Porting a User-Level Communication Architecture to NT: Experiences and Performance

Yuqun Chen, Stefanos N. Damianakis, Sanjeev Kumar, Xiang Yu, and Kai Li
Department of Computer Science
Princeton University, Princeton, NJ-08544
{yuqun, snd, skumar, xyu, li}@cs.princeton.edu

Abstract

This paper describes our experiences in porting the VMMC user-level communication architecture from Linux to Windows NT. The original Linux implementation required no operating system changes and was done entirely using device drivers and user-level libraries. Porting the Linux implementation to NT was fairly straightforward and required no kernel modifications. Our measurements show that the performance of both platforms is fairly similar for the common data transfer operations because they bypass the OS. But Linux performs better than NT on operations that require OS support.

1 Introduction

The primary goal of the SHRIMP project is to investigate how to design high-performance servers by leveraging commodity PC hardware and commodity PC software. A key research component is to design and implement a communication mechanism whose performance is competitive with or better than that of custom-designed multicomputers. The challenge is to achieve good communication performance without requiring special operating system support. This paper describes our design decisions, experience, and performance results of porting our virtual memory-mapped communication (VMMC) mechanism and stream sockets to Windows NT.

Our previous work on fast communication for PC clusters was done for the Linux operating system. During the initial phase of the project, we studied how to design a network interface to support VMMC and successfully implemented a 16-node prototype PC cluster with our custom-designed network interfaces and an Intel Paragon routing network [5, 7, 6]. The VMMC mechanism, our low-level communication layer, performs direct data transfers between virtual memory address spaces, bypassing the operating system. Its main ideas are separating control from data transfer and using virtual memory mechanism to provide full protection in a multi-programming environment. We demonstrated that VMMC on Myrinet achieves 3.75 μs end-to-end latency using a custom-designed network interface.

During the second phase, we designed and implemented a VMMC mechanism with extended features for PC clusters connected via Myrinet, a commercial system area network and the operating system was again Linux [15, 9]. With a programmable network interface (Myrinet), we showed that VMMC achieves about 14 μs end-to-end latency and delivers bandwidth close to the hardware limit. We also designed and implemented several compatibility communication layers such as message-passing [1], RPC [3], and Unix stream sockets [14], and showed that they deliver good performance.

We recently ported VMMC and several compatibility software libraries to PCs running Windows NT 4.0 and 5.0 Beta. Several factors motivated our entry into the NT world. First, we want to leverage more PC hardware devices and software systems that are available for the NT platform. We particularly want to use high performance graphics cards that have only NT drivers. Second, we needed good kernel support for SMPs. Linux did not have such solid support for multiprocessors. The ever changing nature of Linux kernel sources also poses a daunting task for maintaining our VMMC software up-to-date. It also makes it really difficult to distribute our software to other research institutes. Porting VMMC to Windows NT eliminates this concern and allows us to take advantage of the huge NT user base.

This paper describes our design decisions, porting experience, and performance results. We also address the questions, whether the promise of requiring no special OS support still holds when port-

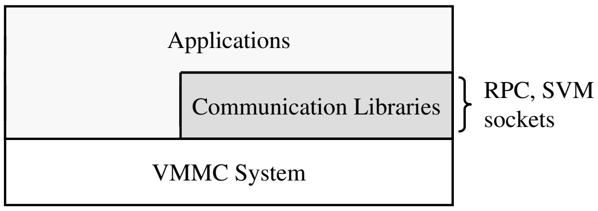


Figure 1: Communication Architecture

ing the communication mechanisms to NT, and whether the porting retains good communication performance. Lastly we would like to share our experience with the readers about the pros and cons of porting user-level communication mechanisms to Windows NT.

2 Communication Architecture

The communication architecture developed in the SHRIMP project (Figure 1) consists of two layers: (i) high-level communication libraries, and (ii) virtual memory-mapped communication (VMMC) mechanisms. Applications such as scientific computation, parallel rendering, and storage servers can use the VMMC primitives directly or use the high-level communication libraries.

The VMMC layer consists of a set of simple, yet powerful and efficient communication primitives for protected, user-level communication in a multi-programming environment. The basic idea of VMMC is to transfer data directly between virtual address spaces across a network with minimal overhead. The approach taken was to provide a mechanism to set up protected communication among virtual memory address spaces across a network, and a separate mechanism to initiate direct data transfers between virtual address spaces at user level efficiently, conveniently, and reliably. The VMMC layer is system dependent. It provides very low-overhead communication with network interface hardware support [5, 7] and quite efficient communication with a programmable network interface [15].

The high-level communication libraries take advantage of VMMC primitives to support applications which use legacy or complex communication APIs. In the SHRIMP project, we implemented Unix stream sockets [14], remote procedure call [4], NX message-passing library [1], and various shared virtual memory systems [17, 21, 6]. Critical to our layered approach is the requirement that the underlying VMMC layer provides convenient mechanisms for the high-level communication libraries to implement zero-copy protocols, in particular, connection-oriented communication protocols. It is also important to consider the tradeoffs as to which layer implements reliable communication. Our approach was to implement a retransmission protocol at VMMC layer, so that we can achieve low-latency and high-bandwidth reliable communication and simplify the construction of high-level libraries such as stream sockets.

The main rationale of the two-layer communication architecture is to minimize system dependence. Because the communication libraries sit directly on top of VMMC, they are system independent. Our porting of the communication architecture from Linux-based PC clusters to Windows NT clusters validated our design rationale.

3 Porting VMMC to NT

In this section, we first describe the components of VMMC, then discuss porting issues, and finally report the lessons that we learned from our porting experience.

3.1 VMMC Components

The VMMC layer consists of three sets of primitives. The first set contains the primitives for setting up VMMC communication between virtual address spaces in a PC cluster. The primitives include `import` and `export` as well as `unimport` and `unexport` of communication buffers in virtual address spaces. Implementation of these primitives requires system calls because they need access to information about memory pages and permission checkings. These primitives are intended for applications to use during communication setup phase. In general their performance is not very critical.

The second set of primitives are for data transfer between virtual address spaces. They include synchronous and asynchronous ways to send data from a local virtual memory buffer (VM buffer) to an imported remote VM buffer, and to fetch data from an imported remote VM buffer into a local VM buffer. Initiation of these primitives is done entirely at user level, after the protection has been set up via the setup primitives. Thus data transfers are very fast and also fully protected in a multi-programming environment. An optional notification mechanism allows a data transfer primitive to invoke a user-level handler in the remote process, upon transfer completion. VMMC also includes primitives for redirecting incoming data into a new VM buffer in order to maximize the opportunity to avoid data copy when implementing high-level communication libraries.

The third set of primitives are utility calls for applications to get information about the communication layer and the low-level system. The primitives also include calls to create remote processes, and obtain and translate node and process ids.

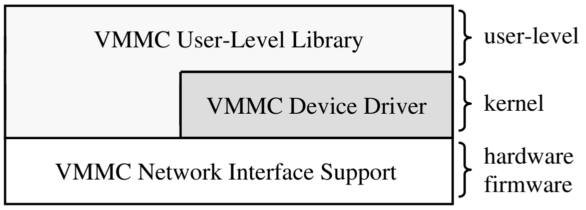


Figure 2: The VMMC System Architecture

Figure 2 shows our initial implementation of VMMC for a Myrinet-based PC cluster which runs on Linux OS. This implementation consists of three components: the network interface (NI) firmware, a device driver, and a user-level library. The NI firmware (also called Myrinet Control Program or MCP) implements a small set of hardware commands for user-level, protected VMMC communication. The device driver initializes the NI hardware, downloads the MCP, and performs firmware initializations. The device driver also implements the setup primitives of VMMC. The user-level library implements all data transfer primitives. For each virtual address space, there is a memory-mapped command buffer for the user program to initiate data transfer commands at user level.

A unique feature of the VMMC is the use of a User-managed TLB (UTLB) to perform address translation. The UTLB mechanism does demand page-pinning. It pins a local buffer when it is used in communication for the first time. Subsequent data transfers using the same buffer will be able to translate addresses efficiently and safely at user level. The UTLB mechanism may unpin the buffer according to some memory allocation strategy. For applications that display spatial locality in their communication patterns, the cost to pin and unpin the virtual pages is amortized over multiple communication requests. A recent paper [9] provides the details about the design, implementation, and evaluation of UTLB.

In addition, the VMMC also implements a retransmission protocol at the data link level and a dynamic network topology mapping mechanism to provide high-level libraries and programs with a reliable low-level communication layer.

3.2 Porting to NT

Although NT is quite different from Linux, the architecture components of the Linux VMMC implementation fit NT quite well. We therefore maintained the original structure. However, we did need to make several changes besides restructuring the device driver for NT.

The first is to use NT kernel threads to enable VMMC for shared memory multiprocessor (SMP) nodes. The original VMMC for Linux did not use threads, because until recently Linux did not officially support threads. Because VMMC provides protected communication in a multi-programming environment, our SMP clusters using Linux used multiple address spaces on each node to implement communication libraries such as SVM [21]. The main drawback of this approach is that the cost of a process context switch is much higher than a kernel thread context switch. During the NT port, we used NT events, semaphores and conditional variables to perform synchronizations for VMMC calls so that multi-threaded user programs can safely use VMMC without explicitly performing synchronizations.

The second is to use NT kernel threads to implement notifications. The Linux version uses Unix signal to implement a notification. After the NI DMAs a packet into the host memory, it puts the the virtual address and the value of the last word in the data into a message queue, which is shared by driver and mcp, and then triggers an interrupt to the host processor. On Linux, the host interrupt handler directly sends a signal to the receiving process. The kernel schedules the signal handler to run in the user process' context. NT does not provide Unix-style signals. Instead, the host interrupt handler (actually a Deferred Procedure Call or DPC handler) triggers an NT event on which a dedicated notification thread is waiting. Upon wakeup, the notification thread calls the appropriate handler in user context. Our NT port also allows a thread to wait on an event explicitly, bypassing the notification handler mechanism. The event can allocated for each exported buffer. This is a useful feature for multi-threaded applications, because each thread can explicitly wait on its own messages.

The third is to deal with remote process creation. The Linux version uses `rsh` to spawn a remote process. However, we cannot use this method because, at the time of porting, we were not aware of any method to specify a working directory for a remotely launched program (on a central file server). To get around this restriction, remote process creation is handled by a service running on each node in the cluster. The master process uses RPC to talk to the remote service to create a process. The service then maps the remote directory to a local drive letter and starts the program. We also implemented cluster management such as redirecting the output of each process to central file system, through the service,

3.3 Lessons Learned

Windows NT provides more mature support for device driver development than Linux. NT has complete documentation, many example sources, and a good kernel debugger to work with. But, since we do not have NT kernel source code, sometimes it is difficult to pinpoint bugs in the driver. On Linux, there is little documentation on device drivers; we had to study existing device drivers and kernel source code. Further, we had to modify the kernel source to export more symbols that were needed for our VMMC device driver.

NT supports SMP and provides a set of kernel primitives to support thread synchronization. However, they can not be used in all places, ie. interrupt handler, kernel DPC handler, and device IOCTL can use different subset of primitives. It took us some time to figure out which primitives to use.

Much of the porting effort was focused on making VMMC thread-safe to take advantage of NT's kernel threads. For example, our Linux SVM system used multiple processes on each SMP, while our newer NT SVM uses a single process with multiple kernel threads. Using kernel threads resulted in a significant performance improvement for SVM.

4 Porting Sockets to NT

The user-level stream sockets library allows applications based on stream sockets to leverage our fast communication substrate with no code modification. This library was initially developed for VMMC Linux clusters. In this section we first describe the sockets model and how it differs on Windows NT. Then we discuss the issues involved in implementing Winsock library using the Unix stream sockets as the base. We end this section with the lessons learned during the porting process.

4.1 Stream Sockets

A wide variety of distributed applications rely on the Berkeley Unix stream sockets model for interprocess communication [18]. The stream socket interface provides a connection-oriented, bidirectional byte-stream abstraction, with well-defined mechanisms for creating and destroying connections and for detecting errors. There are about 20 calls in the API including `send()`, `recv()`, `accept()`, `connect()`, `bind()`, `select()`, `socket()`, and `close()`. Traditionally, stream sockets are implemented on top of UDP so that applications can run across any networks using TCP/IP protocol.

Our stream sockets implementation is for system area networks and was originally implemented on top of *VMMC* for PC clusters using Linux.

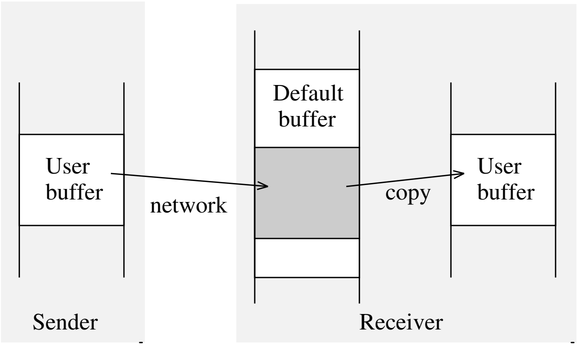
The *VMMC* model now supports a mechanism called *transfer redirection*. The basic idea is to use a default, *redirectable* receive buffer in case when a sender does not know the final receive buffer addresses. Stream sockets makes heavy use this mechanism.

Redirection is a local operation affecting only the receiving process. The sender does not have to be aware of a redirection and always sends data to the default buffer. When the data arrives at the receive side, the redirection mechanism checks to see whether a redirection address has been posted. If no redirection address has been posted, the data will be moved to the default buffer. Later, when the receiver posts the receive buffer address, the data will be copied from the default buffer to the receive buffer, as shown in Figure 3(a).

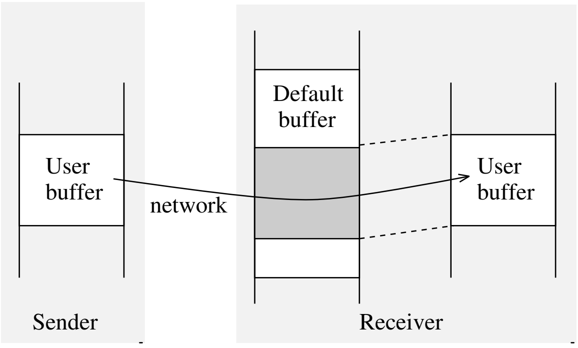
If the receiver posts its buffer address before the message arrives, the message will be put into the user buffer directly from the network without any copying, as shown in Figure 3(b). If the receiver posts its buffer address during message arrival, the message will be partially placed in the default buffer and partially placed in the posted receive buffer. The redirection mechanism tells the receiver exactly how much and what part of a message is redirected. When partial redirection occurs, this information allows the receiver to copy the part of the message that is placed in the default buffer.

VMMC stream sockets performs quite well with a one-way latency of 20 μs and a peak bandwidth of over 84 Mbytes/s. The bandwidth performance is due, in large part, to redirection. Without redirection bandwidth would be limited by the system copy bandwidth because one copy would be required by the receiver to move incoming data to its final destination.

Because our stream sockets is a user-level library, it does not allow open sockets to be preserved across `fork()` and `exec()` calls. With `fork`, the problem is arbitrating socket access between the two resulting processes. `exec` is difficult because VMMC communicates through memory and `exec`



(a) A copy takes place when the receiver posts its buffer address too late.



(b) A transfer redirection moves data directly from network to the user buffer .

Figure 3: Transfer redirection uses a default buffer to hold data in case receiver posts no buffer address or posts it too late, and moves data directly from the network to the user buffer if the receiver posts its buffer address before the data arrives.

allocates a new memory space for the process. This limitation was not due to the sockets implementation but rather it is a fundamental problem with user-level communications in general[1].

4.2 Implementing NT VMMC Sockets

Windows sockets, or WinSock, adapts the sockets communication model to the Windows programming environment. As a result, WinSock contains many Windows-specific extension functions in addition to the core Unix stream socket calls.

Instead of implementing Winsock, we decided to support only a subset of Winsock calls, the same set of functions that were in our VMMC sockets Linux implementation. The current implementation does not support (for now) out-of-band (OOB) data, scatter/gather operations, and polling with `recv()` (MSG_PEEK). We call this implementation NT VMMC Sockets.

The two main issues in implementing NT VMMC sockets were: (i) seamless *user-level* integration of the library for binary compatibility with existing applications, and (ii) integration of user-level VMMC sockets with NT kernel sockets. The solution that satisfied both requirements was using a wrapper DLL (dynamic-link library) that intercepts WinSock calls and allowed for a user-level implementation of sockets.

NT provides support for sockets via two DLLs, `wsock32.dll` (WinSock 1.1) and `ws2_32.dll` (WinSock 2.0). All calls in `wsock32.dll` end up calling into either `ws2_32.dll` or `mswsock.dll`. Since we only support a subset of the original Berkeley sockets functions that are in WinSock 1.1, we just need to deal with the functions in `wsock32.dll`. Our wrapper DLL for `wsock32.dll`, intercepts WinSock 1.1 calls and implements user-level sockets using VMMC. To disambiguate between the two identically named files we refer to the VMMC-based library as $\texttt{wsock32.dll}_{\text{vmmc}}$. By simply placing a copy of $\texttt{wsock32.dll}_{\text{vmmc}}$ in the application's directory (or path) WinSock 1.1 calls are automatically intercepted. Removing (or renaming it) allows the application to use NT's `wsock32.dll`.

We also wanted to support both *types* of stream connections: user-level VMMC and NT kernel (i.e. Ethernet). In order to accomplish this, the user-level sockets library allocates a socket descriptor table that contains one entry for each open socket, regardless of the socket type. When NT kernel sockets are used, $\texttt{wsock32.dll}_{\text{vmmc}}$ forwards calls through to either `ws2_32.dll` or `mswsock.dll`, while still maintaining a descriptor table entry. Also, our library uses calls to `wsock32.dll` in order to bootstrap the VMMC connection. Figure 4 illustrates the software layers in our user-level implementation.

Building a wrapper DLL $\texttt{wsock32.dll}_{\text{vmmc}}$ was straightforward. We used the `dumpbin` utility provided by Microsoft Win32 SDK to produce the list of exported functions in the `wsock32.dll` as well as their forwarding information. We wrote a perl script to find the function prototype for each exported function from a list of header files and produce a stub call that uses `GetProcAddress()` to obtain the address of the corresponding function in either `ws2_32.dll` or `mswsock.dll`. This script also produces the definition (.def) file needed to build the DLL along with the correct function ordinals. $\texttt{wsock32.dll}_{\text{vmmc}}$ uses `LoadLibrary()` to load both `ws2_32.dll` and `mswsock.dll` so that it

[1]Maeda and Bershad [19] discuss how to implement `fork()` and `exec()` correctly in the presence of user-level networking software.

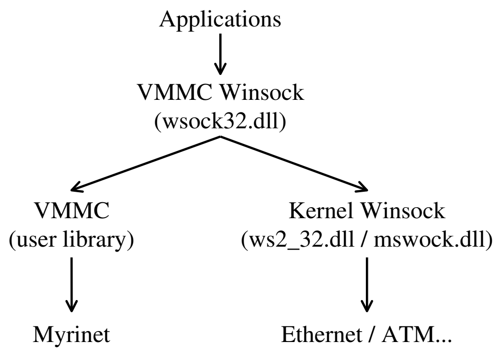


Figure 4: Socket layers

can, as needed, forward calls to them.

4.3 Lessons Learned

It turned out that we were very lucky: A staff researcher at Microsoft later pointed to us that our approach to produce a wrapper DLL worked because `wsock32.dll` is not a *preloaded* DLL. Preloaded DLL is a Microsoft term for DLLs that are loaded during system startup. They cannot be replaced by any wrapper DLLs.

However, one can easily disable the preload attribute of a DLL by removing its name from a registry (KnownDLL). Using this trick, we were able to build a wrapper around `kernel32.dll` to build a user-level fast file system. Working with DLLs gave us the flexibility to leverage user-level communication and minimize the overhead of kernel calls.

The Linux version of VMMC sockets maintains an internal socket descriptor table that is index with the socket descriptor. But WinSock adds a new data type, `SOCKET`, because socket descriptors are no longer equivalent to file descriptors, as in Linux. This change forces `wsock32.dll`$_{\text{vmmc}}$ to convert `SOCKET` values to table indices and vice versa. A small but important detail.

VMMC sockets is thread-safe and written with support for a preemptive user-level threads package for Linux [16]. When VMMC sockets was being developed, support for kernel threads in Linux was not dependable and was still work in-progress. Therefore, we decided to use a user-level threads package instead. Using user-level threads required that we worry about invoking system calls and possibly blocking the process. We added functionality to the VMMC sockets implementation to stop this from happening. VMMC sockets also integrated support for asynchronous I/O. A special socket was reserved through which requests were made to asynchronous I/O. All of this additional complexity was necessary because Linux then lacked kernel threads. Porting to NT allowed us to take advantage of kernel threads and simplify the thread support for `wsock32.dll`$_{\text{vmmc}}$. We no longer needed to support asynchronous I/O or or worry about threads calling blocking system calls.

Recall that the VMMC Linux sockets have the limitation that sockets are not preserved across `fork()` and `exec()` calls. Windows NT eliminates this limitation because it does not support the `fork/exec` model for process creation. But NT introduces a the same problem in a different form because processes can ask share sockets. Still, NT is a net gain for VMMC sockets because socket sharing in NT is less pervasive than `fork/exec` on Linux.

Our experience with a distributed file system (DFS) [22] led us to extend the sockets API to support pointer-based data transfers [13]. Using pointers allows the DFS to eliminate copying from the transfers of file data. We use `ioctlsocket()` to access the pointer-based API of `wsock32.dll`$_{\text{vmmc}}$.

Finally, we plan to add support for out-of-band (OOB) data, scatter/gather operations, and polling with `recv()` (MSG_PEEK). Further we want to produce a more complete implementation of VMMC sockets that supports many WinSock 2.0 calls directly.

5 Performance evaluation

VMMC minimizes OS involvement in common communication datapaths. However, some OS involvement is still necessary for tasks like communication setup and virtual-to-physical address translation. Therefore, we want to evaluate the impact of the operating system on the three distinct aspects of user-level communication:

- OS impact on the communication setup performance
- OS impact on the data transfer performance
- OS impact on the control transfer performance

Methodology The ideal approach would be to run identical applications on both Linux and Windows NT platforms. Unfortunately, we do not have such applications for a variety of reasons. First, our OpenGL applications are written for the NT platform because it is the single most popular platform that all high-end graphics accelerator vendors support. Second, the shared-memory applications currently use an SVM library that was solely developed for the Win32 platform, primarily due to its mature support for kernel threads. The incompatibility between the Linux and Win32 API also made it very difficult to have a single SVM protocol library that works for both OSes.

Since we do not have the identical applications on the two platforms, we perform the evaluation as follows: First, we use microbenchmarks to measure the various aspects of the communication system and the relevant aspects of operating systems on the two platforms. Then we instrument the applications to measure the frequency of occurrence of the various events like UTLB misses and notifications on applications running on windows NT platform. Finally, we combine the two measurements to predict the impact of the OS on the applications performance.

Platform All measurements are performed on a cluster of PCs. Each has a 450 MHz Intel Pentium II processor, 1 GB memory and a Myricom network interface card with LANai 4.x microprocessor and 1MB on-board SRAM. The PCs are set up for dualboot between Linux 2.0.36 kernel[2] and Windows NT 4.0 SP5.

5.1 Impact on Communication Setup

The VMMC communication is set up via *export* and *import* calls. A process can *export* the read and/or write rights on its virtual memory buffer. Another process on a different or the same machine can gain the access rights through an import operation; afterwards, it can directly read from or write to this buffer. The export-import semantics provides protection in VMMC. Export is implemented inside the device driver and invoked through the *ioctl* mechanism. Table 1 lists the cost of kernel entry and exit using the ioctl mechanism.

Platform	Linux	Windows NT
ioctl kernel entry	1.1 μs	6.6 μs
ioctl kernel exit	1.0 μs	7.8 μs

Table 1: Overhead of kernel calls

The export call pins the exported buffer in physical memory. In current implementation, exported pages are never unpinned. The cost of pinning a user buffer is listed in Table 2. On NT, exporting a one-page buffer takes 43 μs of which 18.3 μs is OS specific (ioctl call + pinning a page) while importing a memory buffer costs 68 μs none of which involves the OS.

[2] We can not simply use the newer Linux 2.2.x kernel, because bringing our VMMC software up to the 2.2 kernel requires non-trival changes.

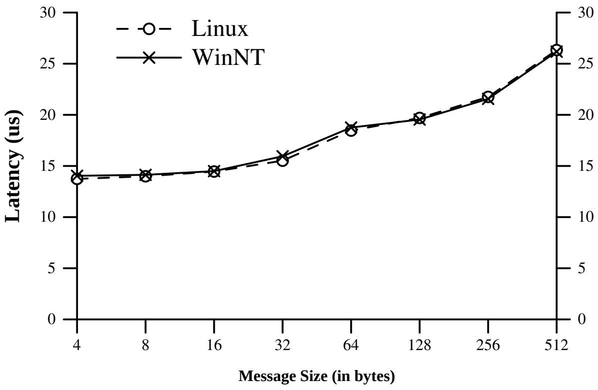


Figure 5: Remote send latency for small messages

In most of our applications, the connection setup occurs only at the beginning. For client-server applications that use sockets, the connection setup happens each time a socket is opened. In both cases, the setup cost accounts for a small fraction of the execution time. The small difference in the setup performance on the two platforms should have little impact on application performance.

5.2 Impact on Data Transfer

The impact of operating system on the performance of user-level data transfer is quite limited. In VMMC, the OS involvement is needed only when the address translation is not available in the UTLB translation table. To evaluate this effect, we first measure the performance in the common case in which all the address translation is in the UTLB translation table and therefore requires no operating system involvement. We then measure the UTLB overheads when the translations are not available in the translation table. Finally we measure the UTLB overheads in a few real applications and parallel programs.

Common case Figure 5 shows the one way latency while Figure 6 shows the bandwidth of synchronous remote send operations on the two platforms. The latency is measured using a pingpong benchmark in which two processes on two different machine send synchronous messages to each other back and forth. The bandwidth is measured using a pair of processes on two different machines where one process continuously does synchronous sends to the second process. Because the PCI bus bandwidth(133 MB/s) and Myrinet bandwidth(160 MB/s) are very high, the dominant portion of time

No. Of Pages		1	4	8	12	16
Linux	Pin	3.6 μs	4.0 μs	5.0 μs	6.4 μs	10.7 μs
	Unpin	2.8 μs	11.0 μs	19.1 μs	28.2 μs	34.5 μs
Windows NT	Pin	2.9 μs	8.9 μs	17.0 μs	24.0 μs	32.8 μs
	Unpin	1.6 μs	4.6 μs	8.6 μs	12.5 μs	16.7 μs

Table 2: Overhead of pinning and unpinning pages in memory (excluding ioctl overhead).

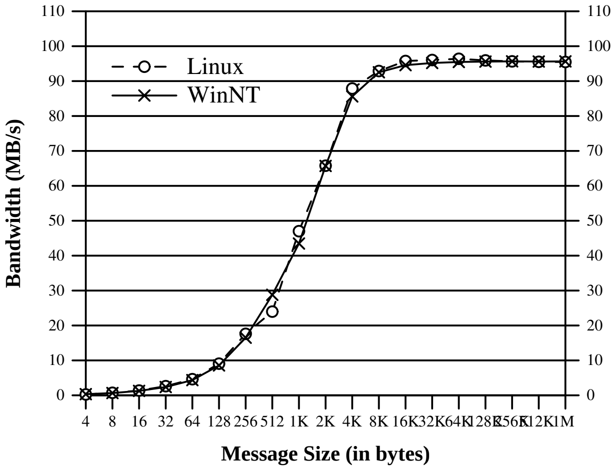


Figure 6: One-way send bandwidth

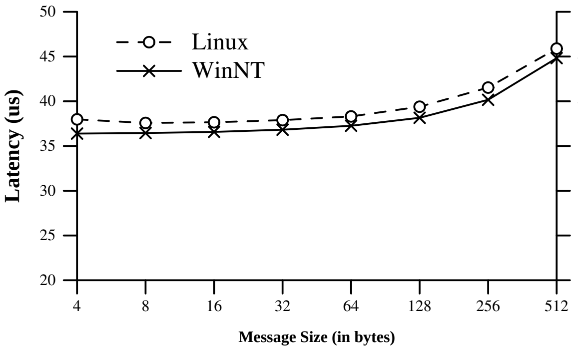


Figure 7: Remote fetch latency

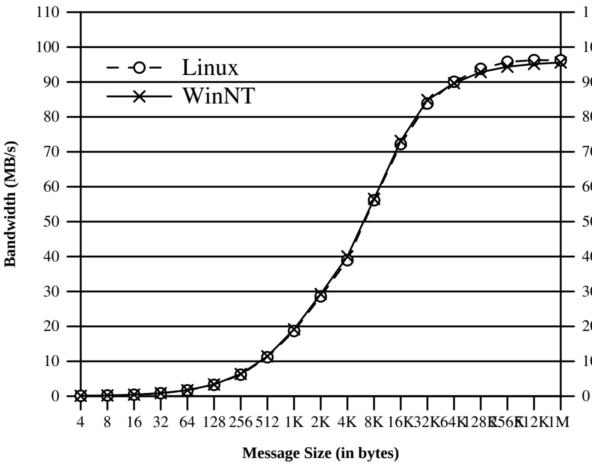


Figure 8: Remote fetch bandwidth

for small message communication is the processing overhead on the the network interface, therefore is independent of the OS. The one-way latency for small messages is 14 μs while the peak bandwidth is about 96 MB/s on both platforms.

The VMMC communication model also provides a remote-read capability to fetch data from the memory of the remote machine without the involvement of host processor that machine. Figures 7 and 8 show the latency and bandwidth performance respectively. Both measurements are performed using a pair of processes on two different machines where one process repeatedly performs reads from the second process' memory using the remote-read operation.

As expected, we see that the OS has little impact on the data transfer in the common case. Later optimizations on NT further reduce the latency by 6 μs.

UTLB overheads Two kinds of overhead occur in the slow path but not the fast path. The first overhead occurs when there is a *host translation miss*: when the local source buffer of a send operation or the local destination buffer of a fetch operation is not currently pinned and needs to be pinned for DMA transfer. Since only a limited amount of virtual memory can be pinned, this may result in unpinning other pages. The second overhead is caused by a translation miss in the network interface UTLB cache; in this case, the network interface directly DMAs the translation from the host memory. Note that only the first overhead involves the operating system (Table 1 for ioctl overhead and Table 2 for pinning and unpinning overhead). To evaluate the overhead of UTLB operations in a more realistic scenario, we instrumented 7 applications that run either directly on VMMC NT or on VMMC Winsock and report the measured UTLB overhead.

Applications From the SPLASH-2 suite, we chose 4 SVM applications, *radix*, *barnes*, *fft*, and *water*-

Applications	vis	glaze	vdd	fft	barnes	radix	water
memory footprint	21.4 MB	39.9 MB	78.9 MB	57.4 MB	16.6 MB	54.4 MB	8.9 MB
host lookup hit rate	99.93%	99.99%	99.99%	79.38%	98.62%	67.36%	99.35%
avg lookup hit cost	0.65 μs	0.63 μs	2.41 μs	0.32 μs	0.46 μs	0.32 μs	0.40 μs
avg lookup pages	1.00	1.94	54.8	1.00	1.00	1.04	1.00
avg pin overhead on NT	54.3 μs	401.6 μs	939.0 μs	47.3 μs	53.1 μs	50.9 μs	55.6 μs
avg pages pinned on NT	1.00	97.0	289.0	1.00 μs	1.13 μs	1.00 μs	1.25 μs
avg miss overhead on NT	63.4 μs	557.1 μs	1294 μs	50.7 μs	57.5 μs	54.6 μs	60.0 μs
predicted ovhd on Linux	41.7 μs	410.1 μs	882 μs	34.7 μs	40.5 μs	42.0 μs	47.4 μs
avg host overhead on NT	0.69 μs	0.69 μs	2.54 μs	10.71μs	1.25 μs	18.04μs	0.79 μs
predicted ovhd on Linux	0.68 μs	0.67 μs	2.50 μs	7.41μs	1.01 μs	13.92μs	0.71 μs

Table 3: UTLB performance without memory constraint

Applications	fft	barnes	radix	water
memory footprint	57.4 MB	16.6 MB	54.4 MB	8.9 MB
memory constraint	50.0 MB	16.0 MB	30.0 MB	6.0 MB
host lookup hit rate	64.46%	96.67%	54.99%	97.63%
host pin-page rate	35.54%	3.33%	45.01%	2.37%
host unpin-page rate	23.36%	2.13%	36.10%	2.27%
avg lookup hit cost	0.36 μs	0.50 μs	0.35 μs	0.40 μs
avg pin overhead on NT	39.1 μs	40.0 μs	42.1 μs	29.6 μs
avg pages per pin on NT	1.00	1.07	1.00	1.07
avg unpin overhead on NT	33.2 μs	36.7 μs	35.2 μs	35.0 μs
avg miss overhead on NT	42.2 μs	49.6 μs	68.7 μs	32.8 μs
predicted ovhd on Linux	21.6 μs	29.3 μs	46.4 μs	8.6 μs
avg host overhead on NT	15.23 μs	2.13 μs	31.1 μs	1.27 μs
predicted ovhd on Linux	7.91 μs	1.45 μs	21.1 μs	0.59 μs

Table 4: UTLB performance with memory constraint

nsquare. They all use the SVM protocol developed by our colleagues at Princeton [26]. The SVM protocol communication is built directly on top of VMMC, using the send and remote fetch mechanism. In addition, we selected 3 applications from Princeton Display Wall Project [11] *glaze*, *isosurface*, and *vdd*. These applications use VMMC Winsock as their underlying communication mechanism.

Vis is a visualization program, implemented by our colleagues that uses a cluster of 13 PCs to visualize isosurfaces of scientific data. The program has three components: client control, isosurface extraction, and rendering. The client control runs on a single PC to implement the user interface for users to steer the entire visualization process. The isosurface extraction uses 4 PCs in the cluster to extract isosurfaces in parallel, and sends them to the rendering program which runs on 8 PCs. Each renderer PC simply receives isosurfaces, renders and displays the rendered images on a tiled, scalable display wall. VMMC Winsock is used for all inter-process communication among the PCs running the client control, isosurface extraction, and rendering programs.

Glaze is a commercial OpenGL evaluation program from Evans & Sutherlands. An OpenGL framework, developed by our colleagues, allows us to use one PC to drive OpenGL animation of any commercial software on a tiled display. We ran the glaze program on a PC; a custom wrapper DLL (opengl32.dll) intercepts the OpenGL calls made by the program and distributes the commands to 8 separate PCs which together drive a 2x4 tiled display. The wrapper DLL transmits rendering commands over the VMMC Winsock layter to the renderers. The renders behave just like those in *vis*: they simply receive the commands and render them for its portion of the tiled display.

Vdd is a virtual display driver that implements a large desktop that has the same resolution (3850x1500) as the Display Wall. It is installed on a 450 MHz Pentium II PC running NT 5B2 OS. A user process is responsible for packetizing the updates made to the vdd's memory-resident framebuffer and distributing them to the 8 PCs that drive the wall. We used NT VMMC Sockets with the pointer-based extensions to distribute framebuffer updates efficiently [13].

Results Table 3 presents OS-dependent components of UTLB overhead, without any memory constraint. The miss rates and overheads for the NT platform are measured directly by running the 7 applications on our NT VMMC cluster. We use these numbers, as well as the micro-benchmark results, to predict the overheads on Linux. In the table, *host pin rate* is the ratio between the number of pin operations and the number of UTLB lookups; and similarly for *host unpin rate*. *Memory footprint* is the total amount of distinct virtual memory that is involved in data transfer for each application. Lower page-pin rate (or lookup miss rate) translates into lower average UTLB overhead. Since the host UTLB miss rates are quite low, the OS overhead for pinning and unpinning user buffers has little impact on the average UTLB host overhead.

Under tight memory constraints, such as in a multi-programming environment or a large-memory applications, there may not be enough physical memory to absorb the entire memory footprint of applications. UTLB deals with such low-memory situations by unpinning virtual pages that are not currently involved in data transfer. This is a critical feature for UTLB-based VMMC to be useful in a multi-programming environment. Table 4 presents the the same experiments with additional memory constraints such that the total amount of application-pinnable physical memory is less than the memory footprint. Due to varying nature of the applications, the memory constraint is set differently on a per-application basis. And also, it turned out that our Display Wall applications require most of their working sets to be pinned as receive buffers. This is because each DisplayWall application uses a small buffer (often less than 1 MB) to gather commands and send the whole chunk to the receivers. Therefore, we only present the memory-constrained results for the 4 SVM applications. Again, the measurements are all taken by running the applications on the NT cluster, and predictions are made for the Linux platform.

With low memory constraint, we see higher UTLB miss rates and page unpin rates. Note that our prediction for UTLB performance on Linux suggests that faster page-pin and page-unpin OS calls further reduce the average UTLB overhead by as much as 50%. We conclude that improving system calls such as page-pin and page-unpin benefits user-level communication.

5.3 Impact on Control Transfer

In VMMC, messages are deposited directly into the memory of the receiving process without interrupting the host processor. The receiving process can detect message arrival by polling memory locations. But it can also use the VMMC notification mechanism to reduce the CPU polling overhead. For instance, the sockets library uses notifications extensively to avoid polling while waiting for data to arrive on a socket.

The VMMC notification mechanism allows a receiving process to associate a message handler with each exported buffer. The message handler is executed when a message with notification request arrives in the exported buffer. Since interrupts are used to deliver notifications to the corresponding process, different OS platforms show different performance.

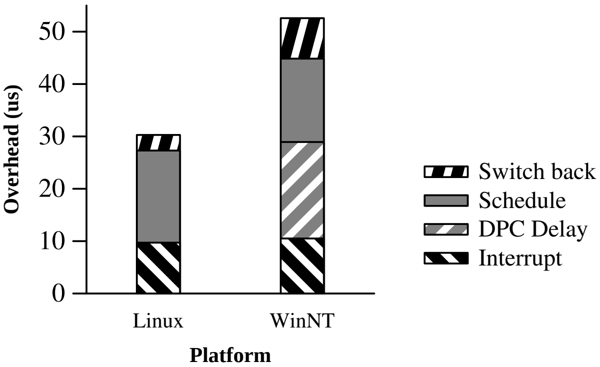


Figure 9: Receive-side notification overhead breakdowns

Figure 9 shows the breakdown of the notification overhead on the receiving machine. The interrupt cost is the time from when the network interface issues the interrupt to the time when the interrupt handler in the driver starts to execute. Surprisingly[3], both Linux and NT take about 10 μs to reach the handler on our test platform. Once interrupted, Linux uses 17 μs to schedule the user level message handler through the process signal mechanism. On NT, because there are restrictions on what can be executed in the interrupt handler, the interrupt handler has to queue a Deferred Procedure Call (DPC) in the kernel which invokes the user-level notification handler later. By the time the control reaches the start of the DPC handler,

[3]The cost on some other similar machines is much less. We are still investigating the cause for this high overhead. This won't affect our platform comparison.

18 μs has already passed. After this, NT uses 16 μs to schedule the user level message handler by waking up the user thread that executes the message handler. Once the notification handler terminates, Linux uses 3 μs to return to the interrupted process while NT uses 7 μs to switch back to the interrupted thread. The overall overhead for a notification is 30 μs on Linux and 52 μs on NT.

6 Related Work

User-level communication architectures for system-area networks is a very active research area [15, 8, 12, 23, 20, 2, 24, 10].

But, only two other user-level communication systems are currently available for Windows NT. A Myrinet-based Fast Messages (FM) [20] implementation has recently been ported to the NT platform. Since it is implemented entirely at user-level, it does not require any OS support. U-net [2] has also been ported to an NT cluster. However, they use Fast Ethernet and require kernel modifications to implement low-overhead kernel entry and exit.

While this paper only presents the results of our porting VMMC to Windows NT, there is a lot of published work describing the various efforts that are part of the SHRIMP project. The VMMC mechanism [5, 15, 9], our low-level communication layer, performs direct data transfers between virtual memory address spaces bypassing the operating system. We have also designed and implemented several compatibility communication software including NX message-passing library [1], RPC [3], and Unix stream sockets [14], and showed that they deliver good performance. Finally, applications are implemented using the higher level APIs: (i) distributed file system [22], (ii) SPLASH2 [25], and (iii) distributed OpenGL graphics applications.

7 Conclusions

Our experience with porting to Windows NT has been positive. NT has fairly complete device driver support and good documentation. In addition, it has good support for threads and SMP systems. We found the DLL mechanism to be very convenient. The Display Wall project uses a Windows NT-based VMMC cluster for high-performance communication.

Our measurements indicate that the OS overhead on Windows NT is significantly higher than on Linux. However, it provides much better functionality especially in terms of threads and SMP systems.

Finally, we find that the VMMC user level communication architecture is successful in delivering high-performance to applications on both platforms.

Our software is publicly available at `http://www.cs.princeton.edu/SHRIMP/`.

Acknowledgments

This project is sponsored in part by DARPA under grant N00014-95-1-1144, by NSF under grant MIP 9420653 and CDA96-24099, and by Intel Corporation.

We would like to thank Paul Pierce from Intel Corp. for doing the initial port of VMMC to NT, Allison Klein and Rudro Samanta from Princeton University for providing us with Display Wall applications, and Richard Shupak from Microsoft Corp. for answering our NT questions. We also want to thank the program committee, anonymous reviewers, and the shepherd of this paper, Thorsten von Eicken, for the helpful feedback.

References

[1] Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Kai Li. Design and Implementation of NX Message Passing Using SHRIMP Virtual Memory-Mapped Communication. In *Proceedings of the International Conference on Parallel Processing*, August 1996.

[2] Anindya Basu, Matt Welsh, and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Presentation at IEEE Hot Interconnects V*, August 1997. Also available as Tech Report TR97-1620, Computer Science Department, Cornell University.

[3] A. Bilas and E. W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *Journal of Parallel and Distributed Computing*, 40(1):138–146, January 1997.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comp. Sys.*, 2(1):39–59, November 1984.

[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[6] Matthias A. Blumrich, Richard D. Alpert, Yuqun Chen, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Edward W. Felten, Liviu Iftode, Kai Li, Margaret Martonosi, and Robert A. Shillner. Design Decisions in the SHRIMP System: An

Empirical Study. In *Proceedings of the 25th Annual Symposium on Computer Architecture*, June 1998. To appear.

[7] Matthias A. Blumrich, Cezary Dubnick, Edward W. Felten, and Kai Li. Protected, User-Level DMA for the SHRIMP Network Interface. In *IEEE 2nd International Symposium on High-Performance Computer Architecture*, pages 154–165, February 1996.

[8] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 245–260, October 1996.

[9] Yuqun Chen, Angelos Bilas, Stefanos N. Damianakis, Czarek Dubnicki, and Kai Li. UTLB: A Mechanism for Translations on Network Interface. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 193–204, October 1998.

[10] Compaq/Intel/Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, December 1997.

[11] Princeton University Computer Science Department. The Princetion Display Wall Project. http://www.cs.princeton.edu/omniwall, 1999.

[12] C. Dalton, G.Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, 1995.

[13] Stefanos N. Damianakis. *Efficient Connection-Oriented Communication on High-Performance Networks*. PhD thesis, Dept. of Computer Science, Princeton University, May 1998. Available as technical report TR-582-98.

[14] Stefanos N. Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream Sockets on SHRIMP. In *Proc. of 1st Intl. Workshop on Communication and Architectural Support for Network-Based Parallel Computing (Proceedings available as Lecture Notes in Computer Science 1199)*, February 1997.

[15] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the IEEE 11th International Parallel Processing Symposium*, April 1997.

[16] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software*. Addison-Wesley, Reading, Massachusetts, 1997.

[17] Liviu Iftode, Cezary Dubnicki, Edward Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of IEEE 2nd International Symposium on High-Performance Computer Architecture*, February 1996.

[18] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System*. Addison Wesley, 1989.

[19] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of 14th ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.

[20] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[21] R. Samanta, A. Bilas, L. Iftode, and J.P. Singh. Home-based SVM protocols for SMP clusters: Design and Performance. In *Proceedings of 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[22] Robert A. Shillner and Edward W. Felten. Simplifying Distributed File Systems Using a Shared Logical Disk. Technical Report TR-524-96, Princeton University Computer Science Department, Princeton NJ, 1996.

[23] J.M. Smith and C.B.S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, 7(4):44–52, July 1993.

[24] H. Tezuka, A. Hori, and Y. Ishikawa. PM: a High-Performance Communication Library for Multi-user Parallel Environments. Submitted to Usenix'97, 1996.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, May 1995.

[26] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, October 1996.