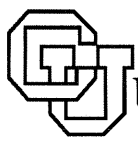
Hemingway,
A Distributed Shared Memory System

Anshu Aggarwal
Dirk Grunwald
Trent R. Hein
Evi Nemeth

CU-CS-813-96



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Hemingway,
A Distributed Shared Memory System

Anshu Aggarwal, Dirk Grunwald, Trent R. Hein, and Evi Nemeth
Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
(Email: {anshu,grunwald,trent,evi}@cs.colorado.edu)

March 23, 1995

Abstract

Distributed shared memory systems can be divided into single-writer and multi-writer protocols. Multi-writer systems can further be divided into loosely-consistent and write-through protocols. Single writer systems can suffer from excessive false sharing. Both single-writer and loosely-consistent multi-writer protocols depend on synchronization communication susceptible to high latency. By comparison, a write-through, weakly-consistent distributed shared memory system relies largely on asynchronous write operations. These writes can be pipelined, consolidated or squashed using write buffers. The remaining latency-sensitive operations include copying readable pages.

If we assume that communication bandwidth will become increasingly plentiful, but that communication latency will not increase as fast as bandwidth, write-through protocols have several advantages. In this paper, we describe those advantages and the design of Hemingway, a write-through, weakly consistent distributed shared memory system. The Hemingway distributed shared memory system writes in short, bursty messages since these are asynchronous and inexpensive in our network model. Memory is read in pages, and the underlying virtual memory system is used to control access and sharing. We also describe the system and networks being used to implement Hemingway and present architectural simulations showing this design provides better performance than single-writer or loosely-consistent DSM systems.

1 Introduction

Specialized high-performance shared memory computers, such as the KSR-1 are difficult to justify economically. Several researchers have investigated using workstations and commodity networks to provide a parallel computing environment similar to such special-purpose multicomputers. Distributed shared memory can provide a shared-memory programming environment for these systems.

Concurrently, a new class of network architecture has arisen that provides considerable bandwidth and low communication overheads. These systems, typified by the Princeton Shrimp system [4], the VAX-cluster [16] and Memory Channel [12] developed by Digital Equipment Corporation and the Hamlyn [28]

system proposed by Hewlett-Packard use sender-initiated placement of data to reduce the operating system overhead of messaging. Most of these interfaces use memory-mapped virtual memory pages to specify the communication mapping. They are an extension of communication hardware that is common in real-time systems. Other memory interconnects support direct communication without consistency, including the Illinois Cedar [11, 17], Cray T3D [9] and Elxsi multiprocessors.

In this paper, we investigate the performance of a distributed shared memory system that exploits appropriate commodity components at each level of the communication hierarchy. We use the hardware in ways that can continue to scale with advancing technology, and take advantage of device and system characteristics to provide a scalable shared memory environment for a moderate number (10's-100's) of processors.

We connect small shared memory multiprocessors using a write-though interconnection network that avoids operating system intervention. Processors within each multiprocessor use the native broadcast mechanism to implement processor consistency. Communication between multiprocessors is divided into write traffic, which is sent in small bursts using the memory mapped communication hardware and read traffic, which copies large virtual memory pages. The communication model between processors uses lazy release consistency, although any weak consistency protocol can be used.

A weakly-consistent memory model implies that a programmer must lock any data structure that will be written to and must not care if an unlocked object being read is subsequently changed by another thread of the program. A number of studies have shown that this is not a burden on programmers.

The result is a *write-though, weakly-consistent* distributed shared memory system that avoids high-latency communication wherever possible, exploits increasing inter-system bandwidth and uses proven technology, such as broadcast-based multiprocessors, to achieve high performance. Our design uses the underlying operating system and commodity communication hardware that will be widely available to construct distributed systems.

In the remainder of this paper, we describe how we combine the multiprocessor and the communication network to provide an efficient, scalable distributed shared memory architecture using operating system support. By necessity, we use simulation to model large systems to determine the scalability of our design. However, we include measurements from components of the prototype systems to quantify the performance we can expect to achieve. We expect to have a prototype implementation running during the summer using a cluster of AlphaServer 2100-4/275 multiprocessors and early versions of the Memory Channel network. In the next section, we describe the design of individual components in the network, the protocol used to maintain consistency and alternative mechanisms for maintaining write consistency. In §2.4 we describe our experimental methodology, the specific characteristics of the systems simulated, the parametric variations we investigated and the performance metrics used to compare performance. The results of our performance study appear in §3.

2 The Design in a Nutshell

We discuss building a multicomputer *system* using two or more multiprocessor *boxes* containing one or more *processors*. We assume communication within a box is managed by a broadcast protocol and boxes are connected with point-to-point links.

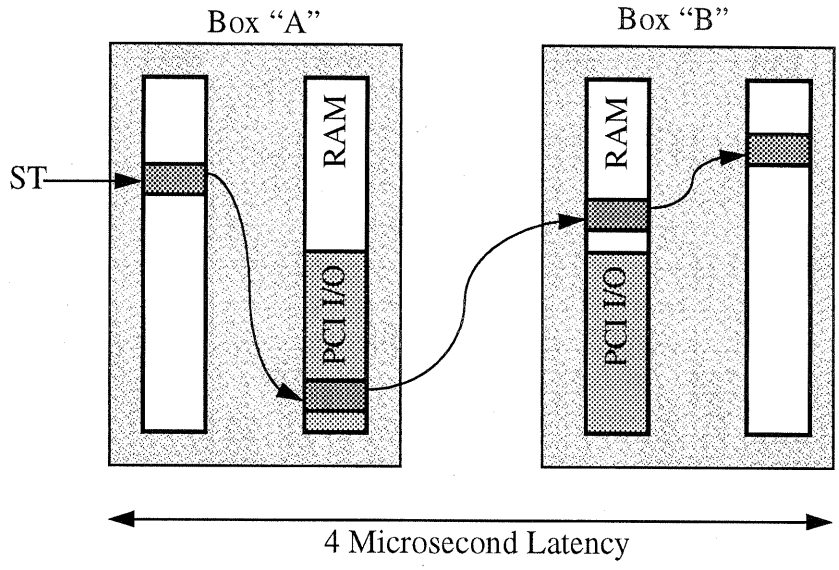


Figure 1: Communication model with Memory Channel hardware

2.1 Hardware Subsystems

Each box is assumed to be a commercial multiprocessor with a few (1-8) processors. We model the AlphaServer-2100 4/275 multiprocessor as closely as possible, since these will be used to implement the Hemingway system. However, we have varied the parameters in our simulation system to study the effects of using a variety of system configurations.

Using a hierarchy of memory coherence protocols can result in considerable performance improvement, since common shared-bus architectures use backplanes supporting 500 Mbytes/s to 2GByte/s, and any inter-processor communication within a box updates all other caches using a snooping protocol. Not only is this a much higher bandwidth than that supported by most existing networks, it provides low-latency communication within a box that can be exploited by runtime systems. Furthermore, most modern architectures presuppose designers will build small multiprocessors; for example, both the MIPS R10000 and the Intel P7 architecture directly support bus coherence protocols for inexpensive systems containing 1-4 processors. Clearly, such systems will be plentiful, and provide an inexpensive source of commodity multiprocessors.

Lastly, multiprocessors usually support cache consistency using a snooping protocol. Since our communication model assumes write-through communication that does not interrupt the remote processor, we must force write communication to update the remote processor caches. Uniprocessor systems, including many workstations, often use the simpler, less expensive approach of invalidating all cache entries that may be affected by I/O. In our situation, this would require that all remote memory references interrupt the receiving processor.

2.2 Communication Infrastructure

Our communication network uses the Memory Channel architecture being developed by Digital Equipment Corporation. This design is similar to the the SHRIMP system developed at Princeton [4].

Consider communication between the two systems shown schematically in Figure 1. In this scenario, a process on box "A" intends to communicate a single word with a process on box "B". Communication is established by the operating system, but the actual communication does not involve O/S overhead. The operating system on "A" maps a page of virtual memory in the process address space to the address space managed by the Memory Channel device on the local PCI I/O address space. The operating system on box "B" has already mapped a page of physical memory to a specific virtual address; it must now "pin" that page to insure the physical page is not relocated. The Memory Channel interface on each box establishes a mapping between the PCI I/O page on "A" to the physical memory on "B". Actual communication from "A" to "B" is performed by store instructions native to the processor.

The first generation Memory Channel has a peak 100 Mbyte/s bandwidth and a 4 microsecond communication latency. Future networks will have higher bandwidth, but latency will remain in the range of 1-3 microseconds. The store instructions are asynchronous and can be buffered; processor caches on box "B" are updated, but communication does not block the sending processor unless there is contention for either the shared bus or the Memory Channel and all buffers are full.

2.3 Implementing Distributed Shared Memory

We can use store operations to send data between processors, but no consistency is provided by the Memory Channel interface. In fact, the interface has minimal support for reading information from remote locations. Most direct-memory interfaces provide some support for reading remote memory, but none provide a consistency protocol.

There are a number of ways to handle processor reads. The simplest mechanism would involve an explicit read of the data from the remote processor; this is implemented in a number of systems, including the Cray T3D and Avalon A12 [3]. While simple to implement, the performance would be poor unless prefetching [6, 25] and lockup-free caches [7] were used.

Alternatively, the value that was read could be cached; this reduces the communication latency but requires a more sophisticated consistency protocol. The MIT Alewife architecture uses a distributed cache protocol to implement a single-writer, weakly-consistent distributed shared memory, where the latency for fetching data is masked by a multi-threaded architecture [1]. Kontothanassis and Scott [15] simulated a write-through, weakly-consistent distributed shared memory system that fetched individual cache lines, and compared the performance to a single-writer protocol modeled after the Stanford DASH [18] system. In general, they found that a multi-writer protocol resulted in a shorter execution time and required less coherence traffic. In fact, their simulated software coherence mechanism often resulted in shorter execution times than the hardware mechanisms they simulated.

Although our designs were derived independently, much of the performance of the Hemingway system can be extrapolated from the work of Kontothanassis and Scott [15]. However, we feel we have made some design decisions that will result in a system that is easier to implement and takes better advantage of higher-bandwidth communication. The design proposed by Kontothanassis and Scott fetches individual cache lines. They propose that memory access is arbitrated using ECC parity control to emulate sub-blocks within a virtual page, much as was done on the Wisconsin Wind Tunnel [20, 21]. Access to a page is controlled using the virtual memory system TLB entries. This is used to pin physical memory pages on remote processors and establish a communication mapping using hardware similar to the Memory Channel. Successive references to individual cache lines signal traps that initiate remote memory reads. All writes to shared references would write-through or write-back to the "home" page for a given shared location.

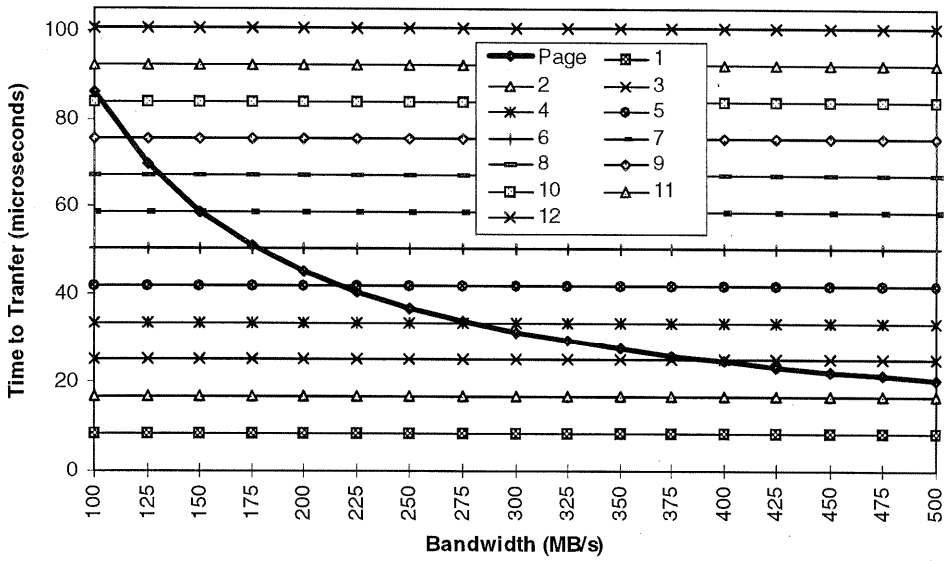


Figure 2: Time to request and receive either a 8KByte page or several smaller 32 byte messages as bandwidth increase and latency remains constant. Transferring a single cache line requires a single 32-byte message. The figure illustrates that as bandwidth increases, the cost of transferring a single page becomes relatively less expensive, because page transfers are dominated by bandwidth while smaller messages are dominated by latency.

Kontothanassis and Scott examined the effect of centralized and distributed management of meta-data concerning shared references.

While this design is elegant, we felt a simpler design is possible and easier to implement. For example, we are unable to change the ECC protection information in our computing infrastructure. Instead, we propose to use existing virtual memory support to read entire pages. In our initial implementation, stores will be duplicated, updating a local copy of shared pages while updating remote pages using write-through communication.

Figure 2 plots communication time, in microseconds, as bandwidth increases. Each line corresponds to the approximate time to fetch either a full 8Kbyte memory page or multiple 32-byte cache lines. Each message is assumed to take $(T_l + \max(T_l, T_b \times B)) \times N$ microseconds, where T_l is the communication latency (4 microseconds), T_b is the time to transmit a single byte (10 nanoseconds), B is the message size in bytes and N is the number of messages. As expected, as communication bandwidth increases, larger messages become relatively cheaper. This communication model is simplistic; it ignores the effects of link contention, and assumes there is no overhead, other than latency, in fetching a remote memory item. This may bias the results in favor of larger messages.

Nevertheless, the model illustrates that increasing bandwidth justifies fetching an entire virtual memory page. Although the current Memory Channel interface has a 100 Mbyte/second bandwidth, it is reasonable to expect the bandwidth of such networks to double or quadruple in the next 3-5 years. At the same time, the history of network design indicates it is unlikely that communication latency will decrease significantly. With current networks, only 10 of the 256 cache lines in a page need to be accessed for page access to be as efficient as single line access. Moreover, since we fetch the page into a cache-coherent multiprocessor, the page benefits multiple processors. More importantly, the sharing introduced by multiprocessors reduces

the demand on the owner of that page to provide too many copies. Faster networks imply that accessing 4-5 individual cache lines will be slower than accessing a single page. The assumption that bandwidth is cheaper than latency is particularly true with extant hardware; in the current Memory Channel, remote pages can only be read by interrupting or otherwise involving the remote processor. Future generations of the network may support an explicit read operation, reducing this overhead.

In our initial design, each page has a "home" machine, and we assume all physical memory has been 'pinned' to simplify the communication protocol. We simply discard copies of shared pages at memory synchronization points. Again, we feel that with increasing bandwidth, it would be faster to fetch a given page again than to engage in a protracted protocol to see if the current copy of the page was inconsistent. We employ the same mechanism for page replacement – rather than swap shared copies to disk, we discard those copies, preferring to fetch data from other processors rather than disks. Even at 100 Mbyte/s, transferring an 8KByte page takes at least 85 microseconds. A 275Mhz dual-issue processor may be idle for over 50,000 instructions while a page is fetched. Our intent is to mask portions of this inter-processor communication using inexpensive threads, although our current simulation environment does not model this.

Remote pages are accessed when the first read request to that page is executed. Following this, the remote page is copied to a local page of physical memory, and all further data is read from the local copy of the shared page. This means that the local processor must update both the local copy of the shared page and the original page on the remote processor. We have considered two ways to update both pages. The first method requires a modified PCI I/O bus interface or "bridge". Normally the bridge transfers requests only for a particular portion of the physical address space. Ideally, the I/O bridge would also be able to transfer write operations destined for specific pages of physical memory. Not only would the range of addresses to be transferred need to be specified, but also the I/O bridge must work with the cache consistency protocol on the system memory bus. For example, the Dragon protocol detects that processors are no longer sharing a particular cache line, and stops broadcasting changes to that cache line. The I/O bridge must indicate that cache lines in a duplicated page are always shared, if a write-through protocol is used to update remote processors. Otherwise, any updates to that cache line would not be propagated to the remote or "home" machine.

We felt it was unlikely that the PCI I/O bridge would be changed in the very near future, particularly on the multiprocessors we are using to build Hemingway. In part, this reflects a compromise we accepted when we decided to use a commodity, high-performance communication interface to construct our DSM system. Although duplicating stores in hardware has a number of uses, including fault-tolerance and disaster recovery, we are using an alternate method for our initial implementation.

Our alternative design is to duplicate the store operations in the program. This is illustrated in Figure 3. We subdivide the virtual memory into two regions. In the figure, the read/write copies of shared pages are shown at the top, and the write-only pages used to update the home copy of shared pages are shown at the bottom. The read/write and write-only pages are separated by a constant distance in the virtual address space. Each store in the program is duplicated. First the store is performed at the original address, and then again at that same address but with the high order byte cleared. For example on the DEC Alpha, the instruction

```
ST Ra, Offset(Rb)
```

becomes

```
LDA  Rc, Offset(Rb)
ST   Ra, 0(Rc)
```

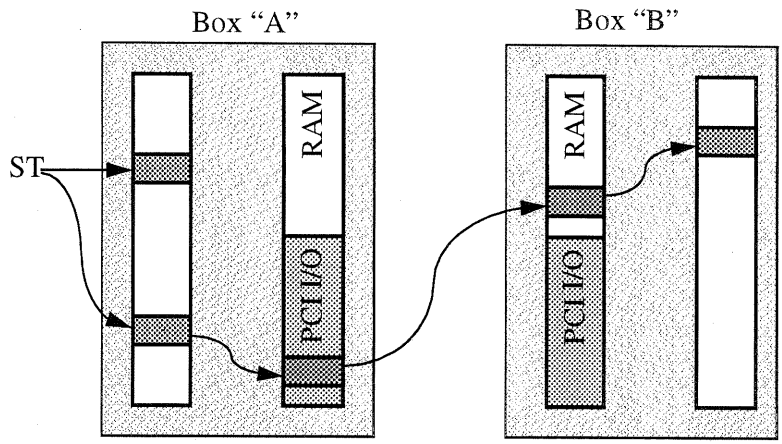


Figure 3: Communication model with duplicate stores

```
ZAP   Rc, 0x40, Rc
ST    Ra, 0(Rc)
```

The ZAP instruction clears a specific byte in a word; in this case dividing our memory space into two sections of 2^{56} bytes. Only stores to shared memory references need to be modified. In general, it is difficult to determine which references will be to shared segments, and we modify all references that are not relative to the stack pointer. We are using OM [24] to duplicate the stores. The OM tool allows whole-program transformation, and reschedules registers and instructions following code modification. We avoid duplicating store instructions in some sections of a parallel program; in particular, we do not modify the signal handler and code used to establish the virtual memory mapping.

Our technique for duplicating stores is similar to the fine-grain access control proposed by Wood *et al* for the Typhoon Blizzard-S memory system [21]. However, the Typhoon system uses fine-grain control to mediate access for loads and stores. In that system, each load and store in the system is expanded to a five or nine instruction sequence. Each access check loads a value from memory to determine if a specific cache line is accessible. However, it is more difficult to schedule the latency incurred by the load in such an access check, and the number of loads and stores in a program can be quite large. Figure 4 shows the fraction of loads and stores encountered during the execution of three common benchmark suites. The measurements were collected on a DEC Alpha using ATOM [23]. Note that the number of loads and stores is about 30% of the instruction stream, but only 7% are stores. Furthermore, simply duplicating the stores introduces no additional loads and fewer pipeline hazards than duplicating both loads and stores. A study comparing the software access control to the hardware mechanism using the ECC bits [21] found that the software method was often twice as slow as the hardware implementation; in part, this may be due to the large number of load operations that need to be modified.

In practice, we duplicate store operations that are not relative to the stack pointer, since we assume the stack is not shared. This means that many non-shared stores are duplicated, but we can not distinguish between shared and non-shared stores. Such analysis would require extensive inter-procedural analysis, and we are unable to perform this analysis at this time. Instead, every write-only page that matches a non-shared page is mapped to a single "junk page", or a section of memory that is never read. Thus, a single mechanism, store duplication, can be used for shared and non-shared references. Load operations always access the read/write page of memory, and references to such pages can be cached at any level of the memory hierarchy and shared between processors in a box.

Duplicating the store operations represents a number of compromises that affect system performance. The store instructions reference different memory pages, and the number of page faults can be expected to

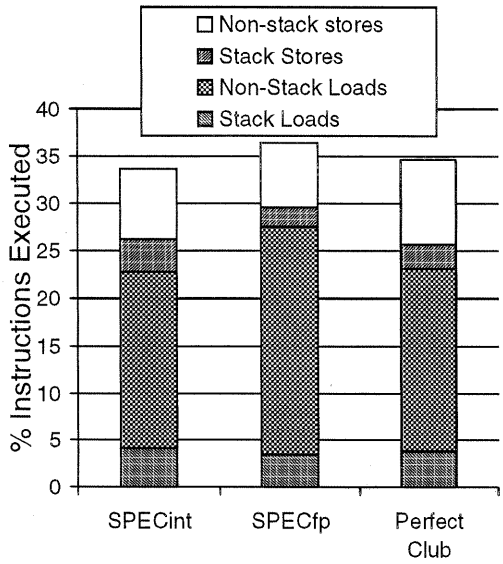


Figure 4: The fraction of load and stores encountered during the execution of common benchmark programs on a DEC Alpha.

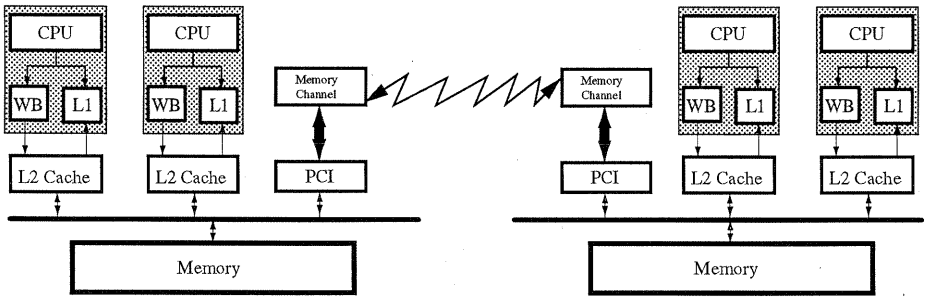


Figure 5: System components in experimental architecture

increase. Likewise, processors such as the Alpha have a limited number of write-buffers used to coalesce and collapse writes. The effectiveness of each of these capacity-limited resources is affected by the decision to duplicate stores. Again, we view store duplication as a compromise until a modified Memory Channel card is available.

We feel the Hemingway design will offer better performance than single-writer coherence protocols, and more importantly that the design offers better scalability. In the next section, we compare the Hemingway design to a single-writer lazy release consistency model and a variant of the TreadMarks [8] design.

2.4 Experimental Design

We constructed a cycle-level simulator for a multiprocessor system modeled after the structure shown in Figure 5. Although we are using DEC AlphaServer multiprocessors in our implementation, we used the Mint [27] system to build our simulations. We simulated 3 systems: our write-through model, a single-writer model and a DSM system modeled after TreadMarks.

Each processor has an 8KByte first-level (L1) data cache, and a 4 MByte second level (L2) cache. We did not model instruction references or the instruction stream. We assume physical memory can be divided into regions that are cachable and non-cachable. Furthermore, cachable memory is further divided into

regions that support write-back and write-through protocols. This model can be implemented on the Alpha architecture, where all levels of caching preserve the write-through or write-back semantics of the memory region. The L1 cache uses a write-around policy [13], where writes that miss in the cache do not allocate cache entries and all writes are presented to the L2 cache. Previous studies [2] have shown that for small multiprocessors (4-8 processors), most coherence protocols have similar performance. In our simulation, the L2 cache uses a write-back policy with processor consistency implemented using the Dragon protocol. Memory that is mapped for communication to remote boxes always uses a write-through protocol. Inter-box communication could use a write-back protocol, but we felt this would lead to large demands on the communication subsystem during memory synchronization, and did not investigate this design any further.

Each processor has a write-buffer consisting of four 32-byte entries. We used the allocation and flush policy described for the DECchip 21064 [10]. When the processor issues a write, the address tag is compared to each entry in the write buffer. If a match is found, the write is performed into that buffer entry, consolidating multiple stores to the same location. Also, stores to contiguous memory are combined into a write-buffer entry, reducing the off-chip bandwidth needed by the processor. More importantly, the Memory Channel hardware sends messages in blocks with bits to indicate which bytes are valid. This results in single-word writes to the memory channel being less efficient than writing a full cache line. Thus, the processor write-buffers can increase the effective network bandwidth. We did not modify the programs we examined to improve write performance, although it is possible to do so [5, 19].

Each distributed shared memory system we modeled used lazy release consistency. We used queue-based locks provided by Mint. In practice, synchronization between threads within the same box does not need to invalidate all pages, however, we did not use this information. The Memory Channel provides a fast barrier mechanism, but we did not model that in our simulations. Shared pages were interleaved across boxes; for example, on a two-box system sharing four pages, the first box would "own" pages 0 and 2, while the second box owns pages 1 and 3. More intelligent or adaptive policies for page placement and ownership transfer are possible, but are not considered in this paper. Threads were statically bound to processors, and each processor services a single thread. An actual system would use more threads than processors to mask communication latency.

The Hemingway system was simulated as described in the previous section. Remote memory pages are demand-fetched, and all shared pages are invalidated at memory barriers. We assumed the I/O bus bridge could snoop shared page references and pass those on to the Memory Channel. We did not simulate the overhead of duplicating store instructions nor the effect of duplicated store instructions on the bus bandwidth or write buffer. When processors updated a page, the new page was copied to the physical memory location of the original page; this avoids invalidating the TLB of other processors in most situations.

The single-writer protocol used a central directory mechanism to record the current write-owner of a page. The central directory was queried, and the current owner was then contacted. Full 8Kbyte pages were exchanged on write-ownership transfers. We did not simulate the TreadMarks system precisely; rather, we made modifications that better fit the Memory Channel hardware we assume. The TreadMarks system [14, 8] is similar to the Myrias computer. Pages are simply copied for read sharing; when a write occurs to a local copy, the page is duplicated, and a difference of the original page and the modified page is sent to the original page owner. The differences are used to integrate the changes of the multiple writers into a final copy. The Myrias computed differences using bit-wise exclusive-or. We do not know how differences are computed in TreadMarks, but it also compresses the differences using a run-length encoding to reduce bandwidth demands. This requires encoding both the data and the address of the data in the difference. Instead, we assumed word-level consistency and used the Memory Channel to send the different words. Essentially, this delays all memory writes until a memory consistency operation. Similar effects could be achieved in

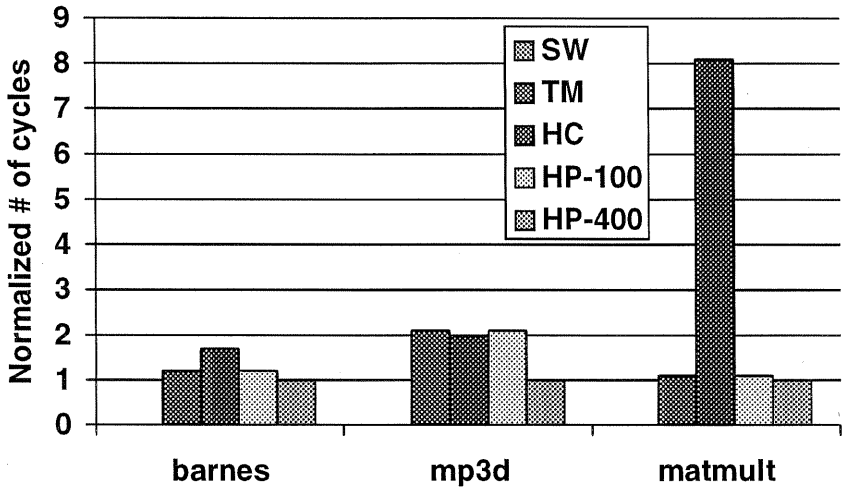


Figure 6: Execution time normalized to the HP-400 values for each application

the Hemingway system with very large on-processor write buffers or better compilers [5, 19]. However, the TreadMarks system must copy pages on writes, incurring considerable overhead to implement a multiple writer protocol.

The simulations modeled contention for the chip bus interface unit (BIU), L2 cache entries, the system memory bus, bulk memory, the PCI bus and the Memory Channel. The Dragon protocol was used for all processor coherence. Since instruction interlocks and latencies were not modeled, the inter-arrival times for memory requests were closer than they would be in practice, placing a greater strain on the memory subsystem. We assumed we could fetch remote pages without interrupting the processor; this benefits systems using many small transfers more than the page-based system we proposed.

3 Results from Simulation Study

We simulated the execution of three shared memory programs, 2 from the Splash benchmark suite [22] (Barnes-Hut and Mp3D) and a multi-threaded version of matrix multiply. Barnes-Hut (`barnes`) simulates the evolution of a system of bodies in a gravitational field. Mp3D (`mp3d`) solves a problem in rarefield fluid flow simulation. For each program, we chose a moderate problem size; the following parameters were used.

- `barnes`: 256 bodies
- `mp3d`: 10,000 molecules, 10 time steps
- `matmult`: 256×256 element matrix

The hardware configuration was fixed at four boxes each with four cpu's. We recorded the execution time, the number of page-level and cache-line transfers and the amount of write traffic across the Memory Channel. By recording the number of active words transferred in each packet, we can determine how effective the write buffers are at reducing write bandwidth. For each application program five protocols were measured:

- SW: Single-Writer
- TM: TreadMarks variation
- HC: Hemingway with cache-line sized reads
- HP-100: Hemingway with 8KByte page reads and 100Mbyte/s bandwidth
- HP-400: Hemingway with 8KByte page reads and 400Mbyte/s bandwidth

Because the simulation results vary by several orders of magnitude between the three application programs, we have normalized values for each application separately. When we display the experiments on the same bar graph, comparisons are only meaningful within clusters of bars, not between clusters. Detailed results follow.

3.1 Total Execution Time

The measure of total execution time is fairly imprecise because we did not model pipeline hazards or latencies. But since all simulated systems used the same assumptions, comparisons are possible. We also did not simulate the delay for interrupting remote processors for page and cache line fetches, since we were assuming our ideal hardware model. Normalized values of the execution times measured are shown in Figure 6.

Each application has entries for the 5 protocols, from left to right: SW[1], TM, HC, HP-100, and HP-400. The page level Hemingway protocols should improve as the bandwidth increases as is seen by the right side of the cluster falling off. In the `matmult` application a page contains 1000 matrix values and the cost of the read is overshadowed by the computations that must be done on the values read.

The baseline values (number of cycles for the HP-400 simulations) against which the data was normalized are:

- `barnes`: 13.8 million cycles
- `mp3d`: 36.9 million cycles
- `matmult`: 246 million cycles

3.2 Bandwidth Utilization

The Memory Channel interconnect has a nominal 100Mbyte/s bandwidth. Using the bus on the AlphaServer 2100, writes are blocked into 32-byte messages, the size of a single write buffer entry on the 21064 chip. The Memory Channel interface is presented with a series of 32-byte blocks, but not all words in those blocks are necessarily valid. We measured the number of valid words reaching the Memory Channel interface, since this influences the effective bandwidth.

Figure 7 illustrates the efficiency of the write buffer by showing the number of packets using 1, 2, ..., 8 words in the write buffer entry. The number of packets is expressed as a percentage. The `barnes` and `mp3d` applications used the write buffers inefficiently, while `matmult` is nearly perfect. This reflects the regular access patterns on static data structures exhibited by matrix multiply. The effectiveness of the write buffer in coalescing *all* writes, not just the writes to shared locations was improved when the size of the write buffer was increased from 4 blocks to 16 blocks. The execution time of `mp3d` dropped by 10% from about 37 million cycles to about 34 million cycles. This result confirms Kontothanassis and Scott [15].

3.3 Number of Small Writes and Page Transfers

The Hemingway system is designed to use short, bursty writes and to spread write traffic over the duration of a long program run. This reduces the latency of memory coherence points by reducing the queuing that would otherwise occur. However, this may use significantly more bandwidth because memory locations are written many times between synchronization points. Some of these additional or redundant writes can be eliminated using compile-time program transformations [5], but we have not investigated that in this paper. In the applications that we ran, `mp3d` issued a memory synchronization event on average every 78 thousand instructions, `barnes` every 110 thousand instructions and `matmult` only on program termination.

[1] The single writer simulations exceeded our hardware resources, failing to load on our largest machine (250Mbytes of virtual memory). We are in the process of acquiring resources that will allow us to try the SW experiments again. Although our data for this protocol is incomplete at this time, comparisons of single writer and TreadMarks protocols have been done in [8].

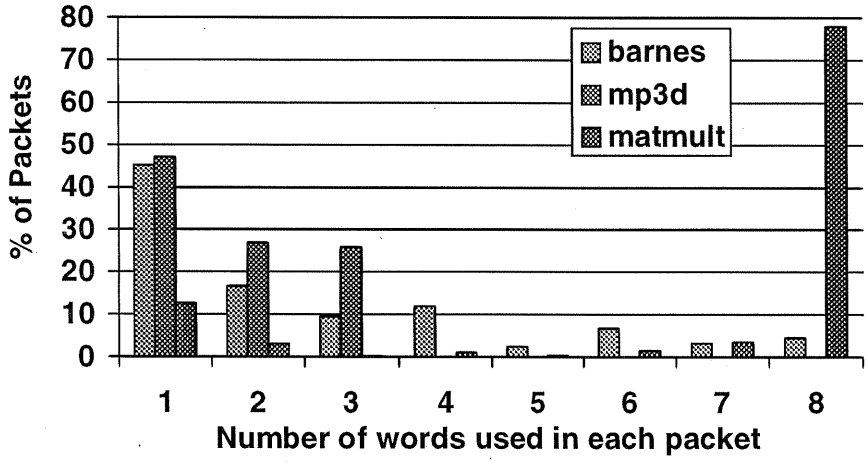


Figure 7: Valid words reaching Memory Channel interface while running simulations

3.4 Analysis

The performance of `mp3d` under Hemingway and TreadMarks is comparable. Since the synchronization events were issued relatively frequently and the average density of the packets sent across the Memory Channel was relatively low (Figure 7), the total number of TreadMarks diffs was small. The latency to update the "home" page at a synchronization point was therefore low. Our simulation did not model the cost of computing the TreadMarks diffs. Since diffs are computed on page aligned 8KByte memory segments and the L1 cache is also 8KBytes their computation causes continuous cache line misses and flushes.

It is interesting to note that for `mp3d`, cache line fetches result in a lower overall execution time. This was further confirmed by measuring the sharing (between processors in a box) of the pages mapped from other boxes. It was found that the total number of pages fetched was about 20,000 and the total number of cache lines used was about 100,000, or on average about 5 cache lines per page fetched. As was mentioned earlier, in order for page-mapping to be more beneficial than cache-line fetching with a 100Mb/s network, about 10 cache lines per page would have to be used. If these measurements are made for a 400Mb/s network, the page-transfer method should result in a significantly lower execution time. The cache line fetch method would not benefit in this way because its performance is dominated by the send/receive request time.

The performance of `barnes` under the different protocols is similar to `mp3d`, except that `barnes` performs marginally better under Hemingway with page-level transfers than cache-line transfers. This can be attributed to higher cache line use, both per processor and per box (on average about 1.5 - 2 processors accessed each mapped-in page).

About 215 cache lines per page are used by `matmult`. Fetching cache-lines on demand is extremely expensive and is reflected in the total execution time of HC (Hemingway cache-line). A comparison of the total execution times of this application under the Hemingway protocol and TreadMarks shows that the latency of the write-through is almost completely masked since the write-through can happen concurrently with processor execution. In our implementation of this program, the termination of the program did not result in a memory synchronization instruction. As a result in the TreadMarks protocol, no synchronization instruction had to be serviced, so the execution time is strictly for memory reads and writes. The comparable execution times under Hemingway and TreadMarks, therefore show that the Hemingway write-through of shared writes incurs no cost for the processor execution.

4 Conclusions, Status, and Future Work

At the current time, we have constructed a simulator for the Hemingway distributed shared memory system and used it to compare Hemingway to other distributed shared memory designs.

The simulations have shown that two important aspects of Hemingway lead to better performance.

- Hemingway scales with bandwidth
- Small multiprocessors are effective as basic building blocks of a DSM system

Latency is more expensive than bandwidth and will continue to be so for the forseeable future. Hemingway's performance is tied to bandwidth, not latency, as shown in the measurements with the 100Mbyte/s and 400Mbyte/s values for interconnection links. At the same time, the performance of other DSM protocols (whose performance is tied to latency instead of bandwidth) suffer.

When the basic unit of the DMS system is a small multiprocessor, the work to read a page is amortized over several processors rather than benefiting only a single processor. This increases the efficiency of the memory system.

We have an experimental infrastructure that should allow us to implement a rough prototype of the Hemingway system during the summer of 1995. We will use our existing AlphaServer-2100 4/275 multiprocessors, each with two processors and 512 Mbyte of memory. The systems will be connected with a beta-version of the Memory Channel interface – the interface and support software will arrive in early April.

We expect much of the implementation to be done in user space. We currently use the OSF/1 signal handling mechanism to map pages in user space. We are implementing the low-latency signal handling mechanism proposed by Thekkath and Levy [26] to make this kernel avoidance more efficient. We have hand-translated programs kernels to insure that we can correctly duplicate the store instructions in a program and have begun extending the OM package to automate this process. Our initial implementation, will only address memory to 4Gbyte, since this simplifies some design decisions in the store duplication and memory management.

While premature expectations concerning software are always rewarded with disappointment, we expect to have performance information from our initial implementation for simple programs by the end of the summer.

5 Special Note to Reviewers

This paper illustrates the design and some basic simulations of the Hemingway DSM as it compares to other existing systems. Unfortunately, during the course of these investigations, it became painfully clear that the resources required to thoroughly experiment with interesting parameters for problem size and network bandwidth are extreme. For example, the `barnes` HP-100 simulation is still running after 70 hours on a dedicated machine. For the final version of this paper, we intend to complete experiments using a wider variety of bandwidth values, processor sets (number of boxes and number of processors per box), and sample benchmarks.

Acknowledgments

This work was funded in part by NSF grant No. ASC-9217394, ARPA contract ARMY DABT63-94-C-0029 and an equipment grant from Digital Equipment Corporation.

References

[1] et al. Anant Agarwal. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

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

[3] Avalon Computer Systems. Memory system structure. (verbal communication), January 1995.

[4] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felton, and J. Sandberg. Virtual memory mapped network interface for the shrimp multiprocessor. In *21st Annual Intl. Symp. on Computer Architcture*, pages 142–154. ACM, April 1994.

[5] F. Bodin, E. D. Granston, and T. Montaut. Evaluating two loop transformations for reducing multuple-writer false sharing. CRPC TR94479, Rice University, October 1994.

[6] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth Intl. Conf. on Arch. Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[7] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS-V*, pages 51–61, 1992.

[8] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepol. Software versus hardware shared-memory implementation: A case study. In *21st Annual Intl. Symp. on Computer Architcture*, pages 106–117. ACM, April 1994.

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

[10] Digital Equipment Corperation, Maynard, Mass. *DECchip 21064 Microprocessor: Hardware Reference Manual*, October 1992.

[11] Daniel Gajski, David Kuck, Duncan Lawrie, and Ahmed Sameh. Cedar – a large scale multiprocessor. In *Proc. Int. Conf. on Parallel Processing*, pages 524–529. IEEE, 1983.

[12] Rick Gillette. Memory channel – an optimized cluster interconnect. In *Hot Interconnects '95*, August 1995. (to appear).

[13] Norm Jouppi. Cache write policies and performance. In *20th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 191–201. IEEE, May 1993.

[14] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Winter Usenix Conference*, 1994.

[15] Leonidas Kontothanassis and Michael L. Scott. Software cache coherence for large scale multiprocessors. In *Proceedings of 1st Conference on High Performance Computer Architecture*. ACM, January 1995. (Also available from ftp.cs.rochester.edu).

[16] Nancy P. Kronenberg, Henery Levy, and William D. Strecker. VAXclusters: a closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[17] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. Parallel Supercomputing Today and the Cedar Approach. *Science*, 231, February 1986.

[18] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proc. Seventeenth International Symposium on Computer Architecture*. ACM, 1990.

[19] R. Michandaney, S. Hiranandani, and A. Sethi. Improving the performance of dsm systems via compiler involvement. In *Supercomputing '94*, November 1994.

[20] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proc. of the 1993 ACM SIGMETRICS Conference*. ACM, May 1993.

[21] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*. ACM, 1994.

[22] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

[23] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*. ACM, 1994.

[24] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, March 1992. (Also available as DEC-WRL TR-92-6).

[25] A. Gupta T. C. Mowry, M. S. Lam. Design and evaluation of a compiler for prefetching. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, Mass., October 1992. ACM.

[26] Chandramohan Thekkath and Henery M. Levy. Hardware and software support for efficient exceptionx handling. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 110–120, Palo Alto, October 1994. ACM.

[27] Jack E. Veenstra and Robert J. Fowler. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 201–207, 1994.

[28] John Wilkes. Hamlyn—an interface for sender-based communications. HPL-OSR 92-13, Hewlett-Packard Research Labs, November 1992. Available from ftp.hpl.hp.com.