Chaiken, et al. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, June 1995.

[3] A. Bilas, C. Liao, and J. P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[4] J. B. Carter et al. Design of the Munin Distributed Shared Memory System. *Journal of Parallel and Distributed Computing*, **29**(2):219–227, September 1995.

[5] J. B. Carter, W. C. Hsieh, L. B. Stroller, et al. Impulse: Building a Smarter Memory Controller. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture* January 1999.

[6] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[7] Data General Corporation. Aviion AV 20000 Server Technical Overview, *Data General White Paper*, 1997.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine, 1994.

[9] A. Erlichson, N. Nuckolls, G. Chesson and J. Hennessy. SoftFLASH: Analzying the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, October 1996.

[10] M. Galles. Spider: A High-Speed Network Interconnect. *IEEE Micro*, **17**(1):34–39, January-February 1997.

[11] M. Gokhale, B. Holmes, and K. Iobst. Processing in Memory: the Terasys Massively Parallel PIM Array. *Computer*, **28**(3):23–31, April 1995.

[12] M. Hall et al. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. *Supercomputing*, Portland, OR, Nov. 1999.

[13] J. Heinlein. *Optimized Multiprocessor Communication and Synchronization Using a Programmable Protocol Engine*. Ph.D. Dissertation, Stanford University, Stanford, CA, March 1998.

[14] M. Heinrich. The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols. Ph.D. Dissertation, Stanford University, October 1998.

[15] C. Holt, M. Heinrich, J. P. Singh, et al. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[16] Y.C. Hu, H. Lu, A.L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. In *Proceedings of the 13th International Parallel Processing Symposium*, April 1999.

[17] InfiniBand Architecture Specification, Volume 1.0, Release 1.0. InfiniBand Trade Association, October 24, 2000.

[18] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. *Operating Systems Review*, **29**(5):213–228, December 1995.

[19] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, **6** (1):109–133, February 1988.

[20] P. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 1996.

[21] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–132, January 1994.

[22] J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[23] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[24] J. Lee, Y. Solihin, and J. Torrellas. Adaptatively Mapping Code in an Intelligent Memory Architecture, In *Proceedings of the Second Workshop on Intelligent Memory Systems*, November 2000.

[25] D. Lenoski. The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor. Ph.D. Dissertation, Stanford University, December 1991.

[26] D. Lenoski, J. Laudon, K. Gharachorloo, et al. The Stanford DASH Multiprocessor. *IEEE Computer*, **25**(3):63–79, March 1992.

[27] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Coherence Protocol for the Stanford DASH Multiprocessor. In *Proceedings of the 17th International Symposium*

on Computer Architecture, pages 148–159, May 1990.

[28] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *ACM Transactions on Computer Systems*,7(4):321–359, November 1989.

[29] T. D. Lovett, R. M. Clapp, and R. J. Safranek. NUMA-Q: An SCI-based Enterprise Server. Sequent Computer Systems Inc., 1996.

[30] R. Manohar and M. Heinrich. A Case for Asynchronous Active Memories. In *ISCA 2000 Solving the Memory Wall Problem Workshop*, June 2000.

[31] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 1.0, 1994.

[32] Motorola Semiconductor Product Sector. RapidIO: An Embedded System Component Network Architecture. February 22, 2000.

[33] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[34] M. Oskin, F.T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[35] D.A. Patterson, T. Anderson, et al. A Case for Intelligent RAM: IRAM. *IEEE Micro*, **17**(2), April 1997.

[36] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325–336, April 1994.

[37] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 90–101, May 1996.

[38] Scalable Coherent Interface. ANSI/IEEE Standard 1596-1992, August 1993.

[39] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low-Overhead Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

[40] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lukas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood. Implementing Fine-Grain Distributed Shared Memory on Commodity SMP Workstations. Technical Report 1307, University of Wisconsin Computer Sciences, March 1996.

[41] R. Simoni. Cache Coherence Directories for Scalable Multiprocessors. Ph.D. Dissertation, Stanford University, Stanford, CA, October 1992.

[42] R. Soundararajan, M. Heinrich, B. Verghese, et al. "Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors". In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 342–355, June 1998.

[43] E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the First Usenix Windows NT Symposium*, August 1997.

[44] E. Speight, H. Abdel-Shafi, and J. K. Bennett. An Integrated Shared-Memory/Message Passing API for Cluster-Based Multicomputing. In *Proceedings of the Second International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 146–153, December 1998.

[45] P. Stenstrom, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 109–118, May 1993.

[46] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[47] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM***33**(12):46–63, June 1990.

[48] S. C. Woo, M. Ohara, E. Torrie, et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[49] D. Yeung, J. Kubiatowicz, and A. Agarwal. MGS: A Multigrain Shared Memory System. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 44–55, May 1996.

[50] L. Zhang, V. K. Pingali, B. Chandramouli, and J. B. Carter. Memory System Support for Dynamic Cacheline Assembly. In *Proceedings of the Second Workshop on Intelligent Memory Systems*, November 2000.