February 1993

Reducing Host Load, Network Load and Latency in a Distributed Shared Memory

Ronald G. Minnich
Supercomputing Research Center

David J. Farber
University of Pennsylvania

Reducing Host Load, Network Load and Latency in a Distributed Shared Memory

Abstract

Mether is a Distributed Shared Memory (DSM) that runs on Sun¹ workstations under the SunOS 4.0 operating system. User programs access the Mether address space in a way indistinguishable from other memory. Mether had a number of performance problems which we had also seen on a distributed shared memory called Memnet[2]. In this paper we discuss changes we made to Mether and protocols we developed to use Mether that minimize host load, network load, and latency. An interesting (and unexpected) result was that for one problem we studied the same "best" protocol for Mether is identical to the "best" protocol for MemNet[6].

The changes to Mether involve exposing an inconsistent store to the application and making access to the consistent and inconsistent versions very convenient; providing both demand-driven and data-driven semantics for updating pages; and allowing the user to specify that only a small subset of a page need be transferred. All of these operations are encoded in a few address bits in the Mether virtual address.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-93-23.

Reducing Host Load, Network Load and Latency In A Distributed Shared Memory

MS-CIS-93-23
DISTRIBUTED SYSTEMS LAB 21

Ronald G. Minnich
David J. Farber

University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

February 1993

Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory

Ronald G. Minnich
Supercomputing Research Center
Bowie, MD

David J. Farber
University of Pennsylvania
Phila., PA

Abstract Mether is a Distributed Shared Memory (DSM) that runs on Sun[1]workstations under the SunOS 4.0 operating system. User programs access the Mether address space in a way indistinguishable from other memory. Mether had a number of performance problems which we had also seen on a distributed shared memory called MemNet[2]. In this paper we discuss changes we made to Mether and protocols we developed to use Mether that minimize host load, network load, and latency. An interesting (and unexpected) result was that for one problem we studied the same "best" protocol for Mether is identical to the "best" protocol for MemNet[6].

The changes to Mether involve exposing an inconsistent store to the application and making access to the consistent and inconsistent versions very convenient; providing both demand-driven and data-driven semantics for updating pages; and allowing the user to specify that only a small subset of a page need be transferred. All of these operations are encoded in a few address bits in the Mether virtual address.

2 Overview of Distributed Shared Memories

Distributed shared memories (DSM) allow processes to share data over a network. They provide a memory model as opposed to a message send/receive model.

In a DSM, programs are provided with a virtual address space. At any given time only a portion of the components of the space are present on the processor the program is running on. When programs access a portion of the space that is not present, the missing portion is fetched. Here is where DSMs differ from conventional shared memory. The data for a conventional shared memory are either copied from disk to main memory, or main memory to cache. In the case of DSM, the data is copied over a network[2]. The latency is therefore much higher. This high latency has many implications for the implementation and use of DSM. It can not be used in the same way that conventional shared memory is used, as the latency can up to 10^4 times higher than a conventional memory bus.

The first description of DSM as we understand the term can be found in [1]. Cheriton describes what he calls problem-oriented shared memory. It is by definition a non-consistent memory. What this means is that different processors may see different values for the same variable at the same time, which is not the case for a consistent multiprocessor shared memory. He argues that trying to achieve consistency in a networking environment is impractical; in this paper he advances the view that the real question is how to properly manage a non-consistent memory.

Two DSMs that provide a consistent shared memory are described in [4] and [2].

In [8], the authors describe what they call a "smart shared memory". This is a memory which understands high-level operators such as queueing operators. A co-processor connected to a high-speed network manages data transport transparently.

These systems vary in terms of how they are implemented and the type of memory they support. They do not vary in that they use the memory model with some modifications as the means by which an application shares data with other applications over a network.

The Mether system, described in [5], constitutes a distributed shared memory. Mether began as a fairly traditional DSM along the lines of [4] and [2]. It has evolved since then, driven by both application demands and performance demands. We will begin this paper with a description of Mether as it now stands; a description of a simple Mether protocol; and a description of an application that uses Mether. We will later in this paper describe the factors that made Mether the way it is.

3 The Current Mether Applications Interface

The Mether system described in [5] was quite similar to systems described in [3], [7] and [2]. Processes map in a set of pages that are distinguished by being sharable over a network. A page could be moved from one processor to another at any time. If a process accessed a page and it was not present on the processor it was fetched over the network.

The goals of the systems were in some cases quite different. The Mach shared memory server supports paging over the Ethernet. Both Mether and MemNet have a different goal: to determine what the next generation of network application interfaces should look like. While it is true that we feel the shared memory model is a part of the picture, we do not believe that

[1] Sun and SunOS are trademarks of Sun Microsystems Inc.
[2] Note that the network is not necessarily a LAN.

it is the complete answer. As networks become faster, the average latency grows, and many shared memory programming techniques will fail in this higher-latency environment. For this reason we are willing to have Mether depart from an emulation of the shared memory model where differences can provide a performance improvement.

As our testing progressed, we realized that the simple pagein-over-network model of the original Mether was not sufficient to meet our needs. In many cases processes need only examine a few variables in a page. Consistent memory is not always needed. Even the demand-driven nature so basic to the pagein-over-network model is not always desirable. We describe these enhancements in further detail below.

Inconsistent Memory

Mether allows a process to access memory as consistent or inconsistent. A process indicates its desired access by mapping the memory read-only or writeable. There is only ever one consistent copy of a page; for reasons described below, we move the consistent copy of a page around, rather than just the write permission to a page.

When a process gets an inconsistent copy of a page, the process holding a consistent copy may continue to write to it. Over time, a read-only copy will become out of date, or inconsistent. There are three ways that an update may occur:

1. A process may request the consistent copy, causing an up-to-date copy of the page to be transmitted over the network, at which time all the Mether servers having a copy of the page will refresh their copy. In this sense the Mether servers are "snoopy".
2. The process holding the consistent copy can cause a new version of the copy to be sent out over the network via a system call.
3. The process holding the inconsistent copy can purge its copy; the next time it accesses the page a new copy will be fetched over the network.

The first two mechanisms constitute a passive update. The last mechanism is an active update. The idea of purging comes directly from the cache operation of the same name.

We have found the inconsistent memory to be useful. While it may seem counter-intuitive for an application to have to deal with inconsistent memory, in fact applications deal with inconsistent memory all the time. Any application that uses windows (either character or bitmapped) manages window descriptors that become inconsistent with the display. At some point the application calls a function that restores the consistency. Applications that look up Internet host names now know that a lookup failure may not indicate that a host name is invalid; it may mean that the global name store is in an inconsistent state and that the lookup should be retried[3]. Programmers who use "out of core" programming techniques on a Cray X-MP[4] or Y-MP also are used to managing the consistency of their in-core data with a backing store (the Solid-State Disk). There are many other examples of applications that manage an inconsistent set of structures and occasionally refresh a backing store. We feel that allowing the application to manage the inconsistent pages in the Mether address space is the correct decision. Mether can not anticipate a program's use of its pages any more than a window manager can decide when to refresh the screen. The application must be given control.

The Cloud System[7] shares some attributes of the current version of Mether, notably the inconsistent page, which they call *weak read-only*. In Cloud, a consistent or inconsistent view is chosen by a system call. In Mether there are a number of address spaces with different semantics, and they are chosen by address bits in the virtual address.

User Driven Page Propagation

Because pages can become out of date, there must be a way to propagate new copies of a page. Since the servers do not always know when propagation should occur, Mether supports user driven propagation. The propagation is supported by two operators in the kernel driver. The first is called PURGE, the second called DO_PURGE. PURGE is used by applications to purge a page; DO_PURGE is used by servers to acknowledge that a PURGE has occurred. For the Ethernet-based implementation of Mether, DO_PURGE is only needed when a writeable page is PURGE'd. PURGE operates differently depending as the page is read-only or writeable.

For read-only pages, PURGE simply unmaps the page from all processes and marks it as invalid. The next time any process attempts to access the page it will be marked wanted and the server must take action to fetch it.

For writeable pages, PURGE will set an attribute for the page called *purge pending*. The process will then go into a kernel sleep until that attribute is cleared. The server, on seeing a page with the *purge pending* attribute set, will broadcast a read-only copy of the page to the network, and then issue a DO_PURGE, which clears the purge pending and wakes up the waiting process.

Short Pages

Another capability added to Mether was support for *short pages*. Short pages are only 32 bytes long. They are actually the first 32 bytes of a full-sized page. A typical use is to store important state variables in the first 32 bytes of the page. The process can access these variables with extremely low overhead and determine whether to access the full (8192 byte) page. The low overhead comes from the fact that page faults cause only 32 bytes as opposed to 8192 bytes to transit the network. The address space for short pages completely overlays the address space for full pages, which is how the short pages can share variables with full pages.

Data Driven Page Faults

An even greater departure from the standard DSM is the support Mether provides for data driven page faults. In the

[3] In fact, it is safe to say that the global name store is almost never in a consistent state
[4] Cray, Cray X-MP, and Cray Y-MP are registered trademarks of Cray Research, Inc.

shared-memory systems described above a page fault always results in a request over the network for a page.

In a data driven page fault, the process blocks as before, but the server does not send out a request. Some other process must actively send out an update in the manner described above. Thus this form of page fault is completely passive. It also results in a very low overhead for a page fault.

The rules for paging pages in and out, mapping them into a process's address space, and locking them into one process's address space are more complex than for other DSMs. A table describing these rules is shown in Figure 1. In this table, *superset* refers to the containing page (i.e. the 8192 byte page in the current implementation); *subset* refers to the contained page (i.e. the short page). Note that we may later have more than two lengths of pages.

Operation	Rule for subsets	Rule for supersets
mapping a page in	All subsets must be present	Supersets need not be present
pagein from the network	All subsets paged in	No supersets paged in
pageout	All subsets paged out	all supersets left paged in but unmapped
lock	all subsets must be present; if all are present, all are locked; otherwise the lock fails and any non-present subsets are marked wanted	No supersets locked but must be present; all are unmapped; supersets not present are marked wanted.
page fault	All subsets must be present	Supersets need not be present
purge	All consistent subsets are purged	Supersets are not affected

Figure 1 The rules for subspace operations

Figure 2 shows how different virtual addresses in the Mether address space reference a single page.

Alternative Applications Interfaces

To the user used to the applications interface provided on, e.g. a Sequent, the Mether interface will seem unnecessarily complicated. On the Sequent or similar machine memory is shared at the byte level. The consistency of shared memory is maintained automatically by the cache hardware. Neither kernel nor user level software need do anything to maintain consistency between caches.

We considered providing such an applications interface, and decided not to. Providing such an applications interface would result in a far more complex and much less reliable implementation of Mether. This decision has a major impact on all the aspects of Mether.

The problems with using such protocols involve the comparatively low reliability of the network we are using; the indeterminacy of the time it takes to purge a cache line; and the inability to order purges.

Notes:

1. The choice of the read-only space or the writeable space is chosen when the application maps the Mether address space in.
2. Note that the consistent space can only be demand-driven.
3. The choice of full or short page, demand or data driven is determined by two address bits in the Mether address space.
4. If further applications demand it, we may opt for four different page sizes- one more bit of address space.

Figure 2 The Mether address space

Most traditional shared cache protocols employ a mechanism for gaining ownership of a cache line. This mechanism is in essence a broadcast message to all other caches to invalidate a copy of a cache line. The algorithms all depend on certain properties of the hardware implementation:

1. The reliability of the cache invalidate message being received and acted on is more reliable than a write to main memory. Thus no explicit acknowledge is needed. The whole set of issues relating to reliable transport are irrelevant to these systems. The transport is by definition reliable and ack-less. A failure of this mechanism is serious enough to warrant a machine check- such a failure is as serious as a main memory failure.
2. The cache lines are small- typically 16 or 32 bytes at most. Even if the cache line needs to be transferred to the requestor's cache, the transfer will occupy only a few cycles.
3. Because no explicit ack is needed for a purge, the cost of invalidating a cache line is the same no matter how many caches have a copy.

Contrast this environment with that seen by a shared memory over an Ethernet or like network using datagrams or a similar connectionless protocol. A host may need to cause all other hosts to purge their copies of a page. It can either broadcast the message or send it to each of the other hosts one at a time. Either way, it must know that all hosts having the page have received the purge command and acted on it. It must therefore know the identity of every host having a copy of the page, as it will have to wait for each host to respond affirmatively. As a result every host must keep track of where every page is. The host must incorporate in the protocol the whole set of mechanisms providing reliable transport, including retries, strategies for determining when a host is down, and so on. A cache purge no longer takes a fixed amount of time; in fact it takes an undeterminable amount of time. Hosts may become unreachable for a period of time and yet still have a copy of the page that must be invalidated. Some strategy for dealing with unreachable hosts would have to be formulated.

Still worse is the problem of ordering of purges. In tightly coupled cache systems the ordering of purges is guaranteed by the hardware: a processor issues a purge and by definition a few clocks later the purge is accomplished and the processor owns the cache line. No purge commands occur in the interval because the processor owns the global cache control bus while it is issuing the purge. Two processors can not issue purges simultaneously: the hardware schedules the purges, usually in a round-robin fashion. Contrast that with the situation on an Ethernet with multiple bridges. There is no ordering of purge requests that can be guaranteed. Two hosts on different trunks can issue purges. Which purge goes out first depends on the depth of the queues in the hosts and the bridges, which in turn depends on background network traffic on each branch. The potential for deadlock, livelock, and deadly embraces is unlimited.

A design which could handle the above situations was examined, and proved to be quite complex. Many of the details of the design were specific to the protocol family we were using (UDP/IP) and the transport medium (Ethernet). Currently Mether communications are managed by a user-level server. Since we plan at some point to migrate the server to the kernel we want to keep it as simple as possible. Burdening the user-level server with a complex protocol would make migration to the kernel impossible.

For these reasons we decided not to use the conventional cache purge mechanism for Mether. In addition, we decided to abandon global consistency as a requirement. That does not mean that access to a consistent view of a page is impossible; rather, it means that we decided to allow the application to decide whether it wanted to pay the price for consistency. There is only ever one consistent copy of a page. If an application needs that copy it pays the price in time for getting it. It will also cause other applications to lose access to the consistent copy for such time as the application needs it. There are, however, many inconsistent copies of a page. These copies represent the state of a page as of the last time it was seen on the network[5]. They may well be up to date, but are not guaranteed to be. The user protocol may decide to access the consistent copy of a page given certain parameters it finds set in the inconsistent copy.

A Sample User Protocol

To give an idea how the different subspaces are used we will give an example of a sample application and a user protocol written to support it. The user protocol we developed used both inconsistent and consistent views of a page as well as demand and data driven views.

The application was a multiple-process sparse matrix solver written by Bob Lucas of the SRC[6]. This program was designed and written for portability and had run on a number of machines, including an Intel[7] iPSC2/VX. The version of the program we have is written in Fortran and runs on the Cray-2. At the heart of his program are send and receive functions modelled after Intel's csend and crecv. To move the program to a new machine requires writing a new version of csend and crecv. The version of csend and crecv we started with transferred data (through the shared memory of the Cray-2) between the processes. There were no shared arrays used.

For this program we wrote a new send and receive function. The functions communicate through two pages. Each process sees a read-only, inconsistent page and a writeable, consistent page, as shown in Figure 1. The WriteGeneration and WriteDataSize in the consistent page are paired with a ReadGeneration and ReadDataSize in the inconsistent page. A write can only proceed when the WriteGeneration in the consistent page and the ReadGeneration in the inconsistent page are equal. A read can proceed only when the WriteGeneration in the inconsistent page is greater than the ReadGeneration in the consistent page. The WriteDataSize is an indicator of how much data to copy out. If the amount of data is less than 32 bytes then the short page can be accessed with a corresponding performance improvement. The amount of data copied (read) is available in the ReadDataSize in the inconsistent page. Because one page on each side is inconsistent, a part of the initialization code purges the current copy of the inconsistent page, so that an up-to-date one will be accessed. This sort of initialization activity is ubiqutous to our protocols; we call it "Deal Me In".

Figure 3 Communications Structures for the Sparse Solver

[5] Because Mether is a broadcast protocol, every time a page transits the network all the inconsistent copies of that page are updated.

[6] The solver was written by Bob before he came to SRC

[7] Intel, iPSC, and iPSC2/VX are registered trademarks of Intel Corp.

The protocol is as follows. The writer locks the page, fills in the data, sets the WriteDataSize, increments the WriteGeneration counter, and issues a purge.

When the reader wants to read the page, it first checks the inconsistent, short, demand-driven copy. If the copy is not present it is fetched. If the WriteGeneration indicates no new data, the reader issues a PURGE on the short page and checks again. If there is still no indication of new data to read, the reader issues a PURGE and then checks the inconsistent, short, data-driven view of the page. At this point the reader blocks until a new version of the page transits the network. When the page comes in, the reader compares the WriteGeneration counter in the read-only page to its ReadGeneration counter. If the WriteGeneration is larger, the reader knows it has some data. It copies the data out, sets the ReadDataSize in its writeable page, increments the ReadGeneration counter, and returns. Note that if the amount of data to be copied out is larger than the short page the reader must access the full-page view. The reader thus views the page in several different ways. The protocol is absolutely symmetric; a write or read from either end proceeds in the exact same way.

Were we less concerned about portability we could make more user of Mether. For example, a data array could be placed in the Mether address space. Nevertheless the program shows linear speedup on up to four processors (all we had available at the time, and the most the Cray-2 version program would run on, as the Cray-2 only has four processors).

We are working on a number of other applications. As the applications we run grow in number, we expect Mether to slowly evolve still further from the classical shared-memory model. We also feel, however, that the major changes have been made.

In the following sections we discuss the factors that affected the Mether applications interface. Changes in the interface were motivated by a simple question: how fast can we transfer a single bit of information- in this case, the change of a counter- over Mether?

4 Application Interface Development

In this section we discuss how and why Mether has evolved to its current design. Many of the changes were driven by testing with a very simple program. The program was effective in that its computation was so simple that its run time was measuring overhead only. Its function is to count up to 1024, cooperatively. We used this program to measure the time it took to change a word and have that change seen at another machine. The program models two processes synchronizing. Because the program does nothing but synchronize, it will exercise the worst-case behavior of all the components of a shared-memory system.

The program constitutes a user protocol. This user protocol will use the Mether applications interface to access pages, modify them, and in some cases purge them.

This program was motivated by our desire to measure both throughput and latency for Mether. In addition we used this program to determine ways to improve the reliability and performance of the user-level servers, the kernel server, and the application interface to the kernel server.

All the trials were run on SUN 3/50s running SunOS 4.0. If both processes were run on the same machine they each took 81 seconds wall-clock time, 37 seconds cpu time. If we simply run the program as one, rather than two, processes, it runs in approximately 50ms[8]. Thus a single processor iteration takes approximately 50 microseconds per increment, including overhead. The two process implementation takes approximately 70 ms per increment, including overhead. The additional overhead is simply the wasted time spent spinning on values that do not change, as there is no (convenient) way to tell the scheduler to go run someone else; it also is the amount of time spent in context switch, which is hard to measure but as a rule of thumb takes a few milliseconds[9]. Given that paging is accomplished in Mether by a user level server, the problem of an Mether client that is spinning on an unvarying memory location should be obvious: the client may be pre-empting the user level server and thus preventing itself from getting the newest version of a page.

This program and subsequent programs embody a user protocol. We had several goals in mind for the tests we performed:

1. To measure and minimize the host load, network load, and latency of the applications interface and server protocols.
2. To find the "best" user protocol for synchronizing over Mether.
3. To determine the relationship (if any) between the "best" protocols for Mether and MemNet.

Depending on the user protocol we observed higher or lower host load, network load, and latency. For this program (protocol) and each subsequent program (protocol) we will describe

1. The protocol
2. The cost of the protocol in space, packets, context switches, and bytes transferred
3. The mean time required for a page fault after 1024 synchronization operations
4. A Loss/Win ratio, that is the number of times the program saw an unchanged variable versus the number of times it saw a changed variable
5. A discussion of the results

First User Protocol: Increment on Full-Size Page

In this protocol, as in all the protocols, the processes increment the first 32-bit word on a full-size page. When we say full-size page we mean that when a process required access to the 32-bit word an entire Sun page (8192 bytes) had to be transferred over the network. For each addition to the word, then, a page had to be transferred and a request had to be processed.

[8] It is not possible to get a more accurate measurement- the resolution of the SunOS process times is 50ms.

[9] A few milliseconds may seem extreme, but 3/50s running SunOS 4.0 are constantly paging

Operation	Cost
Wallclock Time	128 seconds
User Time	10 seconds
Sys Time	30 seconds
Network Load	66 kbytes/second
Context Switches	4 per addition
Average Latency	120 ms
Losses/wins	500

Figure 4 Performance of the first user protocol.

The results of the run are shown in Figure 1.

Discussion It is obvious that most of the program's time is still spent uselessly spinning on unchanging data, as indicated by the Loss/Win ratio. The results of this experiment left us with several areas to explore. We felt the most serious problem was that we were transferring 8 Mb where the actual data that needed to move was 4 Kb. We first addressed network load with an intention to address the host load and latency later. The network load problem was solved by making it possible to have shorter packets.

Second User Protocol: Spin on Short Page

It was obvious that sending 8192 bytes to reflect a change in one word was quite wasteful. We spent some time looking at our applications and decided that some support for short pages would be useful. A short page in this context means that only some subset of the total page is sent when it is requested over the network.

The change to Mether was to partition the address space into two virtual address spaces, and allow the user level server to use the additional address bit as an indicator of how much of a page to fetch. One half of the address space represents full pages; the other half represents the first 32 bytes of each full page. A page fault on the lower half will cause the entire page to be fetched; a page fault on the upper half will cause only the first 32 bytes to be fetched. For the user protocol described above the total data transferred decreases from 8Mb to 146 kbytes[10].

The results are tabularized in Figure 1.

Operation	Cost
Wallclock Time	68 seconds
User Time	3 seconds
Sys Time	17 seconds
Network Load	approx 2.2 kbytes/second
Context Switches	4 per addition
Space	1 page (8192 bytes)
Average Latency	68 ms
Losses/Wins	134

Figure 5 Performance of the second user protocol.

[10] 86kb for data packets, 60 kbytes for request packets

Discussion As we shrank the pages from 8192 to 32 bytes, a ratio of 256 to 1, the user time, system time, wall-clock time, and latency all decreased by a factor of two. At the very least the difference in the ratios suggests we shrank the page too much. Some further calculation indicates that we could make the short pages larger with very little impact on performance; making them smaller would not be worthwhile.

We have significantly decreased network load with this change, but the user cpu time only shrank by one-half. In addition, we obviously wish to improve performance still further. The amount of time spent spinning uselessly increases latency- the time for the other process to access the page- and host load. Our Loss/Win ratio is still quite large. In the next section we examine one of the ways we changed the server and the modified protocol that accompanied the change.

Third User Protocol: Spin on Disjoint Pages, one Read-Only

We had studied this problem on a quite different distributed shared memory (MemNet) and devised a solution which we wished to try on Mether.

In previous examples processes communicated through one page. There is significant overhead in moving the write capability back and forth over the net. The protocol we describe here avoids the problem by leaving the write capability stationary at a given processor, and using pages as one-way links to share information. A process may thus spin on a read-only copy of a page. When a process writes a page it also issues a PURGE, which results in the newest copy of the page being propagated as described in Section 1.

The protocol worked very poorly. Losses outnumbered wins by 10,000 to one. The network was saturated with PURGE packets.

Operation	Cost
Wallclock Time	Never finished
User Time	Never finished
Sys Time	Never finished
Network Load	NA
Context Switches	NA
Average Latency	Very High
Losses/Wins	10,000

Figure 6 Performance of the third user protocol.

Discussion The problem with the protocol on reflection is pretty obvious. It takes no time at all to increment the number; the program spends most of its time finding that its read-only copy is out of date, as it takes tens of milliseconds to get the new copy. However, in addition to doing useless work, the program is generating thousands of useless packets and page faults, all of which must be processed by the user level server, which increases the latency further still. The whole process is degenerative, and in the end it is almost impossible for any work to be done at all.

We tried a number of approaches to fixing the problem. The first atttempt was to add hysteresis, in the form of a fixed delay wait in the protocol after each loss. While this approach improved the win/loss ratio, it was difficult to get consistent timing delays from the SunOS kernel. Also to be considered are the esthetics of such an interface. Asking users to put timing delays in their programs is unacceptable.

Finally, we decided to add hysteresis in a different form. We simply issued a purge after every 100 losses. This worked acceptably well: the program would at least run, but not quickly. Increasing the number to 10,000 decreased the user time to 19 seconds on average, but the system time was 50 seconds. In addition, the ratio of losses to wins was still 80 to one. The results for the protocol with hysteresis are shown in Figure 1. While we were able to improve the performance with this protocol after hysteresis was added, it was clear that we still were not providing the right sort of interface. CPU time was still to high.

Operation	Cost
Wallclock Time	77 seconds
User Time	19 seconds
Sys Time	50 seconds
Network Load	approx 1 kbytes/second
Context Switches	5 per addition
Space	2 pages (16384 bytes)
Average Latency	45 ms
Losses/Wins	80

Figure 7 Performance of the third user protocol with hysteresis.

It was clear that we needed to re-examine some basic assumptions. It occurred to us that to this point the design had always been demand driven. In fact, most distributed shared memories are. When a process faulted, a page request was sent out. We wondered what the effect of putting data driven semantics into our model would be. In a data driven model a process could fault on a page but no active request would be sent out; rather, the server would wait for a copy of that page being broadcast to the network.

To implement the data driven semantics of pages we partitioned the virtual address space one more time. Before we had long and short pages; now we had long and short pages, data and demand driven. There are four views of a page. The data driven view is by definition read-only and therefore inconsistent.

Fourth User Protocol: Spin on Short Page, Data Driven

In this user protocol we access two different views of the page: the demand-driven, consistent, writeable, short view and the data-driven, inconsistent, short, read-only view.

Performance did improve marginally. The win/loss ratio was disappointing, 400 losses for every win. User time increased to 7 seconds on average.

Operation	Cost
Wallclock Time	68 seconds
User Time	7 seconds
Sys Time	50 seconds
Network Load	approx. 1 kbytes/second
Context Switches	10 per addition
Space	1 page (8192 bytes)
Average Latency	65 ms
Loss/Win	400

Figure 8 Performance of the fourth user protocol.

Discussion The goal we set for this system was not met. We had hoped that the process which wrote the counter and then issued a purge would spend a lot of time waiting, however the win/loss ratio indicated that this was clearly not happening. It is easy to see why: the purge returns very quickly, and the process continues to sample a value that is not changing. The process in addition blocks the user level server. This is a testimony of sorts to the efficiency of short pages and the implementation of the user level server.

The experience with this protocol led quite quickly to the final user protocol design. It was clear that a process could block on a page, as long as it did not have a writeable (e.g. consistent) copy of that page. Once again we used two pages, maintaining a writeable, consistent, short, demand-paged copy and a read-only, inconsistent, short, data-driven copy on each machine.

Final User Protocol: Spin on Disjoint Pages, one Data Driven

In this user protocol we maintain two pages. Each program maintains a consistent, writeable, demand-driven, short page. The other sides views that page as an inconsistent, data-driven, read-only, short page.

The results were extremely good. User time dropped to below one second. System time was consistently under 8 seconds. Wall-clock time was on average 57 seconds. Only one packet was ever sent per increment: the PURGE packet from the host with the writeable page. The network load thus dropped by a factor of almost two, as no request packets needed to be sent.

Operation	Cost
Wallclock Time	57 seconds
User Time	.7 seconds
Sys Time	6 seconds
Network Load	.5 kbytes/second
Context Switches	5 per addition
Space	2 pages (16384 bytes)
Average Latency	20 ms
Losses/Wins	3

Figure 9 Performance of the final user protocol.

Discussion In any event, this protocol is ideal in almost every sense, save that it takes two pages instead of one. There is one packet per addition; the loss/win ratio is very low, indicating that a process is either incrementing the variable, checking the variable once or twice, or sleeping on a new version of the variable. The user level server is doing less work as well, which also decreases the total host load. The latency is as low as we are going to get given that we have a user-level transport as opposed to a kernel-level transport- it is close to a theoretical minimum derived from what we know of SunOS 4.0. At this point we have hit a threshhold in which the major bottleneck is now the context switches required to receive a new page. That problem will be solved by a different hardware-based network or a migration of the user level server code to the kernel.

5 Conclusions

Using the information gained from these tests, we built a library which provides support for using Mether efficiently. The library provides named segments with capabilities; pipe-like operations; and other operations to make use of Mether convenient for programmers.

Some applications use shared memory to pass small blocks of data between processes. We earlier gave one example: a program which ran on a Cray-2 and used shared memory, but only for (user-written) send and receive primitives. For those applications which wished to use Mether as a message-passing medium, we support a set of functions that establish pipe-like semantics. One may create a pipe or open an existing pipe. In either case, two pointers are returned, a read and a write pointer. These pointers may be used to read the pipe and write the pipe, using the pointer pair. A bidirectional flow of data is possible. Note that this message-like support is provided for those who use it, but it is not the only way to use Mether. The programming effort of emulating send/receive on Mether is much less than writing send/receive primitives that use, e.g., SunRPC.

6 Summary

Mether is an implementation of DSM over Ethernet. Mether exposes the cost of accessing a consistent page to the application, rather than trying to provide the application with a consistent view of all pages. Applications may access a not-necessarily up-to-date copy of a page at much lower cost in time. Applications may also access subsets of a page. The encoding of how an application accesses a page is performed in the virtual address. Thus an application can reference page subsets or inconsistent copies of the page at will, without the overhead of a system call to change its access mode.

Exposing an inconsistent store to the applications programmer may at first seem non-intuitive. In fact, programmers have been dealing with inconsistency for years. We feel that exposing the inconsistency of Mether to the application allows the best decisions to be made about when consistency should be provided.

We have experimentally determined that the best compromise for decreasing network load, host load, and latency comes at a cost in the number of pages a program uses to communicate with other programs. The experimental results for Mether directly match the analytical and simulation results for Memnet, a distributed shared memory implemented completely in hardware. Finding the identical "best" protocol for Mether, a software DSM, and Memnet, a hardware DSM, is surprising.

In using Mether we have found the programming interface much easier to deal with than, e.g., communicating via TCP/IP or the various RPC libraries available. We feel that the distributed shared memory model is effective for the next generation of high-speed networks.

Acknowledgements

Ron Minnich wishes to thank Maya Gokhale for reviewing early drafts of this paper.

Bibliography

[1] David R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed systems design. In Proceedings of the Sixth IEEE Distributed Computing Systems Conference, pages 190–197, 1986.

[2] Gary S. Delp, Adarshpal S. Sethi, and David J. Farber. An analysis of memnet: An experiment in high-speed memory-mapped local network interfaces. Udel-EE Technical Report 87-04-2, Department of Electrical Engineering, University of Delaware, Newark, Delaware 19716, April 1987.

[3] A. Forin, J. Barrera, and R. Sanzi. The shared memory server. In Usenix- Winter 89, pages pp. 229–243. Usenix, February 1980.

[4] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In Proceedings of the Sixth Symposium on the Principles of Distributed Computing, pages 229–239, 1986.

[5] Ron Minnich and Dave Farber. The mether system: A distributed shared memory for sunos 4.0. In Usenix- Summer 89. Usenix, 1989.

[6] Ronald G. Minnich. Protocols for a high speed network-seminar. Technical report, University of Delaware, Department of Electrical Engineering, 1987.

[7] U. Ramachandran and M. Y. A. Khalidi. An implementation of distributed shared memory. Technical report, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, December 1988.

[8] U. Ramachandran, M. Solomon, and M. Vernon. Hardware support for interprocess communication. In Proceedings of the IEEE Annual Symposium on Computer Architecture, pages 178–188, 1987.