Harnessing The Power of Fast, Low Latency, Networks for Software DSMs

Ayal Itzkovitz*
Department of Computer Science, CIMS
New York University
ayali@cs.nyu.edu

Assaf Schuster Yoram Talmor
Computer Science Department
Technion – Israel Institute of Technology
{assaf,ytalmor}@cs.technion.ac.il

Abstract

Asynchronous communication using fast, low latency networks present the challenge of delivering the performance such networks exhibit, from the network layer up to user level applications. Such applications which need to retrieve messages from the network, using polling based communication packages, often face the problems of responsiveness, CPU utilization and redundant polling. Because of the inherent asynchronous communication nature of software DSMs, these overheads become apparent when the DSM is implemented on top of such communication packages.

This paper presents two critical issues in the design and implementation of software DSM in the presence of fast, low latency networks. First, it describes the implementation of a novel technique, called MultiView, which provides small-size pages. MultiView is used for tailoring the basic sharing units that are used by the DSM to the native application data. This results in false sharing avoidance, smaller message sizes and helps to prevent excessive buffer copying. Second, it proposes a solution for customizing network drivers to efficiently support asynchronous communication, which is best experienced in DSMs. Performance evaluation shows that our methods improve the responsiveness, CPU utilization and the overall DSM performance.

1 Introduction

Distributed Shared Memory systems (DSMs) were introduced over a decade ago, proposing an abstraction of shared virtual address space for distributed processes. Since then, several assumptions on the underlying environment have changed. One example is the use of multithreading in DSMs, built on top of mainstream operating systems such as Windows-NT and Solaris [18, 12, 19], and its inclusion as a standard in modern programming languages such as Java. These made multithreading and shared memory programming attractive with a rapidly increasing volume of applications.

*Part of this work was done while the author was with the Computer Science Department at the Technion, Israel.

The efficiency of the underlying networking layer is a crucial factor in the performance of DSM systems. Thus, the recent advent of fast of-the-shelf networks such as Digital MemoryChannel [7], ServerNet [9], Myrinet [2] and others, seem to open new horizons. The low latencies and high bandwidth obtained through these networks may affect the DSM system design in several ways. New issues arise, such as buffer copying, message size, and asynchronous communication. For instance, a lot of the past research in DSM systems focused on reducing the number of messages while compromising their size; in contrast, when using fast networks, message size has a major influence on the latency, while the overhead per message is relatively small.[1]

In this paper we describe issues in the design of a high-performance DSM system, called Millipage. Millipage was designed and is running on a cluster of Windows-NT machines, interconnected with a low latency fast Myrinet switch [2]. Millipage implements the sequential consistency memory model. It is the first DSM system which presents to the application a strictly consistent memory behavior, and whose performance compares to that of relaxed consistency systems. This is achieved through the following unique aspects.

First, Millipage is the first page-based DSM system which supports sharing in relatively small-size, application-defined units, called *minipages*. Minipages are application-customized, storing data objects and program variables of any size, manipulating them independently and efficiently. This is done through a recently published technique, called MultiView, which uses a novel memory layout and requires no special compiler support [11]. As explained later in Section 2, MultiView uses mapping capabilities which are already available as operating system APIs. Thus, Millipage is portable and is efficiently implemented.

Second, Millipage adapts the networking software layer for *asynchronous communication*. Fast messaging layers such as ActiveMessages [21] and FastMessages [14] require *polling* for fetching a received message, which results in a "polling tradeoff" (lower responsiveness of server threads vs. wasted CPU cycles for polling). In systems which are multithreaded and employ asynchronous communication (a remote request may be issued at unexpected, arbitrary times), this tradeoff substantially degrades the overall performance. Clearly, the polling based policy is inadequate, and must be changed to an event-triggered communication scheme. Because of this observation, the messaging layers of Millipage were care-

[1]In our environment we found that (from a certain size, which is about a hundred bytes) the messages latency increases linearly with the message size

fully customized to meet the demands of multithreaded, asynchronous-communication based applications.

Finally, Millipage ensures that no extra buffer copying occurs when handling remote requests. Modern messaging layers may copy-in/copy-out directly from/to the network interface card to/from the user space. Millipage uses the capabilities of MultiView through a special *privileged* mapping to allow these DMA operations with no additional buffer copying. This also lets the DSM copy-in/copy-out minipages in a multithreaded environment with no need for "suspend-all" algorithms.

For completeness, we mention here that the programmer in Millipage allocates shared memory through a `malloc`-like API. Each such call returns a pointer to a variable which is stored in a minipage, and for which Millipage ensures strictly consistent values.

Millipage was completely designed and built in Windows-NT. In this paper we describe how Millipage employs the Windows-NT capabilities to provide its own unique features: The small size minipages which resolve false sharing and reduce the message sizes, and the customization of the fast messaging layer for improved responsiveness in the presence of multithreading and asynchronous communication. We show how the implementation in Windows-NT results in an efficient, high-performance DSM implementation where false sharing is resolved, message sizes are small, responsiveness is high, buffering is avoided, and short latencies are experienced.

The rest of this paper is organized as follows. In Section 2 we describe how the memory mapped I/O mechanism of Windows-NT is used to create minipages. Section 3 describes our work in customizing the FastMessages drivers, in order to increase the responsiveness of the DSM system. We provide related work in Section 4. We give our conclusions in Section 5.

2 Using Memory Mapped I/O For Implementing Minipages

Among software DSM techniques, page-based DSM systems are the most common; they are easy to develop and portable across architectures. However, due to their use of the OS protection mechanisms, they are restricted in the size of the sharing unit which matches an OS page. Multithreaded computation requires atomic page updates, which further complicate the implementation of DSM systems; the page protection change should precede the modification of the page contents, while simultaneous access of other threads to the page must be prohibited.

Those issues can be addressed in a relatively simple mechanism, called MultiView, that enables fine-grain sharing in page-based DSMs. MultiView was first introduced in [11]. We now describe it briefly and elaborate on the implementation and limitations in Windows-NT.

2.1 MultiView

Consider three variables which reside in the same page. Using three different mappings of the same page to the virtual address space of a process, three *views* of this page can be constructed. Once the views are constructed, memory protection can be assigned to them independently of each other. Now let the application access each variable via a dedicated view, which makes the variables accessible with different, independently assigned permissions, despite the fact that they reside in the same page. Consequently, false sharing is resolved, as one variable may be accessible on one process while another variable (in the same page) is exclusively accessible by a different (remote) process.

We define the portion of the page where a variable resides (and is being accessed exclusively through one of the views) as a *minipage*. Thus, to the OS, the minipage is the same as a page, but is smaller in size. It can be set with any protection value applied for a page, and can be considered remote or local, replicated or not.

Now consider the problem of atomic update of minipages. When a page arrives at a node in which all threads have no access privileges, all that is needed is to write the minipage contents in the appropriate place in memory. To this end, the updating thread should be assigned write privilges, while all other (application) threads should have no access permissions. This is achieved by constructing an additional, *privileged* view in which access is fixed to `ReadWrite`. The privileged view is accessible to the DSM system threads at all times, but is never used by the application threads.

The views mechanism, as described above, can be further extended by larger minipages which contain sets of smaller ones [10]. In this way the DSM may switch aggregation levels and aggregation "set covers", thus enhancing the functionality and richness of the DSM system.

2.2 MultiView Implementation

MultiView requires multiple mapping of memory pages to the process' address space, applying a different protection value to each of them. Indeed, Windows-NT provides a well-documented mechanism called *Memory Mapped I/O* that is described below.

Memory Mapped I/O lets a process map a certain file (or a portion of a file) to a contiguous region in its address space. Once mapped, reading and writing to this memory region is interpreted by the operating system as reading and writing to the file itself. Updates to the file are not guaranteed to be committed immediately, as portions of the file may be cached in memory. Furthermore, a second process which opens this file is not guaranteed to get the most recent modifications by the first process, unless the file is closed by the first process.

Memory Mapped I/O may be very useful for simplifying the access to large files. However, it can also be used as an Inter Process Communication (IPC) mechanism for sharing memory between two processes *residing on the same machine*. Setting a special parameter (-1) as the file name indicates a section object which is allocated from memory and backuped up by the swap file. Mapping this section object by two or more processes to their respective address spaces makes this section object a shared space. As documented, Windows-NT guarantees strict consistency for local views of the same section object. This is implemented simply by mutual translations in the page tables of the processes, so that the corresponding virtual pages are mapped to the same physical page [17].

Millipage uses this mechanism, so that instead of mapping the same section object by different processes, Millipage remaps the same section object several times *by the same process*. This lets us implement multiple views of the same page in the address space of a process.

The implementation consists of two steps. First, a section object of a certain size should be defined. This informs the operating system of the space which must be reserved in the swap file for this section object. A call to `CreateFileMapping` returns a handle for the section object.

In the second step, this section object is mapped to the virtual address space of the process by using `MapViewOfFile`. Alternatively, if a specific address is to be specified for the mapping then `MapViewOfFileEx` may be used.

The ideal implementation of minipages would then be as follows. For each allocation of a variable (e.g., in each "malloc"), create a section object of the size of the variable and map it following the last mapped section object. In this way variables and their mapped views appear in the same order, which would preserve the spatial locality when the variables are accessed via their mapped views. However, in order to have independent protection setting, the size of a section object can not occupy less than a page. If section objects could be declared in granularity of a single OS page then spatial locality could still be preserved, as is depicted in Figure 1(a). However, Windows-NT restricts the declaration of section objects to be a multiplicity of 64KB. Therefore, we had to employ a different implementation of minipages, as follows.

At initialization time, a single large section object is created. Several views are constructed to this section object through multiple mappings. The number of views constructed equal to the maximal number of minipages which may reside in the same page. Figure 1(b) depicts this implementation. Notice that with this setup spatial locality is not preserved: referencing two variables which reside consecutively in the section object will be done via two addresses in the virtual address space which might be quite distant from each other.

2.3 Overheads of the MultiView Implementation

In order to validate the efficiency of our NT implementation of MultiView, we wrote a simple test application which traverses (in a loop) a large array of integers, all packed in minipages. The array is accessed via (equally loaded) views that are constructed, so that the number of views dictates the minipage size. As expected, the total size of committed memory increases with the size of the allocated region, independent of the number of views. We were limited, however, by the size of the available virtual address space in NT, which stands at about 1.63GB.

As can be seen from the graphs in Figure 2, when using less than 32 views we experience an overhead of less that 4%, which may be attributed to the increase in the TLB misses. At certain breaking points the overhead becomes substantial and leads to high slowdowns. These appear precisely when the L2 cache can no longer hold all of the PTEs (128K minipages).

More performance measures which evaluate the Millipage DSM system can be found in [11] and thus will not be repeated here.

3 Customizing Fast Communication to Asynchronous Communication

In this section we describe our experience in adapting the fast network interfaces to asynchronous communication. Generally, this involved switching from a polling based policy to an event driven scheme.

3.1 FastMessages

Millipage uses FastMessages (FM) on Myrinet as its communication layer. FM was developed at the University of Illinois as a low latency messaging layer, working on fast networking media such as Myrinet. We measured a roundtrip delay of 25 μs for small messages (200 bytes) and 180 μs for 4 KB messages. FM achieves network bandwidth higher than 1 GB/sec on our switched Myrinet LAN.

FM provides a reliable and FIFO ordered messaging service. Its high performance is due to two main features. First, it does not switch between user mode and kernel mode, but rather transfers data directly to and from user mode. Second, it minimizes buffer copying of messages. When a send operation is initiated by the user process, FM verifies that there is sufficient space in the network buffers at the target network interface card (NIC). Then the message is read directly from the user space, through the local NIC to the target NIC. Using DMA, the message is copied at the receiver side from the buffers of the network card to the FM reserved (and pinned) memory.

3.2 Asynchronous Communication

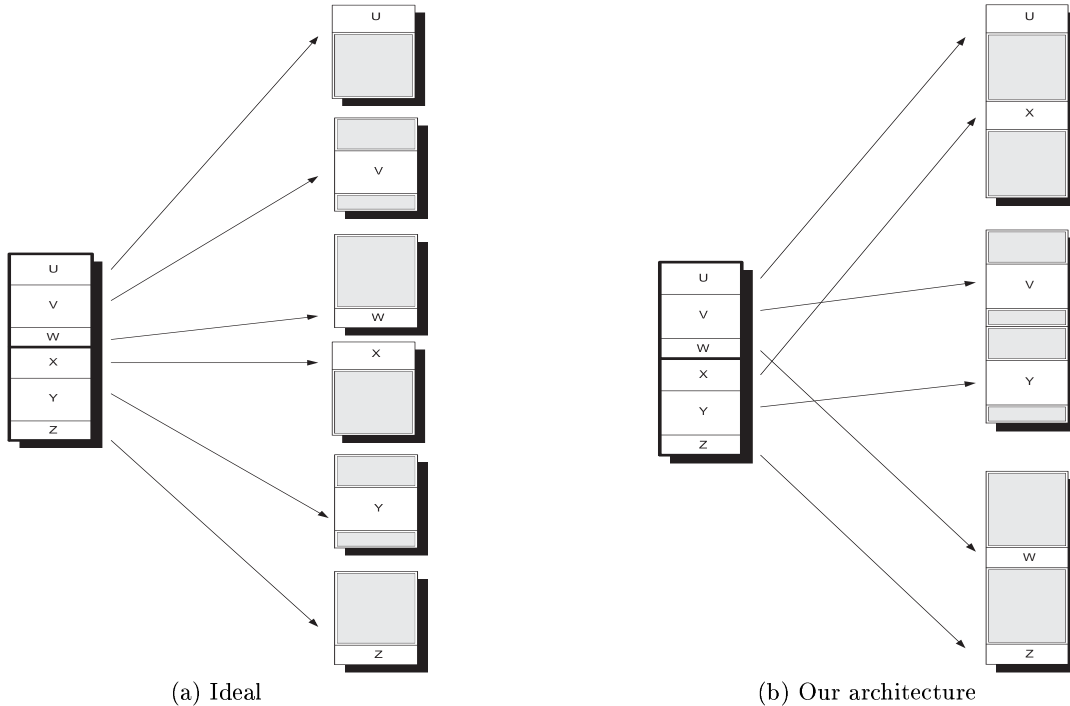
The user level application which uses Fast Messages library has to poll in order to check if a message arrived and can be processed by its handler. Therefore, every extraction of a message requires executing a loop which calls `FM_EXTRACT` (Fast Messages main message extraction function) until it succeeds. This scheme is acceptable for synchronous communication, where the receiver knows when to expect an incoming message and can optimize message extraction accordingly. However, this scheme is not acceptable when using multithreaded asynchronous communications applications. The main problems one encounters when using polling in asynchronous communication include:

Responsiveness: Since the application is multithreaded, the thread which waits for the message using polling, might be swapped out of the CPU when a message arrives. This might have double impact; firstly, the local message retrieval is delayed until the thread is swapped-in back to the CPU, when it can extract the message. Secondly, the remote thread which sent the message might be waiting for the local thread response. The effect of such a delay does not lead to livelock or deadlock, but it might reduce the overall performance considerably.

Wasted CPU Time: Each time a thread executes a polling loop, and there is no message waiting to be extracted from the network, the application actually wastes CPU time, which could be used by another thread not waiting for a message. The wasted CPU time is more apparent when there is more than one thread which is not waiting for incoming message (and therefore can utilize the CPU for computation).

Unnecessary Thread Context Switches Since we don't know when to expect an incoming message, the thread that waits for the message needs to poll the network "often enough" in order to achieve reasonable responsiveness. On the other hand, this thread cannot occupy the CPU for too long, as other threads are starved. Therefore, polling leads to unnecessary thread context switches.

As described in [3], simple interrupt-driven scenario might be insufficient under certain conditions (mainly when the network is fast and the CPU is network bound - as is the case for ATM). However, this is not the case here. As more functions are performed by the network (especially by the



(a) Ideal

(b) Our architecture

Figure 1: **(a)** Ideal multiple mapping which preserves the spatial locality: variables that reside consecutively in the section object are referenced via consecutive virtual addresses. **(b)** The mapping when the section object is forced to be declared in chunks of two OS pages (NT restricts the declaration of section objects to multiplicity of 64KB). Notice how the spatial locality from **(a)** is destroyed.

on card CPU and the card DMA engines), CPU intervention is minimal. In addition, most messages sent by Millipage are small and require only few CPU cycles to process. This led us to construct the mechanism described in the next section.

3.3 Interrupt-Enabled Fast Messages

The overall purpose of the changes we made to the Fast Messages Library and the device driver was to deliver, as fast as possible and with minimal CPU intervention, the notification of incoming messages to the user level thread. Changes made to the original FM include the `fmmcp` (FM Control Program, the program that controls the execution of the network card's CPU), the device driver, and small changes made to `fmcm` (FM Context Manager) and `myrilib` (one of FM lib files). These changes (described in short below) were necessary mainly because the current version of FM supports only polling based message extraction. We came up with two possible solutions, both depending on Windows NT's interrupt mechanism:

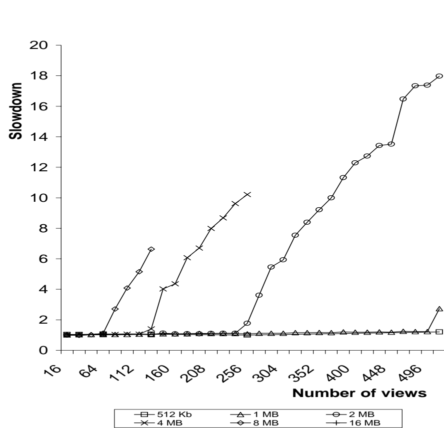
Thread Priority Boost. Since boosting the priority of a user level thread, from kernel mode, cannot be done using documented APIs [3], we had to use undocumented kernel-mode APIs. Basically, we used the user level thread handle to extract kernel mode data structures, which allowed us to access the thread's priority data, and change it from within the device driver's ISR (Interrupt Service Routine). We used a specific thread as the communication thread, running at low priority. Once an interrupt occurs, the device driver increases the thread's priority, allowing it to extract messages. When the communication thread finishes extracting messages (i.e., FM_EXTRACT returns no more information) the thread reduces its own priority back.

Synchronization Event. In this solution, we used a synchronization event object (created by a call to `IoCreateSynchronizationEvent`). This event object is created under `BaseOfNamedObject` directory, were it can be accessed by user level threads as well as the device driver. The communication thread is created with high level priority. Once created, the thread registers on the synchronization object, using `WaitForSingleObject`. Every time an interrupt occurs, the device driver signals the event, thus releasing the communication thread. Since the thread executes at high priority it immediately begins to extract messages. When all the messages are extracted, the thread returns to wait on the event object.

These solutions required the following main changes to the original FM communication package:

Device Driver. The changes to the device driver were required simply to allow it to register an interrupt object, so that interrupts can be used. It also creates/opens the event used to synchronize the communication thread. We kept the ISR as simple as possible in both solutions; its only roles are to boost the priority of the communication thread in the first solution, or to signal the event in the second solution.

Fmmcp. We changed the fmmcp, so that the network card generates interrupts whenever a packet is DMAed into the host memory. Importantly, the Fmmcp does not



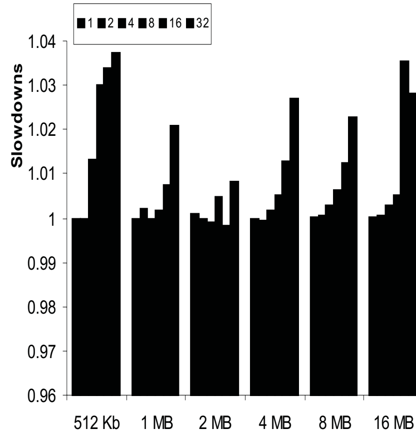


Figure 2: Overheads of MultiView. Note the efficiency of the implementation before the breaking-points in the graphs. Overheads there are due to TLB misses (no apparent OS-related overhead). The breaking points appear precisely when the PTEs can no longer be cached in L2.

wait for the device driver to acknowledge the interrupt, but rather continues its execution. This actually breaks the coupling between hardware generated interrupts and the event object signaling at the user level, using the following rational.

The goal here was to save on redundant interrupts and event object signaling. The basic observation is a potential overlap in time between message extraction in user level and interrupts generated by the network card. This overlap implies that signaling the event object is not needed while the user thread keeps extracting messages. Since the ISR and the event signal are much faster than the actual message extraction, disabling the interrupts mechanism while messages are being extracted does not affect the correctness of the message retrieval algorithm. By following this approach we save on card CPU time, since the fmmcp continues executing and can DMA more packets from/to the network, while previous ones are handled. In addition, the device driver handles less interrupts, issues less event signals (or thread priority boosts, which affects the user thread currently extracting messages). Thus, the number of messages our mechanism can handle over a short period of time increases.

3.4 Performance Evaluation: Polling Vs. Interrupts

Our measurements were carried on a cluster of 8 Compaq Pentium II 300Mhz uniprocessor machines, running Windows-NT Workstation 4.0 SP3, with 128 MB of RAM and 512KB L2 Cache. We ran the experiments under administrator account and increased the maximum working set to 128MB (so the operating system does not initiate trimming).

In order to check the responsiveness of the user level communication thread, we used a simple application which creates an array of DSM minipages in an arbitrary node in our cluster. Then, using Millipage architecture, another node reads the data stored by the minipages. The measurements results, which can be seen in Figure 3 (left), show the average time it takes for the remote thread to read minipages of various sizes. There appears to be no difference between the previous polling mechanism and our new mechanism.

The polling mechanism works as follows: it uses a high priority thread and a low priority thread to poll for incoming messages. The high priority thread uses a fine granularity timer (1ms timer) to synchronize between two consecutive polls. When no other threads occupy the CPU (as is the case in Figure 3 (left)) then the low priority thread is free to poll the network continuously. This implies that our interrupt based scheme is as efficient as polling wrt raw responsiveness of the DSM server threads.

We ran the same application again, only this time we added a background thread in the replying node, that continuously performed arbitrary calculations. The results can be seen in Figure 3 (right). Since the low priority thread of the polling mechanism barely receives CPU time, most of the incoming message handling is done by the high priority polling thread. Since this thread is running only at certain intervals (according to the 1ms timer), with no indication of the arrival of an incoming message, the responsiveness degrades significantly. On the other hand, the new mechanism suffered only a slight reduction in responsiveness (mainly since the background thread also used the CPU). We performed the same measurements for remote writes and got similar results.

Using the same application, we found that the polling version communication thread used about 40% more CPU time than the interrupt based communication thread. Somewhat surprisingly, another set of measurements shows that it takes about $1\mu s$ from the moment the interrupt is generated by the network card, until the signal that is created by the device driver ISR is received by the user level thread. Others report on a $10\mu s$ delay, however using different hardware [1].

3.5 Millipage Performance

In order to evaluate the overall performance we ran several benchmarks using two configurations: one with the previous version of Fast Messages (i.e., the network is polled for new messages), and the other one using our new version of Fast Messages (based on interrupts). The results are summarized in Table 1.

As can be seen, although being a relatively small change

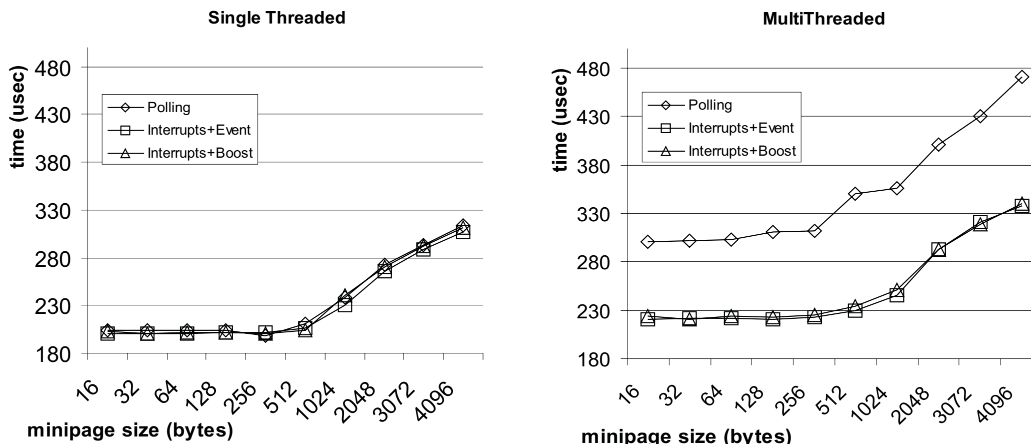


Figure 3: Polling vs. interrupts. **(Left):** when no application thread occupies the CPU then a relatively smart polling scheme is as responsive as the event-triggered schemes. **(Right):** when application threads are active then responsiveness substantially degrades in the polling scheme, while the event-triggered schemes are not affected.

Application	Data Set	Time			Improv.
		Seq.	8 node Polling	8 node Interrupts	
SOR	1024x1024 matrices	7.1	0.99	0.99	0%
LU	1024x1024 matrix, block size 32x32	26.0	6.0	5.72	4.7%
IS	2^{23} numbers, 2^9 values	20.6	2.83	2.69	4.9%
WATER	343 molecules	15.3	2.68	2.29	14.6%
TSP	19 cities, recursion level 12	160.0	34.56	28.57	17.3%

Table 1: Execution times as measured on five benchmark applications using a polling policy and an interrupt based communication scheme.

in the overall execution of the DSM, changing the underlying communication layer behavior from polling-based policy to an interrupt-based scheme results in an overall performance improvement of up to more than 17%. In the benchmark applications we checked, which already exhibit good speedups by using MultiView, the improvement percentage varies a lot. In general, we can see that the more asynchronous the communication pattern of the application is, the more it suffers from poor responsiveness, thus it benefits the most from using the interrupt-based scheme. Applications like SOR, IS and LU, which synchronize mainly via barriers, have a much more synchronous communication pattern, therefore benefit up to 5% in their execution time. However, TSP and WATER, which are lock-synchronized, have a benefit of 14%-17% when improving the underlying communication library behavior. We can also see that no application suffered from any performance degradation when using the interrupt-based communication library.

4 Related Work

A few DSM projects have been conducted on NT. Brazos [18] uses multithreading in NT in order to hide latencies. SVMLib [13] provides a software layer that implements the Win32 API interface on top of Unix, thus is able to develop multi-platform (Unix and NT) distributed shared memory system. Malaxis [4] provides a DLL that supports the DSM functionality, to be linked with MFC and other high-level applications. None of these systems provide fine-grain sharing or use fast networks to provide high performance.

Blizzard [16] and Shasta [15] provided fine-grain access control through code instrumentation, thus bypassing the page protection mechanism. These systems are non portable and involve high overheads. In contrast, our work is solely based on the standard OS mapping and protection mechanisms, thus efficiently providing fine-grain sharing with no need for a special compiler support.

There are several user-level network interfaces developed for Myrinet network. All of them support the use of polling for message reception, and some support interrupts. Active Messages [21] supports interrupt notification of message arrival. In U-NET [20], applications detect message arrival using polling, blocking or asynchronous notification. VIA [6, 22] combined the functionality of previous network interface protocols, such as U-NET and VMMC [5] and it is emerging as the new standard.

[8] experimented with Windows-NT and suggested using polling during high network load. In the domain of DSM, we found polling to be inadequate, while interrupt-based schemes perform well under different loads and actually outperform polling schemes.

Comparison between interrupts and polling was done in [23]. According to this work, tradeoffs between polling and interrupts aren't clear. Passing the interrupt notification user level threads is relatively expensive operation in Unix (costs about 70us). We, on the other hand, use NT's event

model, which allows sharing of user-mode and kernel-mode event notification, therefore reduce the overhead of using interrupts.

5 Conclusions

Our Windows-NT implementation of a software DSM system provides a reach volume of experiences with this operating system. The focus of this paper was the adaptation of the Millipage DSM system to work on top of fast messaging hardware and software interfaces. To this end, we focused on three main issues: Fine grain sharing which avoids false sharing and helps to reduce message size, event-driven network interfaces which enhance the responsiveness in the presence of multithreading and asynchronous communication, and the avoidance of buffer copying.

Application-tailored fine-grain sharing is supported using the very recent MultiView method which employs unique memory mapping. MultiView is also used to provide to the memory, avoiding extra buffer copying. In the presence of multithreading, MultiView helps to escape from "suspend-all" algorithms during copy-in/copy-out operations. We gave details of the NT implementation (the only one existing to-date) of MultiView which is done through the efficient mechanisms of Memory Mapped I/O. We have shown that beyond the hardware constraints and wrt the OS parameters this implementation is very efficient. On the other hand, we described some OS-imposed restrictions which degrade the locality.

Asynchronous communication is heavily used in DSMs. In the presence of multithreading, responsiveness degrades due to the polling tradeoff. We described our experience in removing this obstacle by designing an event-triggered mechanism which uses NT features. This mechanism exploits NT's event model, which allows direct signaling of user threads from kernel mode. Our results prove that this mechanism is highly efficient.

Acknowledgements

We thank Vijay Karamcheti for his valuable comments when writing the early draft of this paper.

References

[1] R. A. S. Bhoedjang, T. Ruhl, and H. E. Bal. User-Level Network Intrerface Protocols. *IEEE Computer*, 31(11):53–60, Nov. 1998.

[2] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[3] M. Buchanan and A. Chien. Coordinated thread scheduling for workstation clusters under windows nt. In *Proc. of the USENIX Windows NT Workshop*, Aug. 1997.

[4] P. Dasgupta. Parallel processing with windows nt networks. In *Proc. of the USENIX Windows NT Workshop*, Aug. 1997.

[5] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, pages 388–396, Apr. 1997.

[6] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–??, Mar./Apr. 1998.

[7] R. Gillett, M. Collins, and D. Pimm. Overview of the Memory channel Network for PCI. In *Proc. of the 41th IEEE Int'l Computer Conf. (COMPCON '96)*, pages 244–248, Feb. 1996.

[8] J. S. Hansen and E. Jul. A scheduling scheme for network saturated nt multiprocessors. In *Proc. of the USENIX Windows NT Workshop*, Aug. 1997.

[9] R. W. Horst and D. Garcia. ServerNet SAN I/O Architecture. In *Hot Interconnets V*, Aug. 1997.

[10] A. Itzkovitz, N. Niv, and A. Schuster. Dynamic Adaptation of Sharing Granularity in DSM Systems. In *Proc. of the 1999 Int'l Conf. on Parallel Processing (ICPP'99)*, Sept. 1999. To appear.

[11] A. Itzkovitz and A. Schuster. MultiView and Millipage — Fine-Grain Sharing in Page-Based DSMs. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation (OSDI'99)*, pages 215–228, New Orleans, Feb. 1999.

[12] A. Itzkovitz, A. Schuster, and L. Shalev. Millipede: A user-level nt-based distributed shared memory system with thread migration and dynamic runtime optimization of memory references. In *Proc. of the USENIX Windows NT Workshop*, Aug. 1997.

[13] S. M. Paas, T. Bemmel, and K. Scholtyssik. Win32 api emulation on unix for software dsm. In *Proc. of the 2nd USENIX Windows NT Symposium*, Aug. 1998.

[14] S. Pakin, V. Karamacheti, and A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2):60–73, 1997.

[15] D. J. Scales and K. Gharachorloo. Shasta: a system for supporting fine-grain shared memory across clusters. In *Proc. of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Mar. 1997.

[16] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVI)*, pages 297–306, Oct. 1994.

[17] D. A. Solomon. *Inside Windows-NT, 2nd Edition*. Microsoft Press, 1998.

[18] W. E. Speight and J. K. Bennett. Brazos: A third generation dsm system. In *Proc. of the USENIX Windows NT Workshop*, Aug. 1997.

[19] K. Thitikamol and P. Keleher. Multi-threading and remote latency in software dsms. In *Proc. of the 17th Int'l Conf. on Distributed Computing Systems (ICDCS-17)*, May 1997.

[20] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 303–316, 1995.

[21] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, May 1992.

[22] T. von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, 31(11):61–68, Nov. 1998.

[23] Y. Zhou, L. Iftode, K. Li, J. P. Singh, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed consistency and coherence granularity in dsm systems: A performance evaluation. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 193–205, June 1997.