SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory

Andrew Erlichson†, Neal Nuckolls‡, Greg Chesson‡ and John Hennessy†

†Computer Systems Lab
Stanford University
Stanford, CA 94305

‡Silicon Graphics Inc.
2011 North Shoreline Blvd.
Mountain View, CA 94043

Abstract

One potentially attractive way to build large-scale shared-memory machines is to use small-scale to medium-scale shared-memory machines as clusters that are interconnected with an off-the-shelf network. To create a shared-memory programming environment across the clusters, it is possible to use a virtual shared-memory software layer. Because of the low latency and high bandwidth of the interconnect available within each cluster, there are clear advantages in making the clusters as large as possible. The critical question then becomes whether the latency and bandwidth of the top-level network and the software system are sufficient to support the communication demands generated by the clusters.

To explore these questions, we have built an aggressive kernel implementation of a virtual shared-memory system using SGI multiprocessors and 100Mbyte/sec HIPPI interconnects. The system obtains speedups on 32 processors (four nodes, eight processors per node plus additional reserved protocol processors) that range from 6.9 on the communication-intensive FFT program to 21.6 on Ocean (both from the SPLASH 2 suite). In general, clustering is effective in reducing internode miss rates, but as the cluster size increases, increases in the remote latency, mostly due to increased TLB synchronization cost, offset the advantages. For communication-intensive applications, such as FFT, the overhead of sending out network requests, the limited network bandwidth, and the long network latency prevent the achievement of good performance. Overall, this approach still appears promising, but our results indicate that large low latency networks may be needed to make cluster-based virtual shared-memory machines broadly useful as large-scale shared-memory multiprocessors.

1 Introduction

Hardware implementations of directory-based coherency protocols provide good performance on small-scale to medium-scale multiprocessors. A particular hardware implementation, however, is typically built to scale to a fixed number of processors. It appears, due to the need to have competitive low-end offerings in the "sweet spot" of the market demand as well as the relatively small market for very large machines, that alternative methods for organizing very large distributed shared-memory machines (greater than 100 processors) may be needed. Software shared-memory protocols have the advantage that they can be used to create a shared address space across an arbitrary number of computers connected with a general purpose network. It has been proposed [4,28] that large multiprocessors be built by connecting hardware-based cache-coherent multiprocessors using software shared memory. Since software shared memory has been shown to be scalable to a small number of processing nodes[17], it would seem well suited to connecting nodes of multiprocessors.

In this paper, we investigate the viability of building multiprocessors using a hybrid scheme whereby software shared memory is used to connect hardware-based shared-memory multiprocessors. Cache line coherency is maintained within the node; page coherency is maintained across the nodes. The system presents the user with one large coherent virtual address space.

Building large machines from networks of hardware multiprocessors has several advantages. The software system extends the shared memory programming environment available on hardware multiprocessors to multiple machines. Processors within the clustered node communicate via the highly efficient hardware shared memory; long software latencies are only incurred when going off-node. Communication between the nodes is in page-sized chunks, amortizing the high per-message overhead of the general purpose network. Between nodes, the flexibility of the software environment permits the implementation of different protocols and adaptive strategies and a larger degree of scalability.

We have implemented an operational prototype of a distributed virtual shared-memory (DVSM) system to investigate the viability of software shared memory in such clustered multiprocessors. The system is comprised of SGI Power Challenge machines connected using a 100Mbyte/sec switched HIPPI (HIgh-Performance Parallel Interface) high speed LAN. The node machines are bus-based multiprocessors; a snoopy invalidation protocol, implemented by the SGI hardware, provides coherency within the node. Software virtual shared memory (our implementation) provides coherency among the nodes. The Translation Lookaside Buffer (TLB) detects access violations and initiates software coherency protocol actions. The software shared memory is implemented aggressively within the Irix 6.2 Unix operating system. We call our prototype system

ASPLOS '96 10/96 MA, USA

SoftFLASH, since it implements the Stanford FLexible Architecture for Shared Memory (FLASH)[18] protocol entirely in software.

Our investigations reveal that clustering is effective in reducing the internode per-processor communication as expected. On the other hand, we find that increases in the latency of memory operations as cluster size is increased offset the advantages. These increases in latency are largely due to the increased cost of TLB synchronization in larger hardware-based machines. Also, we find that since the clustered machine must support the aggregated bandwidth of its constituent processors, the overall bandwidth required between clusters increases with the size of the cluster. In bandwidth-intensive applications, this can become the limiting factor in performance if per-node network resources are not scaled.

The rest of this paper is organized as follows: Section 2 introduces the prototype system and discusses our major design choices. Section 3 reports the cost of the primitives in our system. Section 4 presents the performance of our system for configurations of up to 32 application processes, analyzing the effects of clustering and other design decisions on the performance. This section concludes by summarizing the bottlenecks in the prototype system. Section 5 discusses related work. Finally, in Section 6 we conclude.

2 SoftFLASH Prototype System

SoftFLASH provides a limited set of primitives to the application programmer: shared memory, mutual exclusion locks and synchronization barriers. Our system assumes that programs are written in a synchronized style where all references to shared memory are protected by locks or barriers. The prototype provides limited extensions to permit application programs to synchronize on memory flags without mutual exclusion locks under Delayed Invalidate Release Consistency[8] (DIRC). The application is expected to fork into parallel threads which work together to solve a problem, communicating via shared memory and synchronizing using locks and barriers.

SoftFLASH supports prefetching of page data through application function calls. The prefetching mechanism can be used to hide the latency of long operations. From the standpoint of the system, prefetching calls are identical to regular memory references once outside the requesting node.

2.1 Key Design Choices

Two Level Design

The prototype system is clustered throughout its design. From the standpoint of the directory protocol, each clustered node, with its many processors, is considered as one sharer. Processes within a node share memory using only hardware by having all processes within the node share a common page cache and page table. Once one process on a node has faulted a page into its address space in the writeable state, no further software overhead is incurred to share the page. Additional processes on the node can read and write data from that page by taking only cache misses and TLB refill misses, the normal sharing mechanism for shared data on a multiprocessor. This hierarchical approach eliminates trips across the network when sharing data only among processors within the node. Problems that exhibit frequent nearest neighbor sharing are expected to run well on such a system since the common case of communicating between neighbors is handled by the hardware shared memory. Applications that perform simulations of physical phenomena using a spatial decomposition often exhibit nearest neighbor sharing.

Locks are cluster-based as well. We implement queuing directory-based locks. The system assigns each lock to a home, and maintains information about the lock state in a lock directory at the home. Initially, a process must send a message to the home to get the lock (unless the home is local). After the lock is obtained by a process, other processes within the node may pass it back and forth without contacting the home. If a process on another node needs the lock, it sends a message to the home requesting the lock. The lock directory at the home forwards the message to the node holding the lock (possibly local). If the lock is available on that node, the network handler immediately sends it to the requestor. Otherwise, the lock will be sent to the original requestor by the next process to release it. The locking protocol is similar to the FLASH shared memory protocol, with two changes: all requests are the equivalent of write requests and the negative acknowledgment (Nack) mechanism that is used to break contention for memory is replaced by a queue of waiters at the home.

Transparency of Shared Address Space

Much work has been done on software shared memory using explicit program annotations to access the shared named space [2,5,13,19,23]. The difficulty of programming shared-memory machines using specific annotations, however, has hindered the widespread adoption of those systems. Hardware shared memory, while easier to program, is only available on medium-scale systems. To naturally scale application codes to larger systems without porting, transparent shared memory is required. Hence, we implement a page-based scheme that permits the user to access shared memory with no annotations other than locks and barriers.

The software shared memory is transparent to the user application. The prototype supports a limited soft fork that permits applications to inherit the memory state of the non-shared portions of the address space at the time of fork. This facility permits us to run applications written to the fork model unchanged. Most of the SPLASH II[27] programs can be run unmodified on the system, requiring only recompilation and a different set of Argonne National Laboratory (ANL) macros[10]. (Some applications did require moving the ANL initialization functions.) The ability to run unmodified applications eliminates tedious porting and permits users to program to a single interface that is already familiar to them.

Relaxed Memory Consistency

We implement the memory consistency modes that are used by FLASH. The prototype supports the two FLASH modes, Delayed and Eager. In Delayed mode a writer must wait until all invalidates for a coherency object have been acknowledged before continuing. In Eager mode, the writer may continue as soon as ownership is transferred and the data has returned. Eager mode, a form of release consistency[11], permits the faulting cache to retire writes before all invalidation messages have been acknowledged. At the next release operation, the process must wait until all previous writes have been acknowledged before proceeding.

In the FLASH implementation of Release Consistency, a single counter per node is used to keep track of the number of outstanding writes. Such a scheme does not work within the confines of our shared software page cache. If a page is brought in Writeable to a node, any other process may immediately write to it, with only the delay of a TLB refill exception. A single counter for outstanding misses will not work, because the counter may never go to zero if other threads continue to execute writes and increment the counter. A simple per-process counter scheme will not work either, since updating the counter for a process requires detecting when write hits (in the TLB) occur to pages that have outstanding invalidation requests caused by another process on the node. It is not possible, on existing hardware, to change the mechanism a processor uses on a write hit to the TLB.

Instead, we introduce a new multiple counter scheme, *Epoch-based Release Consistency*, that permits delay-free access to shared data within a node while maintaining the semantics of release consistency. Epoch-based Release Consistency is an extension of Release Consistency to any system with shared caches (in our case, it is a shared software page cache). Write references from all processors on a node are divided into epochs, demarcated by release operations performed on that node. For each epoch, there is a counter. The epoch counters are implemented as a software ring, one counter per process (if only one process exists, it degenerates to the FLASH scheme). Each write miss is assigned to the current epoch and tagged with its identifying ID. Upon a write miss, the counter at the requesting node corresponding to the current epoch is incremented. When the write miss globally completes, the same epoch counter (possibly not the current one any longer) is decremented. At a synchronization release point, the current epoch ends, and a process waits until that epoch and all previous epochs have been cleared. Hence, we enforce the rule that a process must wait for all writes that occurred earlier in time from any process in the node. This constraint is sufficient to guarantee correctness. While a process waits for an epoch to clear, other processes can continue to execute writes, which only affect subsequent epochs.

In addition to the FLASH modes, we implement a modified form of Delayed Invalidate Release Consistency[8] (DIRC) that postpones the processing of invalidation messages on a remote node until the next synchronization acquire point. We implement DIRC partially to reduce false sharing, but mainly to reduce the impact of invalidation traffic on performance. By grouping all the invalidations together and processing them at once, we are able to perform one TLB synchronization operation for the many invalidations. TLB synchronization is very expensive on hardware MP systems because it requires interrupting all the processors on the system to remove a mapping (see Section 4.4 for more information).

Kernel Level Implementation

Our implementation runs entirely inside the Irix 6.2 Unix kernel, with the exception of the directory-based locks, which run at user level using persistent TCP socket connections. Irix 6.2 is a modern 64 bit Unix operating system derived from BSD Unix and System V. The default page size within Irix 6.2 is 16kbytes for multiprocessor systems. Coherency faults are taken by the standard virtual fault handlers inside the kernel and passed directly to our SoftFLASH handlers, permitting rapid dispatch of the coherency violation without the long delays normally associated with running fault handlers in user-space via the unix “signal” mechanism. Once inside our handler, we manipulate the virtual memory structures directly.

For network communication, we implement a software layer that resides in the kernel driver for the HIPPI network interface. On the receiving side, network messages destined for coherency handlers are dispatched using the software interrupt mechanism of the MIPS processors. A network hardware interrupt initiates the process by scheduling a software interrupt to run the protocol message on a processor.

Sending and receiving pages from one node to another involves no memory-to-memory copies within a node. The sender starts the network DMA directly from the user’s physical memory. On the receiving side, the page is put into a page-aligned buffer directly by the network card via DMA. The kernel then maps the page into the user’s virtual address space. Such an optimization would be impossible with the user level primitives available in the Irix operating system and most commercial Unix operating systems.

A kernel implementation also provides extensive monitoring and statistics gathering opportunities that are not available at user level in Irix 6.2.

Parallelism of Network Request Handling

Because the underlying nodes in our systems are themselves multiprocessors, we designed the protocol code that handles network messages to be highly concurrent. We provide directory and page cache locking on a per-page basis. To limit the number of locks needed, the page number is hashed to one lock from a large pool of locks. Concurrent operation of the handlers is achieved for all cases except those where there are concurrent operations on the same page or a conflict in the lock hash table.

2.2 Implementation Details

Each node contains between 16 and 18 MIPS R8000 processors running at 90Mhz (310 SPEC FP). The R8000 processor contains 16 kilobyte primary data and instruction caches and a three-way set associative 384 entry TLB. The primary caches are backed up by four megabyte 4-way associative unified secondary caches. A shared one Gbyte/sec snoopy bus connects the processors. Most nodes are configured with four gigabytes of main memory.

The nodes contain SGI HIPPI I/O boards connected to a HIPPI switch, manufactured by Essential Communications. The SGI implementation of the HIPPI network I/O board limits the aggregate bandwidth between the board and the host to 100Mbyte/sec, although the board is capable of running at 100Mbyte/sec in each direction. This limits the bandwidth to 50Mbyte/sec for one-way traffic if there is opposing traffic at the same time. DMA I/O in the SGI machines is cache coherent, making it a good candidate for virtual shared memory.

From a software standpoint, our shared-memory space resides within the virtual memory system as a mapped file system, an idea taken from Minnich’s Mether-NFS system[21]. Memory coherency on a page is enforced by removing a processor’s ability to translate virtual addresses into physical addresses. Globally shared pages maintained by SoftFLASH are kept within a shared software page cache in main memory on each node. The FLASH protocol requires each page to have a home. On the home node, a page’s physical memory will be maintained even if all mappings to the page on the node are invalid. On all other nodes, the node de-allocates the page when it invalidates all virtual-to-physical mappings to the page.

Application processes on a node can run on any physical processor, if permitted by scheduling. Hence, a page mapping, once established, can wind up in any one of the physical TLBs of the various processors within the node. The MIPS memory translation mechanism has several layers of TLB miss handlers. At the lowest level, a processor will execute a fast TLB refill operation (utlbmiss) whenever a memory reference attempts to access a page for which there is no mapping within the processor’s TLB. The processor takes an exception and executes the utlbmiss handler in kernel context. That handler refills the TLB with the appropriate page table entry from the shared software page table. The refill handler does not check, however, that the entry is valid. If it loads an invalid entry, it will take an additional exception when the processor is restarted. It is this exception which is caught by a more general handler and passed to the SoftFLASH handlers for resolution.

Page Cache States

Each node contains a shared page cache in main memory which contains tags for all locally cached and homed pages and pointers to the physical pages, if present. The states supported by the page cache on each node are in Table 1.

Table 1: Page Cache States

State	Meaning
Readable	Page is readable by all processes on node
Writeable	Page is writeable by all processes on node
Invalid	Page is not accessible
Inval_Occur	Invalidation received on pending page

In addition, regardless of current state, any of the pages can be Pending. A page is Pending if a request is outstanding for it. Any time a page is Pending, all subsequent misses will merge with the outstanding one, even if the outstanding miss will not resolve the new miss (e.g., read miss, then write miss). The page cache state is kept separate from the kernel page table entries to afford parallelism of access. Within Irix, the page table of a process is not very multithreaded. Hence, we shadow its state for the sake of concurrency.

Directory States

The prototype supports the directory states specified by the Stanford FLASH protocol. The hashed per-page locks that provide mutual exclusion to directory entries also provide mutual exclusion to page cache entries, making it possible to lock both the directory entry and the page cache entry for a page at the same time. This differs from the Stanford FLASH machine, where communication between the directory on a machine and the processor cache is accomplished via two-way queues. The ability to lock both the directory state and the page cache state on a node at the same time substantially simplifies pieces of the FLASH protocol, which normally must deal with the case where the local page cache state is different from what is reflected in the directory. For example, in FLASH, the directory may believe the local cache has the data, but it may not actually have the data if it has already been written back by the cache, but the message has not yet arrived at the directory. Essentially, the FLASH directory-based protocol must treat all caches, even the local cache, as being accessible only through ordered two-way networks. We treat the local cache state as being logically locked along with the local directory state. This is one of the advantages that software affords.

3 Cost of the Primitives

Our experimental system contains 4 HIPPI connected nodes, each one with up to 16 processors. Fundamentally, to keep the software shared-memory layer coherent, the system must take faults, manipulate the virtual memory structures and move data across a network from one physical node to another. Table 2 contains data measured from test programs designed not to stress the network but only to exercise the primitives. This method gives us the lower bound latencies of the machine.

Table 2: Selected SoftFLASH Primitives

Primitive	Latency in µs and in pclocks (90Mz)
Null Read Fault	41µs (3,690)
Null Write Fault	37µs (3,330)
TLB Invalidate (1 cpu/node)	21 µs (1,890)
Barrier (4 nodes, 1 process/node, 16 cpus/node)	3230 µs (291,000)
Barrier (2 nodes, 16 processes/ node, 16 cpus/node)	5860 µs (527,000)
Latency of Remote Read. Page dirty at the home, 16k page. (2 nodes, 1 process/node, 1 cpu/ node)	1164 µs (105,000)
Latency of Remote Read. Page dirty at the home, 16k page. (2 nodes, 1 process/node, 8 cpus/ node).	1419 µs (128,000)

The first two primitives measured are the *Null Read Fault* time and *Null Write Fault* time, the time it takes for the operating system to detect a fault and pass it to the coherency handlers, plus the time it takes to return and restart the process. These values are measured by timing the inner loop of a test program designed to repeatedly fault and then subtracting the time to handle the fault as measured by the statistical counters within the operating system. In general, all of our other statistics are measured within the OS and do not include this time.

TLB Invalidate, the next entry, is the time it takes a MIPS R8000 processor to invalidate a single TLB entry on a single processor system. This is the mechanism we use when reducing the privileges of page to enforce coherency. TLB Invalidate will be called when reducing privileges down from Writeable to Readable and from Readable to Invalid. Since we implement a single-writer, invalidation-based protocol, TLB Invalidate will be called any time ownership changes. As such, it is a very important primitive.

The structure of the MIPS TLB is such that it is unknown which processors' TLBs contain the mapping. The TLB is constructed this way because, under normal system operation, changes in the page tables are relatively rare, and hence TLB synchronization is rarely required. To synchronize the TLBs on a multiprocessor, the kernel must perform a cross call to interrupt all the processors that might contain the mapping. As an initial optimization, we changed the implementation of the TLB Invalidate mechanism to invalidate the TLBs of only those processors which have at one time run a DVSM process from the current application. The cost of TLB invalidation is explored further in Section 4.4.

The next primitive measured, *Barrier* Time, is the time for all application threads to rendezvous. The primitive is measured in two ways, first under low load, and then under slightly higher load. The cost of a loaded barrier, over 5ms, is substantial, but the barrier primitive is not heavily used in the application programs we examine. The barrier code is not clustered: it sends one message per process that arrives at the barrier, rather than having the node send one message for all processes. We expect that the barrier time for the 16-processes-per-node case could probably be reduced to close

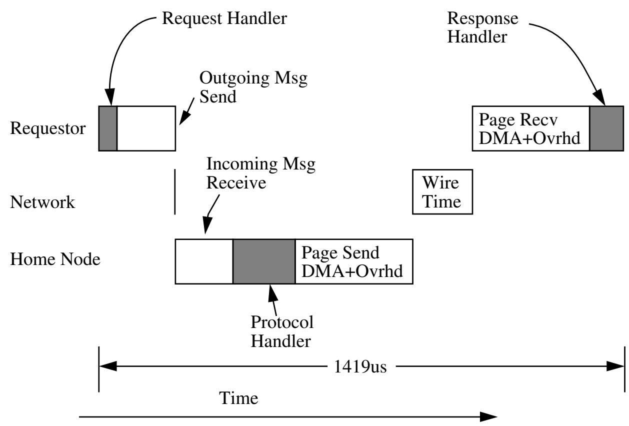


Figure 1: Breakdown of Time to Resolve Read Fault, Page Dirty at the Home

to that of the one-process-per-node case by clustering the barrier implementation.

The next entry in the table is the latency to perform a *Remote Read Miss* of a 16kbyte page where the state of the page is Writeable at the home. For this measurement we use two nodes, each with all but one of its processors disabled. A remote read fault, where the page is writeable at the home, is extremely common in our system, since our page cache resides in a large main memory and suffers no capacity misses. All misses are, therefore, either communication, cold or false sharing misses. Communication and false sharing misses are often misses on pages that are in the Writeable state at the node of the last writer. With good data distribution, the last writer is often the home. This latency, 1164µs for a 16kbyte page, compares favorably with latencies recorded on other systems. For example, TreadMarks, a user-level implementation on a uniprocessor system, reports a latency of 1956µs to fault a 4kbyte page over ATM[15]. A user-level implementation of SoftFLASH (done for debugging), which runs the same protocol code, moves a 16kbyte page in 3216µs over HIPPI (150 Mhz R4400-based systems). SoftFLASH (kernel level) moves 4kbyte pages on four-cpu R4400-based systems in about 1100µs. Clearly, a kernel-level implementation of DVSM offers substantial performance advantages over a user-level implementation.

The next measurement in the table is again the cost of a remote read page fault, but on two eight-cpu systems with separate processors dedicated to NI processing. Having separate processors for NI processing requires sending cross-call interrupts to handle protocol messages. It also involves shooting down more processor TLBs. The latency increases to 1419µs from 1164µs. This number is a better estimate of the lower bound latency available to the applications, since even small runs with small numbers of processes per node are run on large machines in our study.

The latency for a remote read implies that the maximum bandwidth which a process can sustain without prefetching is approximately 10Mbyte/sec. Since our network link is 100Mbyte/sec shared, we expect that there is some expansion room to support the bandwidth requirements of additional processors.

Where does the time go in a 1419µs read miss? Figure 1 contains an approximate breakdown of where the time is spent in a read request between two 8-cpu nodes. Protocol code makes up only a small part of the total time. One of the limiting factors in performance is that the HIPPI network card does not begin a frame transfer until the entire payload has been moved from host memory to the card. Hence, much of the latency is serialized in a request. This points to one of the drawbacks of general I/O networks, versus tightly coupled hardware shared-memory networks; they tend not to wormhole routed. The largest part of the latency that could be eliminated with further tuning is network message send and receive overhead. The HIPPI network does support the transfer of one frame from host memory to the card, while another frame is being transferred over the wire.

The prototype uses the HIPPI Link Encapsulation layer (LE) for communication. HIPPI LE, the same layer used to implement TCP, is not the ideal communication layer for DVSM, but is used for the sake of rapid prototyping. LE packets generate hardware interrupts from the HIPPI board to a fixed processor. That processor then interacts with the HIPPI board to receive a pointer to the buffer used by HIPPI for receipt of the data.

Table 3: Applications

Code	Problem Size	Growth Rate of Comm. to Comp. Ratio
FFT	4 and 16 million complex points	$(P-1)/(P\log DS)$
LU	2048x2048	$(\sqrt{P})/(\sqrt{DS})$
Barnes	32k Particles	$(\sqrt{P})/(\sqrt{DS})$
Ocean	2050x2050	$(\sqrt{P})/(\sqrt{DS})$

4 Application Results

The problem sizes and communication to computation growth rates of the applications used in the base study are shown in Table 3 (*DS* is the problem size, *P* is the number of processors). We ran a subset of applications from the SPLASH-2 suite of applications: FFT, Barnes, LU, and Ocean. FFT[25] performs a fast Fourier transform on a 1D array of complex points. It is known to have very high communication demands and typically does not run well on software-based systems[15]. Barnes[12] simulates a system of particles using the Barnes-Hut hierachical N-body method. LU[26] is a linear algebra kernel that factors a matrix into two matrices; one lower triangular, the other upper triangular. It is known to have a relatively small communication to computation ratio. Ocean[26] simulates large ocean currents. It is interesting because it exhibits nearest neighbor sharing, and should therefore be an excellent candidate for a clustered system. For an overview of the SPLASH applications and their characteristics see [27].

The applications were chosen for a variety of reasons. First, they represent a wide spectrum in terms of communication to computation, with FFT known to have high communication demands, that drop off slowly as problem size is increased, and LU known to have relatively light communication demands. Second, they all provide easily scalable problem sets, permitting us to run large problems. Third, they do not exhibit any pathological false sharing problems with a single writer protocol. One SPLASH application, radix, can be easily scaled and would be interesting to run, but is known to suffer from bad false sharing at large coherency block sizes[27].

All runs were performed on one to four (16 to 18)-processor R8000 machines in various clustered configurations of up to eight

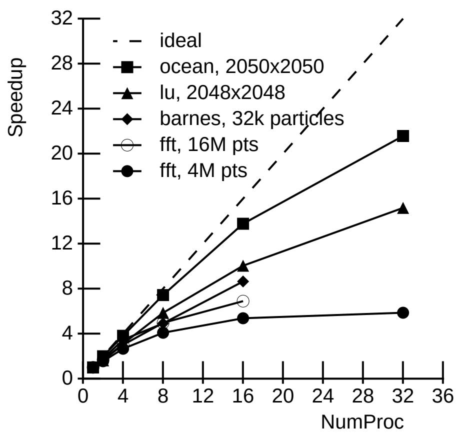


Figure 2: Overall Speedup

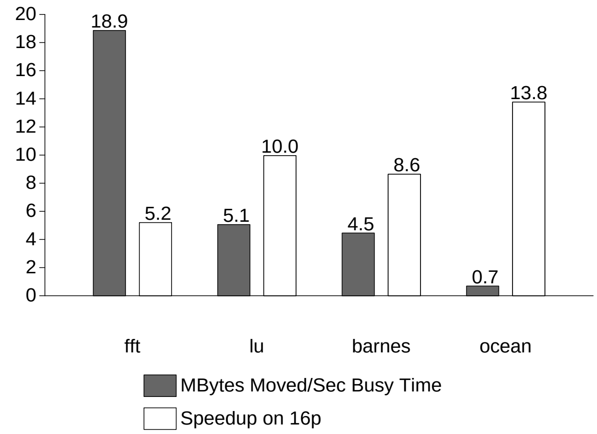


Figure 3: Traffic to Computation and Speedup

application processes per node, for a total of 32 application processes.

4.1 Baseline Results

For the baseline results, the system is configured to run with Delayed Invalidate Release Consistency (DIRC). In DIRC mode a node receiving an invalidation will enqueue it until the next synchronization lock acquire or barrier access, but immediately acknowledge it to the sender, so the directory protocol remains unchanged from the FLASH protocol. When the receiving node processes the queued invalidations, all affected page table entries are invalidated and then one TLB synchronization is performed, invalidating all TLB entries. Up to 200 invalidation messages are allowed to enqueue before they will be automatically processed to make room on the pending invalidation queue. As implemented, DIRC is an extension to the FLASH Eager mode. When running in DIRC mode, the fault handler performs a lookup on every write miss to see if the write miss conflicts with a pending invalidation message. If so, the invalidation message is immediately processed to avoid violating the underlying directory-based protocol, which assumes that a node has seen an invalidation once acknowledged. As stated earlier, DIRC has two advantages: it will reduce Reader/Writer false sharing and it will reduce the cost of keeping the TLBs synchronized.

The system as configured dedicates four additional processors to network interrupt (NI) processing. Those processors do not run the application and are not counted for purposes of assessing speedup. In addition, another processor is reserved just to take the low level HIPPI hardware interrupt. Application processors still participate in some low level HIPPI driver tasks when sending a message but they are never interrupted to run a protocol handler. NI messages are sent round robin to the dedicated processors by interrupt calls from the low level HIPPI hardware interrupt. Once interrupted, a processor continues to process NI messages until the incoming queue is empty, so load balancing of the NI messages onto the NI processors may not be achieved. The impact of having extra processors reserved for NI processing is considered later in Section 4.3.

We run each application in a variety of clustering configurations with a range of processors. For example, we ran two 16 process runs, one with two nodes and eight processes per node, another with four nodes and four processes per node (plus dedicated NI processors). The best results were always obtained by minimizing the number of nodes and maximizing the number of processors per node. The overall speedup results for 4NI-DIRC appear in Figure 2 for the best clustered configurations. FFT is shown for two different problem sizes. The FFT runs are with user controlled page prefetching turned on.

Ocean performs well up to 32 processors, obtaining a maximum speedup of 21.6 on 32 processors. As we will see, it is mainly because of its low communication to computation ratio. LU and Barnes perform similarly, obtaining speedup of about 8 on 16 processors. FFT, the communication-intensive application, performs poorly both for the 4 million and 16 million point problem. Increasing the problem size in FFT does not help much, since the computation to communication grows as $O(\log n)$.

The performance of the applications is well predicted by the traffic to computation ratio. Figure 3 contains a comparison of the speedup achieved and the traffic to computation for 16-processor runs (4 node by 4 processes per node and dedicated protocol processors). The traffic to computation is taken to be the total number of Mbytes moved per node per second of busy time, where busy time includes all local cache miss time but excludes all time spent blocked waiting for remote memory. Traffic to computation is related to the inherent communication to computation but is affected by the coherence granularity and the degree of latency hiding.

As we can see, FFT exhibits the highest traffic to computation ratio by far, and achieves the correspondingly worst speedup. LU and Barnes have traffic to computation ratios that are in the middle range, and have similar speedups, achieving an overall processor utilization of about 50%. Ocean, which performs best, has the smallest traffic to computation ratio. The prototype system can not support the communication demands of applications like FFT.

Figure 4 shows a normalized breakdown of the execution time for the four applications (16M pt. FFT shown), for four node by four processor runs (16 processes total). *Read* time is the time spent blocked on read requests. *Write* time is the time spent blocked on write requests. *Lock* time is the time spent waiting for application mutual exclusion locks. *Fence* time is the time spent waiting at a barrier or lock release for all previous invalidation messages to be acknowledged. *PreOver* is prefetch overhead, and includes the time to make the prefetch syscall into the kernel from the application, and send the outgoing protocol message. *Busy* time is the time remaining when all other time is subtracted. It includes time spent in hardware cache misses.

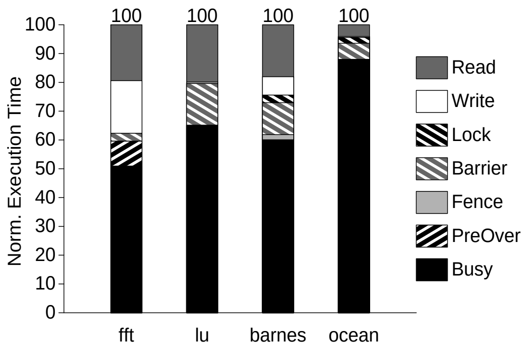


Figure 4: Execution Time for 4 Node by 4 Processor Run (4x4p), DIRC, 4NI

For FFT, a large percentage of time is spent in both Read, Write and Prefetch Overhead. All read misses are page-prefetched but many of the prefetches do not return to the node in time, resulting in some remaining miss latency. The Write and Prefetch Overhead is somewhat surprising. FFT distributes its data among the nodes so that all Write misses will be local upgrade misses (misses to pages which are Readable at the node and homed locally). Such misses only require sending invalidation messages, updating the page from Readable to Writeable and returning. The long latency experienced by those misses, an average of 667µs per miss, is due to an artifact of the low-level HIPPI network driver used. When adding something to the outgoing queue on a message-send operation using the HIPPI LE layer, the sending processor helps in low-level network input processing, until the input queue is empty (The low-level processing does not include running protocol handlers, which are run only on the dedicated NI processors). The driver is designed in this way to avoid hardware interrupts when there is outgoing traffic. While the design may improve the overall efficiency of handling the network (and decrease end to end latency), it adds to the latency of sending messages. The amount of input processing to be done is greater in FFT, since the communication rate is high. Both the write misses and the prefetch operation send network messages. The high overhead of sending messages substantially reduces the speedup of FFT on 16 processors (speedup 6.7).

LU's speedup is limited by time spent in read miss and time spent waiting at barriers. The barrier time is partially caused by an asymmetry in the number of read misses taken by node zero (the start-up node), versus nodes 1-3. Node zero takes approximately one half the number of read misses as the other nodes, possibly due to the data being warm on that node.

Barnes has about half of its stall time spent in read misses, with the rest evenly split between write and barrier time. Barrier time is partially a result of vastly differing numbers of read communication misses on the different processors, and a high degree of "nacked" protocol messages (between 10 - 30%), indicating contention for directory entries.

Ocean spends a small percentage of time in both read and write misses, as expected from its low traffic to computation ratio, leading to its good performance (speedup of 13.8 on 16 processors).

4.2 Clustering Effects

To analyze the effects of clustering, we compare the results of two eight-processor runs, the four node by two processor (4x2p) run and the two node by four processor run (2x4p), which differ only in the degree of clustering, and use eight processors for computation. All runs include extra dedicated NI processors. We find that clustering reduces the internode communication rate dramatically, shown in as Figure 5a, but that the reduction in miss rate does not translate into a commensurate improvement in overall performance, as shown in Figure 5c.

We start by examining the overall clustering effects for FFT. We see that the number of misses taken by the application drops when clustered by about 20%. The miss latency per miss, however, shown in Figure 5b, increases when clustered by nearly a corresponding amount, countering the effects of the reduced miss rate. The overall result, shown in Figure 5c, is that the execution time goes down by only about 2%.

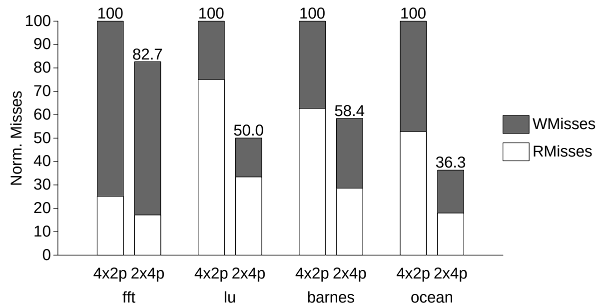
Analytically, FFT exhibits all-to-all sharing[9]. Hence, in the 4x2p case, each process communicates with six off-node processes. When more highly clustered, in a 2x4p configuration, each process communicates with four off-node processes. Hence the per-process communication miss rate should go down by a factor of 4/6. However, the more highly clustered 2x4p configuration contains twice as many processes per node, and hence the overall per-node remote traffic should increase by a factor of 8/6 or 1.5. Given that the execution time goes down slightly and the traffic is predicted to go up, it would seem likely that an increase in bandwidth is causing the increase in latency; But, as Figure 5d shows, the overall traffic remains the same when going from 4x2p to 2x4p for FFT. The traffic remains the same because there is fragmentation using 16kbyte pages for this problem size. When there are more processes per node, a greater portion of each page brought in to the node is used, reducing the overall miss rate. Effectively, this is a clustering effect, whereby one process prefetches data for another.

The increase in Read latency in the more highly clustered FFT run can be partially explained by the increased cost to synchronize the TLBs when more processors are involved in the computation. Synchronizing the TLBs, by invalidating a TLB entry in all processors which may contain it, is necessary whenever the privileges to a page are downgraded from either Writeable to Readable or Readable to Invalid. When a larger number of processors participate in the computation, a larger number of processors must be interrupted for TLB synchronization. Section 4.4 contains measurements and analysis of the TLB synchronization effects. The remaining increase in the miss latency is due to increased overhead in the messaging and dispatch layer.

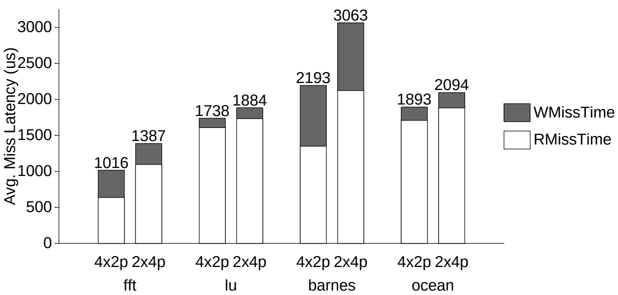
LU exhibits a dramatic reduction in the miss rate when clustered. The latency of read faults increases by a small amount, due to the increased cost of TLB synchronization. The result is that LU sees a 10% reduction in execution time when clustered. The per-node traffic increases when clustered, which is generally the expected result when clustering, since the number of processes within the node increases, but indicates that the application is not seeing substantially less data fragmentation when clustered.

Barnes's miss rate also goes down dramatically when clustered. That effect is partially offset by an increase in latency. The large increase in average miss latency is due to a shift toward longer misses. The 4x2p run has a higher percentage of short "nacked" requests, which largely disappear in the 2x4p run. The remaining misses are longer real misses. Overall, the execution time still goes down by about 10%. Barnes, like LU, sees an increase in traffic, and hence, per-node bandwidth required (since the execution time goes down).

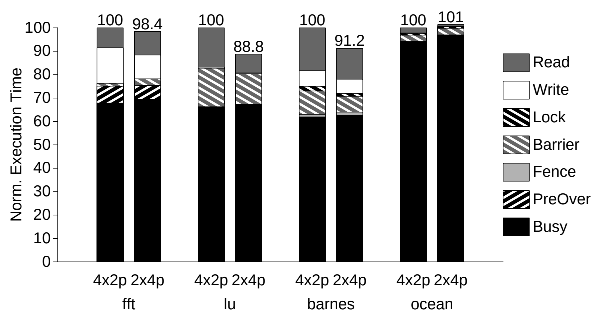
Ocean exhibits a dramatic reduction in the communication miss rate, as expected since it exhibits nearest neighbor sharing. It suffers a slight increase in the read latency, once again due to the higher cost of TLB synchronization in the 2x4p run. Overall, Ocean



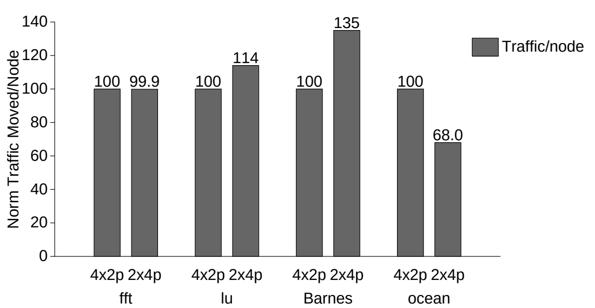
a) Normalized Misses



b) Miss Latency



c) Execution Time



d) Traffic Per Node

Figure 5: Clustering Effects

experiences a small increase in the execution time, because it spends practically no time communicating between nodes. Ocean's reduction in miss rate is so dramatic that per-node traffic goes down when clustered. Ocean is very well suited to clustering, but does not communicate enough to see any measurable advantages.

In summary, clustering is very effective in reducing the internode miss rate for the applications studied, but increases in the cost of communication when clustered and low communication miss rates mitigate the effect on execution time. In more highly clustered configurations, bandwidth eventually becomes a limiting factor, increasing the costs of communication even further.

4.3 Impact of Extra Dedicated NI Processors and DIRC

In this section, we investigate the performance impact of dedicating extra processors to input processing and of using Delayed Invalidate Release Consistency over regular Release Consistency. The comparisons are done for slightly smaller problem sizes (FFT, 4M points; LU, 1024x1024; Barnes, 16k particles; Ocean, 514x514) than were shown in the baseline result section, and for a slightly less efficient TLB synchronization method (TLB Invalidation routine flushes the entire TLB rather than a single entry).

To measure the performance impact of adding processors that are dedicated to NI message processing, runs were performed with and without extra dedicated NI processors. (All runs in the previous sections were with the extra NI processors.) For the runs without extra NI processors dedicated, all processors that participate in the computation participate in the input processing (there are no extra processors). For the runs with NI processors, there are four additional processors per node taking NI requests, and one processor per node taking hardware interrupts. Table 4 contains the improvement in execution time for four node by four processor runs (16 total application processes), when going from no dedicated NI processors (16 total processors used) to four dedicated NI processors per node (16 application processors, 16 NI processors and 4 HIPPI interrupt processors).

FFT sees only an 7% improvement with additional NI processors present. Inspection of the data shows that it sees a dramatic improvement in perceived (this code is prefetched) read miss latency with dedicated NI processors, but an increased write latency. Since FFT spends so much of its time blocked waiting for data, there is often an idle processor to take the network interrupt, reducing the impact of adding processors to NI processing. LU and Ocean both see larger improvements in execution time when extra dedicated NI processors are present. Barnes sees only a 2% improvement, but inspection of the per-processor data reveals that in both cases, with and without dedicated NI processors, it is operating in the presence of substantial "nacking" and protocol retries because of directory contention. Hence, adding to the latency by removing network resources reduces the contention, mitigating the improvement seen. Overall, having dedicated NI processors improves the performance of the applications, but not proportionally with the number of extra processors used.

Table 4: Impact of Adding Dedicated NI Processors. 16 Process (4 Node by 4 Process) Run

Application	Improvement in Execution Time
FFT	7%
LU	16%
Barnes	2%
Ocean	20%

Table 5 shows the effect of Delayed Invalidated Release Consistency on performance. Again, two sets of runs were done,

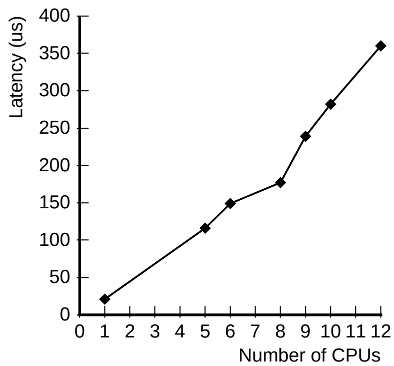


Figure 6: TLB Synchronization Latency as a Function of the Number of Processors Interrupted

both with dedicated NI processors, but one with DIRC turned on. DIRC dramatically improves the performance of FFT because FFT suffers from a substantial amount of invalidate traffic and DIRC postpones and bunches invalidate processing, performing only one TLB synchronization. LU and Ocean exhibit about a 10% improvement in execution time from DIRC. Barnes shows very little improvement. These results overstate the effect of DIRC because the method used to synchronize the TLB is less efficient.

Table 5: Effect of DIRC on Execution Time. 16 Process (4 Node by 4 Process) Run

Application	Improvement in Execution Time
FFT	39%
LU	13%
Barnes	1%
Ocean	10%

4.4 Limiting Factors in Performance

The main limits to better performance are the non-scalability of the TLB synchronization process, the latency of network operations and limited network bandwidth.

TLB Synchronization

The cost of keeping the TLBs synchronized limits performance for configurations with large number of processors per node. Figure 6 shows the measured time to invalidate a single TLB entry held by varying numbers of processors (TLB Synchronization). For large number of processors the time it takes to get through a TLB synchronization operation climbs to above 350µs. The process used to synchronize the TLBs scales poorly. The calling processor interrupts all other processors and waits for them to invalidate the TLB entry and rendezvous[24]. With larger numbers of processors the probability of finding a processor in an interruptible state, and having to wait, increases.

The results shown in this paper run with a modified unix TLB invalidate routine that only interrupts processors that have had DVSM processes scheduled on them. This optimization permits us to run configurations with unused processors without incurring very large synchronization costs. In addition, DIRC is very helpful in postponing and grouping the TLB synchronization operations.

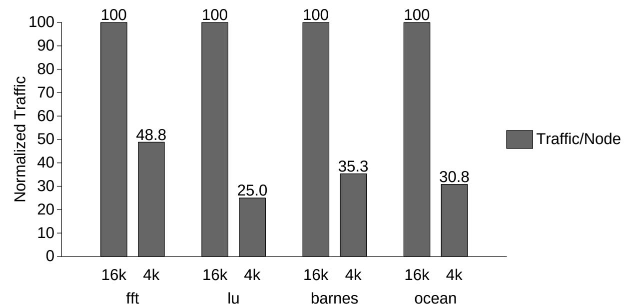


Figure 7: Effect of Large Page Size on Per Node Traffic

To reduce the overhead of synchronizing the TLBs for large configurations, it is necessary to somehow know which processors' TLBs contain mappings to a given page. There are two obvious ways of knowing this. The first way is to support separate per-process page tables, and force each individual process to fault in a page. This approach violates our design goal of keeping sharing within the node as efficient as possible so as to exploit clustering (nevertheless, splitting the page table might provide the best performance).

The second way to track which TLBs contain the mapping is to annotate the TLB refill handler so that each processor shadows its state in main memory. The TLBs are small and very little memory would be required. The shadow information could be accessed during a TLB synchronization operation and only those processors that have the mapping need be interrupted. Although this method will have an adverse affect on TLB refill performance, we think it could be a net gain for MP applications that communicate often but have relatively small datasets.

Latency of Network Operations

The long latency of network operations limits performance on Barnes, FFT and LU. Our current network scheme, integration into the LE layer of HIPPI, leaves much room for improvement through software. We expect that it would be possible to reduce total latencies by a factor of two through tuning. However, such an improvement would not likely improve the performance of FFT substantially unless it also substantially reduced the fixed overhead of sending messages on the network.

Bandwidth Limitations

For highly clustered configurations, with eight processes per node, we are beginning to saturate the network, even though the network used (100 Mbyte/sec HIPPI) is relatively fast by current standards. At runs with greater than eight processes per-node network latencies go up dramatically. For many applications that exhibit all-to-all or all-to-many sharing, the requirement to scale network resources as the node size increases makes it difficult to build very large machines using large clustered multiprocessors. The networks required to attach next-generation machines, with hundreds of processors per machine, might be too costly to be an effective solution to scaling the machine.

Although simulation studies (not presented) show that the miss rates of these applications decrease as the coherency object size is increased from 128 bytes to 16kbytes, the miss rate curve is flat between 4kbytes and 16kbytes. As a result, the use of 16kbyte pages often wastes bandwidth. Figure 7 shows the data traffic per node, when increasing page size from 4kbytes to 16kbytes for a 16 process run (4 node x 4 processes per node). The measurements are made using smaller problems on a MIPS R4400-based system, which permits 4k pages (unlike the R8000 system). A substantial amount of bandwidth is wasted. For some applications, like FFT, the result is dependent on the small problem size. For large enough problem sizes there is no wasted traffic. The HIPPI network and the driver used are tuned for large block transfers. As a result, the latency of operations would not go down proportionally with the block size when moving from 16kbytes to 4kbytes. Nevertheless, the use of 16kbyte pages and an invalidation-based protocol in the prototype system exacerbates the problem of having limited bandwidth for highly clustered runs.

5 Related Work

Our work builds on the previous work of several other researchers. The use of the virtual to physical translation mechanism to enforce coherency was originally suggested by Kai Li[20]. The directory-based invalidation protocol used between the nodes was developed by Kuskin et al[18]. Delayed invalidate release consistency was proposed by Dubois[8] as a way of reducing false sharing, but we use it to limit the overhead of TLB synchronization.

Several other fully operational software shared-memory systems exist, including Munin[3], Treadmarks[17] and MGS[28]. Neither Munin nor Treadmarks have been studied in clustered environments outside of simulation.

MGS by Yeung et al. is the closest to our system. Their proposal for a mixed page- and cache line-based system is similar to ours but their investigation differs in many ways. Because their experimental test bed, the MIT Alewife[1] machine, lacks TLB translation hardware, their experiments simulate the operation of the TLB using compiler generated software translation instructions and software handlers. The issue of TLB synchronization is discussed in the design section, but it is unclear whether it was accurately modeled in the software handlers. The experiments are performed on a partitioned shared-memory machine with DVSM message traffic using the fast shared-memory interconnect versus going over a LAN. Our work differs in that we implement the system using modern components that are matched in performance. Our prototype uses a general purpose HIPPI LAN connected to the I/O interface as opposed to a tightly coupled hardware interconnect. Their remote read miss time for a page of 32,446 processor cycles is nearly an order of magnitude less than ours making comparisons difficult. Most importantly, since they partition a hardware shared-memory machine to build a software shared-memory machine, they scale the internode bandwidth as they increase the cluster size, which we do not.

Clustering effects in hardware cache coherent shared-memory multiprocessors have been investigated in [7] and [9]. Those investigations do not consider page-based systems. Hybrid hardware software page-based systems, similar to what we look at, are studied in simulation by Cox et al[4] and Karlsson et al[14]. Cox looks at different applications then we do, making direct comparisons difficult. Karlsson looks at one of the same applications, FFT and their results are similar to our findings in terms of overall speedup obtained for comparable degrees of clustering.

Hybrid hardware software parallel programming has been studied by Chase using the Amber system[6]. The system does not provide a global coherent address space, instead providing a shared object space that requires explicit annotations. They look at the effects of clustering on the SOR application, but find no difference in performance as the degree of clustering is varied, because of the low communication to computation exhibited by the application at the problem sizes studied.

TLB Synchronization costs in multiprocessor systems have been studied by Rosenburg[22]. He suggests a scheme for reducing the cost of keeping the TLBs synchronized that is similar to the current scheme used in Irix 6.2. Our findings on the scalability of the technique do not agree with his, possibly due to the interference of network interrupts in our system.

6 Conclusions

Extending hardware shared-memory machines using software shared memory is attractive for many reasons. The shared-memory programming model can be retained across a network of more loosely coupled machines. The program can exploit fine grain sharing within the node while permitting the software shared memory to move data in more efficient page sized blocks across the networks.

To investigate the viability of clustered DVSM, we implemented a prototype system that runs on current generation hardware and networks. We find that while clustering is effective in reducing the per-processor internode communication, several factors reduced the impact of improved communication miss rate on overall performance. As the cluster size is increased, the latency of memory operations increases, due to higher TLB synchronization costs in large hardware multiprocessor systems and internode bandwidth limitations.

A kernel implementation of DVSM offers substantial performance benefits over a user implementation, but the latency of memory operations is still not low enough to support communication-intensive applications like FFT. For codes that have low communication to computation ratios, such as Ocean, and which exhibit nearest neighbor sharing, a sharing pattern that is tailored to clustering, the approach appears promising.

The use of large 16kbyte pages and an invalidate-based coherency protocol wastes bandwidth relative to an implementation with smaller coherency objects. Unfortunately for communication-intensive codes like FFT that do not waste bandwidth for large problem sizes, multiple writer protocols will not improve the situation and will increase the latency of memory operations, reducing overall performance.

In addition, we find that protocols such as Delayed Invalidate Release Consistency do more than help false sharing; they can substantially reduce the burden of keeping the virtual address space coherent on a multiprocessor by permitting grouped processing of invalidation messages.

Overall, the approach appears promising if lower latency (possibly memory mapped) networks of sufficient bandwidth are available to connect the large hardware multiprocessors.

7 Acknowledgments

This research was made possible by the generous support of Silicon Graphics Inc. Additional support was provided by DARPA contract DABT63-94-C-0054. Frans Kaashoek and the anonymous reviewers provided many helpful comments on earlier versions of this paper. The authors would also like to thank Mark Heinrich and the rest of the Stanford FLASH hardware development team for making the FLASH protocol available for use.

8 References

[1] Anant Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D Kranz, J. Kubiatowicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance, In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 2-13, June 1995.

[2] Brian Bershad and Matthew J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, Carnegie Mellon University Technical Report No. CMU-CS 91-170, September 1991.

[3] J.B. Carter. Design of the Munin Distributed Shared Memory System, *Journal of Parallel and Distributed Computing*, 29(2):219-27, September 1995.

[4] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus Hardware Shared-memory Implementation: a Case Study, In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 106-17, April 1994.

[5] Rohit Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols, In *Proceedings of International Conference on Supercomputing '94,* pp. 274-288, July 1994.

[6] Jeffery Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors, *In Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pp. 147-158, December 1989.

[7] D. R. Cheriton, H. Goosen and P. Boyle. Multi-level Shared Caching Techniques for Scalability in VMP-MC, In *Proceedings of the 16th International Symposium on Computer Architecture*, pp. 16-24, May 1989.

[8] M. Dubois, J. C. Wang, L. A. Barroso, K. L. Lee, and Y. Chen. Delayed Consistency and its Effect on the Miss Rate of Parallel Programs, *Proceedings of SuperComputing '95,* pp. 197-206, November 1991.

[9] Andrew Erlichson, Basem Nayfeh, Jaswinder P. Singh and Kunle Olukotun. The Benefits of Clustering in Shared Address Space Multiprocessors: An Applications Driven Investigation, *Proceedings of SuperComputing '95,* Dec. 1995.

[10] Ewing Lusk. *Portable Programs for Parallel Processors,* Holt, Rinehart, and Winston, New York, 1987

[11] K. Gharachorloo, Dan Lenoski, James Laudon, P. Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, In *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 15-26, May 1990.

[12] Chris Holt and Jaswinder Pal Singh. Hierarchical N-Body Methods on Shared Address Space Multiprocessors, In *Proceedings of the Seventh SIAM International Conference on Parallel Processing for Scientific Computing*, pp. 313-18, February 1995.

[13] Kirk Johnson, M. F. Kaashoek and D. Wallach. CRL: High-performance All-software Distributed Shared Memory, In *Fifteenth AC Symposium on Operating Systems Principles*, pp. 213-28, December 1995.

[14] Magnus Karlsson and Per Stenstrom. Performance Evaluation of a Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers, In *Proceedings of the Second International Symposium on High-Performance Computer Architecture,* pp. 4-13, February 1996.

[15] Peter Keleher. *Lazy Release Consistency for Distributed Shared Memory*, PhD Thesis, Rice University, Houston, January 1995.

[16] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory, In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 13-21, May 1992.

[17] P. Keleher, Alan Cox, S. Dwarkadas and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, In *Proceedings of USENIX Winter 1994 Conference*, pp. 115-32, January 1994.

[18] Jeff Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, K, Gharachorloo, J. Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum and John Hennessy, The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 18-21, April 1994.

[19] W. Leler. System-level Parallel Programming Based on Linda, In *Proceedings of the Third North American Transputer Users Group*, pp. 175-9, April 1990.

[20] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems*, 7(4):321-359, November 1989.

[21] Ron Minnich. Mether-NFS: A Modified NFS which supports Virtual Shared Memory, In *Proceedings of Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pp. 89-107, September 1993.

[22] Bryan S. Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared- Memory Multiprocessors, *In Proceedings of the Twelfth ACM Symposium on Operating System Principles*, pp. 147-158, December 1989.

[23] Dan Scales and Monica Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines, In *Proceedings of 1st Symposium on Operation Systems Design and Implementation*, pp. 101-14, November 1994.

[24] Michael Y. Thompson, J. M. Barton, T. Jermoluk, and J. Wagner. Translation Lookaside Buffer Synchronization in a Multiprocssor System, In *Proceeding of USENIX Association Winter Conference*, pp. 297-302, February 1988.

[25] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors, In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pp. 219-229, October 1994.

[26] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors, Stanford University Technical Report No. CSL-TR-93-593, December 1993.

[27] Steven Cameron Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations, In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 1995.

[28] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: A Multi-Grain Shared Memory System, In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 44-55, April 1996.