Network Virtual Memory

by

Joon Suan Ong

B.Eng. (Hons), Nanyang Technological University, Singapore, 1994.

M.Sc., University of Bradford, United Kingdom, 1995.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming

to the required standard

The University of British Columbia

July 2003

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

11 July 2003

Date

Department of Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, BC
Canada V6T 1Z4

Abstract

User-mode access, zero-copy transfer, and sender-managed communication have emerged as essential for improving communication performance in workstation and PC clusters. The goal of these techniques is to provide application-level DMA to remote memory. Achieving this goal is difficult, however, because the network interface accesses physical rather than virtual memory. As a result, previous systems have confined source and destination data to pages in pinned physical memory. Unfortunately, this approach increases application complexity and reduces memory-management effectiveness.

This thesis describes the design and implementation of NetVM, which is a network interface that supports user-mode access, zero-copy transfer and sender-managed communication without pinning source or destination memory. To do this, the network interface maintains a shadow page table, which the host operating system updates whenever it maps or unmaps a page in host memory. The network interface uses this table to briefly lock and translate the virtual address of a page when it accesses that page for DMA transfer. The operating system is prevented from replacing a page in the short interval that the network interface has locked that page. If a destination page is not resident in memory, the network interface redirects the data to an intermediate system buffer, which the operating system uses to complete the transfer with a single host-to-host memory copy after fetching in the required page. A credit-based flow-control scheme prevents the system buffer from overflowing.

Application-level DMA transfers only data. To support control transfers, NetVM implements a counter-based notification mechanism for applications to issue and detect notifications. The sending application increments an event counter by specifying its identifier in an RDMA write operation. The receiving application detects this event by busy waiting, block waiting or triggering a user-defined handler whenever the notifying write completes. This range of detection mechanisms allows the application to decide appropriate tradeoffs between reducing signaling latency and reducing processor overhead. NetVM enforces ordered notifications over an out-of-order delivery network by using a sequence window.

NetVM supports efficient mutual-exclusion, wait-queue and semaphore synchronization implementations. It augments the network interface with atomic operation primitives, which have low overhead, to provide MCS-lock-inspired scalable and efficient high-level synchronization for applications. As a result, these operations require lower latency and fewer network transactions to complete compared with the traditional implementations.

The NetVM prototype is implemented in firmware for the Myrinet LANai-9.2 and integrated with the FreeBSD 4.6 virtual memory system. NetVM's memory-management overhead is low; it adds only less than 5.0% write latency compared to a static pinning approach and has a lower pinning cost compared to a dynamic pinning approach that has up to 94.5% hit rate in the pinned-page cache. Minimum write latency is 5.56μs and maximum throughput is 155.46MB/s, which is 97.2% of the link bandwidth. Transferring control through notification adds between 2.96μs and 17.49μs to the write operation, depending on the detection mechanism used. Compared to standard low-level atomic operations, NetVM adds only up to 18.2% and 12.6% to application latencies for high-level wait-queue and counting-semaphore operations respectively.

Table of Contents

List of Tables

List of Figures

Acknowledgements

Working on a Ph.D. is not only an academic exercise, it is also a personal journey. I have been privileged to embark on this journey surrounded with the greatest people. There are so many people to thank and so much to say. These few words below cannot fully express my gratitude, which I owe to every one of them.

To Mike Feeley, for supervising and guiding me through this journey, for his incredible insight into both academic and nonacademic matters, for his unwavering support through the rough periods, for being an example and setting the standard for me to strive for, and for always reminding me to ask the perennial *"Why?"* question. To Norman Hutchinson, for his humor and friendship, for being a patient listener and accommodating my endless ramblings about my work, and for reminding me that I'm done with his quotation in Chapter 10. To Alan Wagner, for playing the devil's advocate and bringing a different and valuable perspective to the research, for encouraging me during the final stretch, and for all the pleasant and enlightening conversations over the years.

The Distributed Systems Group Laboratory is like a second home to me. I am fortunate and honored have everyone in the lab as friends: Elisa Baniassad, Alex & Dima Brodsky, Yvonne Coady, Chamath Keppitiyagama, Jan (Matt) Pederson and many others. To the gang, for the brainstorming sessions that sometimes lead into cool ideas (and many oddball ones), for reminding me that the answer to the ultimate question has always been forty two, for formulating the four axioms of Operating Systems research (which are still consistent), for helping me rediscover the wonders of coffee, and for the friendship, support and encouragement through these years. Also, to Holly Mitchell, for her cheerful smiles and kindness, and to Tsang Kay Mak, for always lending a hand whenever the Myrinet cluster misbehaved.

And most important, to my parents, for encouraging me to begin this journey, and for supporting me throughout all these years.

Joon Suan Ong

The University of British Columbia
July 2003

1 Introduction

New applications demand increasingly high bandwidth and low latency communication. As host and network hardware improves, communication software must capitalize on these advances to meet these demands.

Data transfer and synchronization are two key aspects in network communications. Existing state-of-the-art systems improve data transfer by tightly coupling the application with the network interface, which allows the network to directly access application data efficiently. However, this tight coupling also places significant constraints on memory management for applications and the operating system. As a result, applications must adapt to these constraints and the operating system must cope with reduced resource control to benefit from these new networks. Synchronization is a basic problem in concurrent programming. In particular, efficient synchronization is essential for applications that exhibit fine-grained interaction in order to benefit from low-latency networks. However, existing schemes that do not exploit the underlying network hardware cannot provide the most scalable and efficient implementations. As a result, applications that rely on traditional implementation of synchronization schemes cannot fully take advantage of improved network performance.

This thesis examines the use of intelligent programmable network interfaces to improve data transfer and synchronization with two key contributions. The first contribution is a network-interface design that provides efficient data transfer without the existing memory-management constraints. The second contribution is a design that exploits intelligent network interfaces by augmenting them with efficient primitives that optimize common synchronization idioms. The thesis demonstrates that, by providing suitable functionality on the network interface, data transfer and synchronization operations can be efficient, easy to implement, and incur low overhead.

1.1 High-performance networks

The performance of modern local area network hardware has improved to a point where software overhead is now a dominant factor in the communication performance for distributed applications. Traditional network architectures can no longer deliver the full performance potential of the underlying hardware to applications due to the high host-processor overheads in kernel-based transport stacks.

These traditional kernel-based network architectures severely limit application communication throughput and latency, as well as increase the processor load on the host. The bandwidth for network hardware is improving by an order of magnitude with each new generation (for example: in 10M, 100M and 1Gbps Ethernet). However, the software overhead in the network stack remains roughly unchanged, because host-processor improvements do not directly translate into higher throughput. Other system components, such as host memory bus and IO bus, have not kept up with the network hardware improvements. Throughput is thus limited to the rate at which the kernel is able to transfer data through the network stack.

The kernel protocol-processing and interrupt-handling overheads now dominate the communication latency. An application invoking a system call into the kernel to handle data transmission or a network interface interrupting the kernel to handle data reception will significantly increase the application's end-to-end messaging latency, especially for small transfers. Low latency is particularly critical to the scalability of distributed applications that need to synchronize over the network. These time-sensitive synchronization operations are not bandwidth bound, because they usually do not contain much data.

Interrupting the host processor to handle network messaging is costly, especially on modern processors. Newer host-processor architectures achieve higher performance using sophisticated parallelism techniques such as deep pipelines, out-of-order execution and speculative branching. However, they also suffer from higher penalties during an interrupt because they have to flush all their internal state before servicing the interrupt. The processor time incurred during an interrupt is often unfairly accounted to the unfortunate process that is currently scheduled.

1.2 Existing state-of-the-art approaches

Three important approaches have emerged for high performance communication. They are *user-mode* access, *zero-copy* transfer and *sender-managed* communication. The first two approaches are now widely recognized as essential for lowering communication overheads. Both

of these approaches move the network interface closer to the application by removing the host processor and operating system from the communication critical path. The third approach, while less widely accepted, allows applications to directly access memory in remote applications.

1.2.1 User-mode access and zero-copy transfer

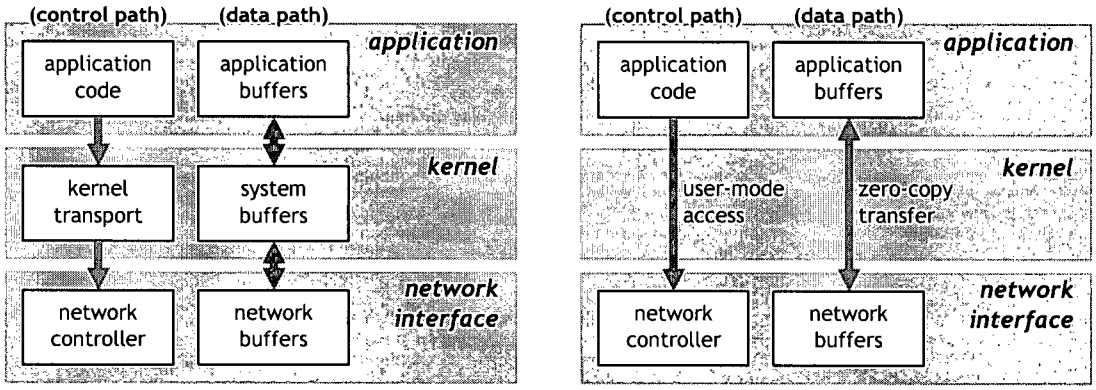

Figure 1. Traditional kernel network transport vs. user-mode access and zero-copy transfer.

Figure 1 shows the comparison between the traditional kernel-based network transport and the existing state-of-the-art user-mode access and zero-copy transfer approaches. In the traditional approach, an application sends data by invoking the operating system kernel through a system call. The kernel transfers the application data into the network interface through a series of host-processor copy operations; it first transfers the data from the application memory into the system buffer in kernel memory and then from a system buffer into the network interface. Finally, it initiates the network controller to send out the message onto the network. When a message arrives from the network, the network interface interrupts the kernel to transfer the data from the network interface into a system buffer. The kernel determines the location of the data and transfers it to final destination in application memory. It then schedules the blocked application that was waiting for the data to arrive. In both sends and receives, the kernel may need to copy the data several times among the system buffers as it shepherds the data through the different network software layers.

Zero-copy transfer and user-mode access and bypass the kernel in the data path and the control path to the network interface respectively.

Zero-copy transfer allows a network interface to transfer data directly between the application buffers and the network hardware, using *Direct Memory Access* (DMA), instead of requiring the kernel copy it among application and system buffers. This technique has two advantages. First, DMA transfers are very efficient, often operating at the bandwidth of the connecting IO bus. Second, the CPU does not need to explicitly copy data between application memory and the system buffers, or between the system buffers and network hardware. Host-processor copy operations are expensive; they interfere with and pollute the cache, they significantly increase the memory bus bandwidth by handling the data several times in host memory, and they are far less efficient than DMA transfers. Zero-copy transfer is thus essential to maximize communication throughput.

User-mode access allows an application to bypass the operating system and directly communicate with the network interface controller. Removing the control transfer through the kernel from the messaging critical path improves latency, especially for small messages. User-mode access is thus essential to minimize communication latency.

Taken together, user-mode access and zero-copy transfer significantly reduce latency and increase throughput.

1.2.2 Sender-managed communication

Apart from user-mode access and zero-copy transfer, the third approach that is equally important for many applications is adopting the *sender-managed*, or Remote DMA (RDMA), communication model. In this model, an application can directly write to, and in some cases read from, the memory of a remote application, by specifying both local and remote addresses in a data-transfer operation. The remote host processor is not involved in the transfer operation; the data is simply copied into, or out of, its memory via zero-copy DMA transfers. These remote read and write operations either augment or replace the send-receive operations found on traditional network interfaces.

In contrast to sender-managed communication, send-receive communication has more familiar semantics. In this model, the receiving application explicitly performs a receive operation for each data transfer. The sending application specifies only the local source address of the data. The completion of the receive operation indicates that the data has arrived in the destination memory. For point-to-point send-receive operations, each receive must match a corresponding send. The model is *receiver-managed* if the application specifies the destination address during

each receive operation. The alternative is that the receive operation returns to the application the address of the received data, which is determined by the communication system.

Sender-managed communication has two key advantages. The first advantage is that it allows an application to transfer data without transferring control. This separation of data and control flow substantially improves the performance for network operations that require only data flow, by allowing an application to transfer data without interrupting or involving the remote host processor [75]. In contrast, send-receive semantics typically require that the remote processor post a blocking receive operation. The arrival of the message interrupts the host processor to match the message with the receive operation and schedule the blocked application. The second advantage of sender-managed communication is that it avoids the possibility of a buffer overrun on the receiver because the sender always specifies the destination address in the receiver's address space, which only has a finite range. Therefore, there is no need to buffer data on the receiver for flow control because the network interface can always deliver every incoming message to its predetermined location. In contrast, the send-receive model requires the receiver to buffer messages that do not yet have a corresponding receive posted. Flow control between the communication parties is necessary to avoid any buffer overruns on the receiver.

1.3 Pinning for RDMA

This thesis addresses a key problem with RDMA communication, which is that the memory-addressing models used by the applications and the network interfaces are mismatched. On the one hand, applications name memory using virtual addresses. The memory-management unit (MMU) in the host processor and the VM-system page table in the operating system transparently translate virtual addresses into their corresponding physical addresses for the application. On the other hand, network interfaces name memory using physical addresses when accessing host memory using DMA. DMA is essential for good performance when sending messages larger than several hundred bytes and when receiving messages of any size. Traditionally, IO devices use DMA to only access physical memory buffers managed only by the operating system. *Application-level* DMA for zero-copy transfers breaks this tradition; the network interface can no longer act as a simple system-IO device. Instead, it must independently determine the physical memory location of the application data in host memory to support zero-copy transfers.

Application-level DMA faces two key issues. First, the network interface must translate the virtual address of an application page into its corresponding physical address before accessing the page. Second, the network interface must ensure that the address translation remains valid for

the entire duration of the DMA data transfer. Thus, the operating system cannot remove or replace its page mapping without first synchronizing with the network interface.

Previous systems that supported zero-copy transfer ensure address-translation consistency by requiring the operating system *pin* pages before handing their physical addresses to the network interface. Once a page is pinned, the operating system cannot remove it or replace it without first revoking its mapping from the network interface. To unpin a page, the operating system synchronizes with network interface to invalidate its mapping. Two traditional approaches to pinning memory are *static* pinning and *dynamic* pinning.

In the static pinning approach, applications call the operating system to statically pre-pin contiguous blocks of physical memory, typically to serve as communication buffers. Applications must restrict the source and destination addresses in an RDMA transfer only to these pre-pinned pages. Thus, static pinning is effective for applications that confine communication between relatively small and well-known portions of their virtual address spaces. However, it is less suitable for applications that require complex data-sharing patterns. Applications that share a large virtual address space or are unable to predetermine the source and destination addresses will find it impractical to pin all of its memory for communication.

Static pinning limits the usefulness of application-level DMA because it requires applications that transfer data between memory resident data structures also ensure that these data structures reside only in pinned memory. There are two problems with this requirement. First, applications may have to pin large portions of its heap and possibly stack to accommodate the required data structures. Pre-pinning large blocks of memory can significantly interfere with the operating system's ability to effectively manage memory for all applications running on the host. The operating system cannot use application-pinned pages for any other purpose until it reclaims them from the application. A single application pinning large blocks of memory or a few applications simultaneously pinning smaller blocks of memory will quickly exhaust all usable physical memory on the host. Second, application programmers face a significant burden to decide which pages must be pinned and to design network-addressable data structures that fit in pinned memory. These data structures may be scattered throughout the application virtual address space, programmers have to either locate the pages they reside in and pin them, or reorganize and coalesce them into fewer pinned memory regions. In any case, static pinning breaks the well-understood virtual memory abstraction, by forcing the programmer to view application memory in terms of the underlying physical memory resources.

Dynamic pinning is an alternative to static pinning. In this approach, the operating system pins pages when the application accesses them for communication and maintains a cache of recently pinned pages to amortize the pinning costs. Dynamically pinning source pages is relatively simple, because the operating system can always pin the required pages only when the application accesses them and guarantee that the local network interface can always locate the pages in host memory. However, poor access locality resulting in a low hit rate in the pinned-page cache will slow down the communication rate, because the overhead of invoking the system call to pin source pages appears in the critical path of message transfer. Dynamically pinning destination pages is problematic, the network interface may have insufficient time in the critical path of a message reception to interrupt the host to pin the destination pages and still guarantee that it does not drop the message. As a result, systems, such as UNet/MM [78], that dynamically pin destination pages cannot support sender-managed communication. They also require additional flow control to prevent message loss due to receiver buffer overrun.

1.3.1 RDMA without pinning

The key contribution of this thesis is the design, implementation and evaluation of the *NetVM* network interface. NetVM provides user-level access, zero-copy transfer and sender-managed communication without requiring the operating system to pin source or destination memory. An application can transfer data from any virtual address on the source node to any virtual address on the destination node, if it has obtained the appropriate access permissions. These source and destination pages are never pinned by the host operating system.

NetVM integrates with the host virtual memory system to maintain a shadow page table on the network interface. The host operating system updates the page table, using programmed-IO writes, whenever it pages data in or out of memory. The network interface stores a lock flag for each host physical page entry in the table. It activates a page's lock flag only for a brief interval during an active DMA transfer on that page, which typically lasts only a few microseconds. The operating system checks, using programmed-IO reads, for an inactive lock flag before replacing the page mapping. Thus, the network interface can prevent the operating system from replacing a page simply by locally modifying its lock flag.

NetVM's page-locking scheme is fast enough to lie in the critical path of message reception. As long as the shadow page table stores an address translation entry for the destination page, the interface can complete a zero-copy transfer on the page without involving the host processor. If the entry is not present in the table, NetVM resorts to a one-copy scheme to transfer the

data. It first transfers the page to a system bounce buffer and interrupts the host processor. The host processor then fetches the destination page into host memory and copies the data from the bounce buffer. The cost of redirecting the data through the bounce buffer is small compared to the cost of retrieving the page from the backing store. NetVM uses credit-based flow control to prevent the bounce buffer from overflowing. A sending node must possess sufficient credits for the bounce buffer on the receiving node before it can transmit any data to that node. In the common case, the bounce buffer is not used and these credits are efficiently recycled.

1.4 Synchronization with RDMA communication

In addition to data transfer, synchronization is also an important aspect of network communication. The RDMA model separates data and control transfer; it specifies only remote memory access semantics without explicitly defining any synchronization semantics. For example, the RDMA write operations only transfer data to a remote memory location. The remote application, or even the host processor, does not know when the data actually arrives in its memory. Applications require synchronization support to coordinate with each other.

Efficient synchronization is essential for concurrent applications that exhibit fine-grained interaction in a low-latency network. Synchronization schemes should exploit the underlying hardware capabilities to provide the most efficient implementations to applications. Two features influence the implementation of these schemes in NetVM.

The first feature is that NetVM provides remote memory access semantics. Therefore, synchronization schemes that apply to traditional shared memory multiprocessors also apply to NetVM. Some of these schemes can potentially offer lower latency than an equivalent host-based scheme using send-receive communication because remote memory access is significantly faster than invoking a remote host process.

The second feature is that NetVM can deploy custom synchronization primitives in the network hardware to optimize key operations in the synchronization schemes. It already uses the programmable network interface to effectively implement memory-management support for RDMA data transfers. The network-interface design constraints that apply to RDMA data transfers also apply to synchronization primitives; network interfaces have limited local memory and limited processor capability. Thus, designing primitives that do not stress these limitations is important.

1.4.1 Blocking and busy waiting

Two mechanisms for applications to detect synchronization events, such as the arrival of a control message, are *blocking* and *busy waiting*. With blocking, an application deschedules itself by calling into the operating system and blocks waiting for the event. The operating system wakes up and reschedules the application once the event occurs. Blocking, however, can add significant overheads in two ways. First, the application always incurs the overhead of the blocking system call, even if the event it is waiting for has already occurred. Second, the application incurs high response latency, as the network interface has to interrupt the kernel, which then has to wake up and schedule the blocked application. This increased latency is significant compared to a remote write and is undesirable for applications that require fine-grained synchronization. Send-receive communication systems typically use blocking, a return from the receive call implicitly synchronizes with the corresponding sent message. Many RDMA systems also implement standard send-receive operations for synchronization in addition to the RDMA-specific mechanism described in the following paragraph.

Alternatively, an application can busy wait or *spin* on a shared variable until it is updated with a specific value indicating that the synchronization event has occurred. This approach is only possible in environments that support globally addressable shared data structures, such as shared memory and RDMA systems. Traditionally, busy waiting is essential for parallel programming shared memory multiprocessors, when the expected wait time is smaller than the scheduling overhead or when keeping the processor busy waiting is acceptable. Busy waiting only on local memory, and not on remote memory, is essential for RDMA systems, because it incurs a lower latency and avoids congesting the network. Busy waiting on remote memory requires a roundtrip network transaction for each polling operation, which quickly generates hotspots and saturates the interconnection network even if just a few applications busy wait simultaneously. Furthermore, busy waiting on *cached* local memory even avoids stressing the local memory bus if the IO system supports cache-coherent DMA. An application can strike a compromise between consuming processor cycles due to busy waiting and increasing latency due to blocking. It busy waits for a brief period during fine-grained synchronization operations, but reverts to block waiting if the wait time exceeds the process-scheduling overhead [20].

1.4.2 Transfer of control

A NetVM application transfers control by *notifying* a remote application. The semantics of NetVM notifications are similar to Eventcounts by Reed et al. [67]. Every receiving application manages a set of notification objects, each named by a notification identifier and comprised of a pair of *signal* and *acknowledge* event counters. A sending application signals an event by

specifying its notification identifier in a data-transfer operation, which increments the signal counter when the data transfer completes. The receiving application detects this event by finding a signal count higher than the acknowledge count. It subsequently acknowledges the event by incrementing the acknowledge counter.

NetVM counter-based notifications have three advantages over the send-receive blocking and the busy-wait alternatives. The first advantage is that an application can selectively transfer control when transferring data. This separation of control and data flow substantially improves the performance of network operations that require only data flow [75], by allowing an application to access remote memory without involving the remote host processor. In contrast, the send-receive model requires the remote host processor to execute a receive operation, which either busy waits for the arrival of the message or block waits for a host interrupt. Because not all NetVM data transfers need to carry a notification, the system incurs the overhead to manage the event counters only when necessary. When notifications do exist, the network interface accumulates these events efficiently without any host-processor intervention, by storing and incrementing the signal counters in its local memory. The receiving application does not need to explicitly synchronize on each event. It checks the counter values only when required, without any risk of dropping an event. With send-receive messaging, synchronization is implicit with the return from a blocking receive call. The application needs to make the receive call to match every message, even if they do not carry useful data. High numbers of messages result in high overhead on the host to match these messages and to provide flow control over the receive queue.

The second advantage of NetVM notifications is that the receiving application can choose to selectively wait for a particular event. While it waits, other incoming notifying transfers with different notification identifiers continue to increment their respective signal counters and do not affect the waiting application. With send-receive blocking, the application typically has only a single queue to receive incoming messages to accept, buffer and track all messages, to prevent the receive queue from overflowing and to prevent dropping out-of-order signal events.

The final advantage of NetVM notifications is that the receiving application can choose to busy or block wait for events, or even do both. This choice allows the application to decide the appropriate tradeoffs between reducing detection latency and reducing host-processor overhead. It has four ways to detect notifications. First, it can selectively compare the signal and acknowledge counters in network interface memory, only when necessary, to determine the number of pending notifications. Second, it can direct the network interface to update a

shadow signal counter in host memory when it receives a notification. The application busy waits on this shadow counter to synchronously detect the notification with low latency at the expense of consuming processor cycles during the busy wait. Third, it can direct the network interface to interrupt the operating system each time it receives a notification. The application blocks waiting for the wakeup from the operating system without consuming the host processor, but at the expense of higher detection latency. Finally, it can register a notification-handling procedure that will execute whenever a notifying transfer completes. Thus, like UNIX signals, the application can continue processing while asynchronously handling incoming notifications.

To support notifications over an out-of-order delivery network, NetVM defines ordering semantics that relate synchronized with unsynchronized transfers. Specifically, NetVM guarantees that the receiving application only detects the notification for a current transfer after it has received all previous data transfers and notifications from the same sender. The data-transfer operations themselves can complete out of order, NetVM only enforces the invariant for the partial order between the current notification and preceding data transfers. To do this, NetVM maintains a sequence window to track incoming messages and buffer notifications but not the data. It delivers a notification to the application only after processing all preceding messages. Therefore, an application that receives a notification can safely assume that it has also received all the data that the sending application transmitted to it before issuing that notification. As a result, the receiving node avoids data buffering and reordering while it provides simple delivery-ordering semantics for applications.

1.4.3 Synchronization idioms

NetVM implements three general synchronization idioms that are useful to applications: mutual-exclusion, wait queue and semaphore. There are many alternatives to implementing them in a globally addressable memory machine. However, NetVM offers specific features, which exploit the underlying hardware, to provide a more efficient implementation. Section 1.4 described two of these features, which are remote memory access and network-hardware programmability. A third feature is also useful, which is that applications can use the notification mechanism to adaptively switch between busy waiting and blocking. Many traditional local-spin synchronization algorithms for shared memory multiprocessors require applications to busy wait for synchronization events. This approach provides lower detection latency but also consumes host-processor cycles. NetVM allows applications to do both busy and block waiting using the same underlying notification mechanism. Thus, it enables applications to attain low response latency as well as incur low processor overhead whenever appropriate.

NetVM implements a mutual-exclusion synchronization construct based on the MCS [51] distributed lock. Briefly, a central data structure in globally addressable memory stores the state of the lock, which is either available or points to the last process in a queue trying to gain the lock. An application performs read-modify-write operations on the central data structure to acquire or release the lock. Multiple applications waiting to acquire the lock form a queue with a distributed linked list. Each one busy waits on a local flag until its predecessor hands over the lock by updating that flag with a remote write. NetVM improves on the original design by allowing the waiting application to adaptively switch between busy waiting and block waiting, which reduces host-processor overhead.

To support the read-modify-write operations required by MCS locks, NetVM implements a set of standard atomic operations commonly found on multiprocessor systems. These operations are useful for implementing many synchronization algorithms that already exist for non-cache-coherent multiprocessor systems. Atomic operations are similar to read operations. The application first sends a request message to the remote node. The remote network interface responds by transferring the operand from host memory into its local memory, performing the specific atomic operation, writing the result back to host memory, and sending a reply to the requesting node. The requesting application, in the meantime, busy waits locally until it receives the reply from its network interface.

NetVM implements the wait queue and semaphore synchronization idioms with an MCS-lock-inspired approach. Like MCS locks, applications using the wait queue or semaphore atomically update a central data structure and form a distributed queue when necessary. However, the standard atomic operations are insufficient to support these idioms while reducing the number of required network transactions, because they cannot atomically update all the necessary state information in the central data structure with a single network transaction. As a result, traditional algorithms use a three-phase approach to implement these synchronization operations [21]: the first phase acquires exclusive access to the central data structure. The second phase manipulates these data structures, which may require several network transactions, and the third phase releases exclusive access. To efficiently implement these idioms, NetVM augments the network interface with synchronization primitives that optimize key synchronization operations. These extended atomic operations essentially combine the key steps in the three phases into a single network transaction. They are also lightweight and thus do not incur significant overhead over standard atomic operations. Thus, applications require fewer network transactions to perform the synchronization operations, resulting in lower operation latency, lower overhead on the network interface and improved scalability.

1.5 Protection

Communication protection mechanisms prevent applications from gaining unauthorized access to local and remote memory. These mechanisms are especially important for systems that support user-mode access and sender-managed semantics for two reasons. First, user-mode access allows applications to directly interact with the network interface. In some cases, these applications can overwrite the firmware on the network interface, or direct the on-board DMA engine to inadvertently or maliciously transfer data from system memory areas. Many research communication systems that support user-mode access into the network interface avoid the protection problem by assuming a *trusted* communicating application on the host. Second, sender-managed semantics allow applications to directly access remote memory. Without enforcing remote access permissions, applications can interfere with the memory in all other remote applications on the network.

To address these problems, NetVM provides protection at the *local access* and *remote access* levels. NetVM uses a VM-based approach to enforce local access protection. An application only obtains a small virtual address window into the network interface to communicate with the NetVM network controller. Each isolated application acquires its own separate window; it cannot access data structures that belong to the system or other applications. NetVM uses a capability-based mechanism, similar to Hamlyn [12], to enforce remote access protection. An application can access remote memory only after it has acquired the protection key for that memory. The remote network interface verifies the keys for *every* data transfer. If the underlying network is physically secure, an application that cannot obtain a valid key also cannot forge a message to access remote memory.

1.6 Research contributions

This thesis makes the following research contributions:

1. It describes the design and implementation of integrating the virtual-memory-management system with the network interface to support reliable zero-copy transfers without the need to pin host pages. This integration allows the operating system to retain full control over host memory without incurring significant overheads. The evaluation examines the costs and benefits of using this approach and shows that, compared to the static and dynamic pinning approaches, NetVM does not significantly affect operating system operations, incurs a low overhead on the network interface compared to static pinning, and performs favorably compared to dynamic pinning.

2. It describes the design and implementation of a flexible control-transfer mechanism that allows applications to selectively issue and detect notifications with a tradeoff between reducing control-transfer latency and reducing host-processor overhead. This mechanism supports ordered notifications over an out-of-order delivery network. The evaluation highlights the tradeoffs in signaling latency for three notification-detection alternatives: busy waiting, block waiting, and triggering a user-defined handler.

3. It describes the design and implementation of scalable low-latency and low-overhead wait-queue and counting-semaphore operations. This design reduces the required number of network transactions by augmenting the network interface with synchronization primitives, which are easy to implement. The evaluation demonstrates that these operations have low latency for the application and incur low overhead on the network interface.

Thus, the thesis makes the following statement:

Intelligent network interfaces provide an efficient mechanism for data transfer and synchronization. Integrating memory management with the network interface enables fast data transfer without significant overheads on the network processor, or restrictive constraints on applications or the operating system. Embedding key synchronization primitives on the network interface facilitates scalable synchronization idioms with low latency and low overhead.

1.7 Thesis organization

Chapter 2 describes the background of the thesis and the work related to it. It introduces the system-area network and user-level network interfaces, presents existing memory-management strategies and address translation techniques in these network interfaces, and looks at hardware-based synchronization alternatives for notification and mutual exclusion. Chapter 3 presents an overview of NetVM. It lists the design goals, introduces the user API, describes the system architecture and briefly examines the Myrinet programmable network interface.

Chapters 4 to 8 present a detailed design description of NetVM. Chapter 4 describes the memory-management functionality in NetVM, which is a key contribution of this thesis. Chapters 5 and 6 describe the data and control (notification) transfer operations respectively. Chapter 7 presents a user-level implementation of channels using the NetVM API. Chapter 8 examines the design of three general synchronization idioms: mutual-exclusion, wait queues, and counting semaphores.

Chapter 9 presents a detailed evaluation of a complete working prototype. It examines the costs and benefits of NetVM's memory-management approach, presents the performance results for data transfers, notifications and channels. It also analyzes the performance and overhead of NetVM's atomic and synchronization operations. Finally, Chapter 10 concludes this thesis with a brief summary of the key points. It also discusses research directions and future work in this area.

2 Background and related work

This section describes the background of the thesis and work related to it in four broad areas. The first part introduces system-area networks and user-level network interfaces. The second part reviews page-pinning strategies in user-level communication systems. The third part summarizes address-translation mechanisms in network interfaces. Finally, the fourth part describes the synchronization and mutual-exclusion alternatives for non-cache-coherent global address space and shared memory architectures.

2.1 System-area networks and user-level network interfaces

New applications demand increasingly high bandwidth and low latency communication. These applications, such as databases, parallel simulations and backend servers, access large amounts of data and require significant compute capability. The uniprocessor machine is increasingly unable to meet these demanding requirements in terms of computation, memory and IO performance. Parallel computers can help by distributing the workload over multiple machines. Two common types of parallel architectures have emerged: the *Symmetric Multiprocessor* (SMP) and the *computer cluster* [65]. The SMP extends the traditional uniprocessor architecture with multiple processors on the single system bus, thus improving the total compute capability of a single machine. However, this approach is not scalable as only a limited number of processors can share the system bus. The computer cluster addresses this limitation by linking multiple uniprocessor or SMP machines together with a high-bandwidth and low-latency network, which is the *System Area Network* (SAN) [7].

Traditional local area networks (LANs) cannot the provide communication performance required by these demanding applications. LANs have significantly higher latency and lower bandwidth compared to an SMP system bus. SANs bridge this performance gap between LANs and system memory buses; they deliver lower latencies and higher bandwidths that are several orders of magnitudes better than LANs. Examples of SANs include the Myrinet [10], Virtual Interface Architecture (VIA) [17], Infiniband [40], Meiko Computing Surface (CS-2) [14, 38, 49],

Quadrics QNet [63, 64], DEC Memory Channel (MC and MC2) [28, 31], Scalable Coherent Interface (SCI) [34, 35] and the IBM SP2 [72].

A key benefit of SANs is that they can achieve very low communication latencies. Application-to-application latencies can be several microseconds with bandwidths as high as the connecting IO bus. Low latency is essential to applications that require fine-grained communication; request-response protocols, database queries and scalable group synchronization all communicate using small messages and are thus sensitive to the performance of low-latency fine-grained messaging. For examples, Kay et al. [42] suggest that the majority of NFS traffic consists of small messages less than 200 bytes. DAFS [19] relies on the underlying low-latency VIA [25] network to efficiently access client file buffers with RDMA transfers and achieve good performance for network storage operations. Network-bound applications that stall waiting for remote responses, exchange small amounts of data at runtime, or rely heavily on group synchronization, benefit most from improvements provided by SANs.

Deploying a SAN alone is insufficient for applications to achieve good communication performance. Traditional kernel-based network transports do not take advantage of the new SAN benefits, because they are designed to provide protected communication over unreliable LANs. Consequently, applications suffer from poor performance when they rely on the kernel for communication. User-level network interfaces allow applications to bypass the kernel in the control and data path to the network by directly accessing the network interface for communication. Lightweight communication protocols support these applications by bringing the application closer to the network.

Three factors dominate communication latency on a LAN or SAN: the protocol control path through the communication system, the data-transfer path between application memory and network interface, and the latency of the network hardware. Traditional heavyweight protocols, such as TCP/IP, that were previously suitable for LANs are now no longer suitable for SANs due to their excessive software overheads in two areas. First, legacy protocols typically assume that LANs are unreliable and can therefore arbitrarily delay, drop or corrupt packets. This assumption results in complex software protocols that incur high overheads to handle these data transmission contingencies. Second, legacy protocols route all data through the operating system. Context switching between the application and the operating system [5] for fine-grained communication or shepherding data through the transport stack for large data messages is expensive. This situation is worsening because operating system improvements cannot keep up with host-processor improvements [58].

Lightweight communication protocols using user-level network interfaces in a SAN environment eliminate the overheads described in the preceding paragraph. SANs offer a good compromise between LANs, which provide good scalability and flexibility, and memory buses, which provide performance and reliability. They are reusable and configurable in different systems and environments, they scale to very large cluster networks, they are sufficiently fast for applications that require fine-grained communication and they are sufficiently reliable to require only minimal error detection and correction. User-level network interfaces allow an application to directly access the network hardware without relying on the operating system. Bypassing the operating system in the communication control and data paths eliminates traditional software protocol overhead in the kernel transport. For examples, the Thinking Machines Corporation Connection Machine 5 (CM-5) allows protected user-mode access to the underlying network hardware by mapping the interface registers into the application address space. The DEC Memory Channel maps the global address space managed by the network interface into the application address space, thus allowing the application to access global memory simply by reading from or writing to the mapped region. VIA and many research-oriented communication protocols on programmable network interfaces allow applications to interact directly with the network interface firmware to initiate a data-transfer operation. Because SAN interconnects offer better performance and reliability, and because user-level network interfaces offer more efficient and direct access to the hardware, lightweight communication protocols on SAN-based user-level networks are simpler and more efficient than traditional kernel-based protocols. Thus, they offer good communication performance with little software overhead. Mukherjee et al. [52] and Bhoedjang et al. [8] provide a general survey of design issues for user-level network interfaces.

Two alternative mechanisms for communication systems to transfer data between application host memory and the network interface memory are *Programmed IO* (PIO) and *Direct Memory Access* (DMA). With Programmed IO, the host processor individually transfers each word between host memory and the network interface. With DMA, the host or NI processor initiates the data transfer between host memory and the network interface, usually in a series of transfer bursts. Once initiated, the DMA engine operates autonomously until the entire transfer completes. Thus, DMA is the preferred alternative for large data transfers. For small data transfers, the overhead of setting up the DMA engine registers typically exceeds the latency to copy the data through the host processor. Thus, programmed IO is the preferred alternative for small data transfers.

Communication systems often use a combination of programmed IO and DMA. For systems without bidirectional DMA support, such as MC, or systems that do not support application-level

send-side DMA, such as FM, programmed IO is the only alternative to moving data into the network interface. Programmed IO is also preferred for group operations, because they are typically latency sensitive and require only small messages. For host-to-NI transfers, many communication systems, such as VNTP [24], Hamlyn [12], BIP [66] and NetVM, adaptively switch between programmed IO for small transfers and DMA for large transfers. For NI-to-host data transfers, DMA is the only alternative to transfer data without involving the host processor. Most RDMA systems use DMA transfers on the receiving node regardless of transfer size. Only a few systems, such as AM-II, use programmed IO to receive small messages.

NetVM is a user-level zero-copy communication system for the Myrinet SAN. Applications directly interact with and share access to the network interface. All data transfers bypass the kernel whenever possible. NetVM uses programmed IO for small outgoing transfers but adaptively switches to zero-copy DMA for larger transfers. NetVM always uses DMA on the receiving side. In the common case, the network interface delivers data directly to the application buffer with zero-copy transfers. However, NetVM switches to a combination of DMA and host-buffer copying by the kernel when the application buffer is not directly accessible by the network interface.

2.2 DMA-registration strategies

A key contribution of NetVM is the design of an NI-based page-locking mechanism that avoids the need for the operating system to pin pages for DMA transfers. This section describes other page-pinning alternatives used in user-level communication systems. Application-level DMA faces two key requirements. First, the communication system has to provide the physical addresses of all pages serving as communication buffers to the DMA engine. Second, the system has to ensure that the operating system does not replace a page in host memory during a DMA transfer on that page. To do this, systems typically pin pages and hand their physical addresses to the network interface.

As described in Section 1.3, communication systems either *statically* pin or *dynamically* pin pages for DMA transfers. With static pinning, the system statically pre-pins all communication buffers, usually for the entire duration of the application (an application may also explicitly pin and unpin buffers for each major phase in its execution). With dynamic pinning, the system manages a dynamic set of pinned pages, which it pins and unpins as necessary throughout the entire duration of the application. Table 1 on the following page shows the variations in each of the two page-pinning alternatives and the NetVM NI-based page-locking approach.

Pinning	Strategy	Examples
Static	system buffers user buffers	FM BIP, GM, Hamlyn, Trapeze, VIA, SCI, Infiniband, VMMC-1 DVMA, FLASH, ELAN (CS-2 and QNet), MC, MC2
Dynamic	on-demand pinned-page cache	BIP, GM, Panda Firehose, Pin-down Cache (PM), VNTP, U-Net/MM, VMMC-1, UTLB (VMMC-2)
NI-based Locking	per-page locking	NetVM

Table 1. Key DMA-registration strategies.

2.2.1 Static Pinning

A system that uses static pinning can either pin *system buffers* or pin *user buffers*. Pinning system buffers is the simpler, but less flexible, approach. Pinning user buffers allows true zero-copy transfers, but it also requires additional support for the network interface to directly access the pinned user pages.

2.2.1.1 Statically pinning system buffers

In this approach, the communication system pre-allocates pinned buffers in kernel memory to serve as host-DMA staging areas for data transfers. This approach is akin to the traditional kernel-based transport-stack where all network-bound data must transit through unpageable kernel buffers for DMA transfer. The key difference is that, in this case, the operating system maps the staging area into the application address space, thus exposing the system buffer to the application and allowing it to directly access this pinned staging area. To transmit data into the network, the application has to copy the data from its original location into this system buffer before initiating the DMA transfer from the pinned area. To receive data from the network, the application has to copy the data from the system buffer to the final destination, or consume it in place, after the network interface has completed the DMA transfer to the pinned area.

FM [59-61] uses the system buffer to store the receive queue for incoming messages to support stream-oriented active-message-style [26] communication. When a message arrives from the network, FM executes a function handler specified in the message header, which may copy the data from the system buffer to the destination data structures in application memory. Thus, FM does not support zero-copy receive operations. FM does not rely on a pinned system buffer for sending messages. Instead, it splits messages into 128-byte packets and transfers each fragment directly into the network interface using programmed IO. For this small fragment size, using programmed-IO writes is faster that using a two-step process of first copying the data to the system buffer and then transferring it by DMA to the network interface. It also allows FM to

better pipeline these fragments through the network interface. FM does not rely on programmed IO for receiving messages, because uncached programmed-IO reads are significantly slower than programmed-IO writes. Furthermore, the network interface is able to autonomously transfer data to the receive queue without any host-processor intervention. To prevent the receive queue from overflowing, FM uses a return-to-sender protocol to reflect the message back to the sending network interface if the queue is full. The sending network interface has to reserve sufficient space, enough to accommodate all unacknowledged messages, in its local memory to handle this contingency.

There are three key problems with using this intermediate pinned system-buffer approach. First, zero-copy transfer is not possible; all DMA-transferred data must pass through the system buffer. Second, the remote network interface typically has to interrupt the host processor to copy the data from the pinned buffer to the final destination and signal the application, thus increasing the latency and host-processor overhead [15]. Third, the communication system usually supports only a single trusted application, because the operating system has to expose the system buffer to the application. Thus, sharing the buffer among multiple applications with protection is difficult. VNTP [15, 47] allocates a dedicated, but pageable, system buffer for each communicating application.

2.2.1.2 Statically pinning user buffers

In this approach, the application statically allocates pinned user memory to serve as communication buffers. Many user-level communication systems belong to this category. They usually provide a special memory allocator that assigns typically contiguous physical memory to the application. The application must register all locally and remotely accessible memory segments before initiating any DMA transfer with them. The network interface directly accesses these registered memory segments with zero-copy transfers.

Applications must register communication buffers before performing any zero-copy data transfer on them. The application typically registers the memory during program initialization or before the first API call that accesses that memory. Some systems, such as BIP, are slightly more dynamic and maintain a list of pinned buffers in the communication library. When a BIP application transmits a buffer that is not currently pinned, the library pins the buffer and leaves it pinned for the remainder of the program execution. Static pinning is desirable if the working set of the communicating application can fit within available host memory. Once pages are pinned, the operating system cannot remove or replace them without first synchronizing with the network interface, usually when the program terminates.

Sender-managed communication systems, such as Hamlyn [12], VIA, Infiniband and SCI, maintain a translation table on the network interface to map each registered memory segment to their corresponding physical addresses. Any incoming network message that can name the memory segment can also access it from the network interface. Trapeze [3, 80] maintains an incoming payload table to store the physical addresses for reply buffers used by the page-based request-reply protocol in GMS [27]. VMMC-1 [9, 24] also stores a per-application translation table for all registered memory segments that the application exports to remote processes.

There are two key problems with static pinning for user buffers. First, allocating per-application pinned memory is not scalable; too many applications requesting pinned memory will quickly exhaust the available physical memory on the host. Consequently, applications using this approach can only pin a limited amount of memory each. Second, communication is restricted to the pinned buffers. Applications must allocate network-addressable data structures only within these buffers. The network interface usually drops any message directed to an unpinned page because it does not have the physical address of that page. Some systems prevent this message drop. VMMC-2 redirects any data that does not yet have a receiving address into a *default* buffer. When the application eventually posts the receive operation, VMMC-2 immediately copies the data from the default buffer into the final posted receive address. NetVM redirects the data to its bounce buffer and directs the operating system to fetch the destination page from the backing store and copy the data to that page.

2.2.2 Dynamic Pinning

A system that uses dynamic pinning manages a dynamic set of pinned pages. The application, or the communication system, pins and unpins pages as necessary to limit the total number of pinned pages in host memory. There are two alternatives for dynamic pinning: *on-demand* pinning and *pinned-page caching*.

2.2.2.1 On-demand pinning

On-demand pinning only pins pages when the application is ready to transmit or receive the data. This approach is usually only suitable for large messages because the pinning costs on both nodes amortize over the large amount of data to transfer. In the two-step *rendezvous* [6] protocol, the requesting application first handshakes with the remote application to indicate the set of buffers it has pinned or it needs the remote application to pin. For remote writes, the remote application pins the requested buffer and responds to the sending application, which completes the operation by transmitting the data. Remote writes, therefore, require a

roundtrip network transaction that involves both host processors for the rendezvous protocol just to pin the target buffer. As a latency optimization, the requesting application can concurrently pin its local source buffer while waiting for the rendezvous reply. For remote reads, the remote application pins its source buffers and responds with the requested data to the sending application, which must have already pinned the target pages. In both cases, this pinning overhead is always in the critical path of the data transfer.

On-demand pinning can be lazy. After a rendezvous, applications may continue to access the pinned buffers and thus delay the unpinning step. Unpinning is problematic, because it requires another rendezvous for both applications to synchronize and invalidate the set of pinned buffers. Therefore, some systems, such as GM [55], avoid unpinning buffers until the application terminates, which essentially suffers the disadvantages of the static pinning approach. As a compromise to optimize bandwidth, communication systems may use a one-copy scheme with a small statically pinned buffer for small messages and a zero-copy scheme with rendezvous for large messages. In this case, small data transfers, although not zero copy, incur relatively little host-processor overhead to copy the small data, and large transfers, although requiring a roundtrip rendezvous, complete with zero-copy DMA transfers. Nieplocha et al. [56] show that this tradeoff is critically dependent on the relative overheads of the network hardware and operating system support.

2.2.2.2 Pinned-page caching

Pinned-page caching manages a dynamic set of pinned pages for the application. The application or system library pins buffers when they are required in a DMA transfer, either transparently when the application or network interface accesses them, or explicitly when the application anticipates receiving data on them. In order to limit the total number of pages pinned by an application on the host, the cache-replacement policy unpins and ejects the least valuable pages from the cache whenever it needs to pin and add new pages into a full cache. Pinned-page caching, like all other caches, only works well if there is locality in the accesses. Depending on the application workload and access pattern, a poor hit rate in the pinned-page cache results in a high overhead for pinning and unpinning pages.

Pinned-page caching for sending data is straightforward. Whenever an application attempts to send data from buffers that are not pinned, the system pins them, after unpinning other buffers if necessary, and obtains their corresponding physical addresses. The PM user library uses the pin-down cache [74] to manage the set of pinned pages. An application sending a message from an unpinned buffer calls the PM kernel module to pin the page and hand its physical ad-

dress to the network interface. The kernel module maintains the cache of pinned pages and defers application requests to unpin pages if possible. Doing so reduces the overhead if the application repeatedly requests to pin and unpin the same set of pages. However, the application always incurs the system-call overhead in the critical path of the send operation.

Firehose [6] maintains data structures to track the locally and remotely pinned pages. When the application attempts to access a remote page that is not pinned, it rendezvous with the remote application to pin the target page and stores the mapping in a remote-page table. Firehose holds a reference to the remote page and caches the page mapping to amortize the synchronization and pinning costs for frequently accessed pages. In order to manage the total number of pinned local and remote pages, Firehose applications synchronize with each other to remove their page references and unpin the least recently accessed pinned pages.

The network interface for VMMC-1 maintains an outgoing page table to store the mappings of all pinned pages on the host. If it cannot locate the mapping for a page to access via DMA, it interrupts the host kernel to pin the page and restart the transfer. The overhead of the host-processor interrupt on the sending node is not detrimental, because it simply delays the DMA transfer of the data from host memory into the network interface without the possibility of losing any data. VNTP shares multiple virtual endpoints, each belonging to an application, on a single network interface. When a local application touches any registered segment belonging to an endpoint that is not resident, the page-fault handler in VNTP fetches the endpoint into the network interface, including the physical addresses of all the pinned pages belonging to the endpoint. Similar to host-processor interrupt overhead in VMMC-1, this endpoint-fetching overhead is also not detrimental on the sending node. It is, however, detrimental on the receiving node. Thus, VNTP drops the message whenever the receiving node is too late in paging in the endpoint and relies on a retransmission mechanism to ensure that the sending network interface retries the operation.

There are three general alternatives for a receiving network interface to recover when it does not know the physical address of the target buffer. The first alternative is to *interrupt* the host processor for the operating system to fetch the buffer, pin it and provide the network interface with its physical address. The second alternative is to either *drop* the message or reflect it back to the sender, and then rely on a higher-level flow-control protocol to resend the data. The third alternative is to *redirect* the data into a temporary pinned buffer on the receiving host if the target page is not pinned. The operating system then copies the data to the final destination.

UNet/MM uses the first alternative described in the preceding paragraph. It maintains a pre-translated pinned free buffer queue for each application. When a message arrives from the network, the network interface transfers the data to the first buffer and informs the application of the address of the buffer. If the free buffer queue is empty, the network interface interrupts the kernel to pin more pages for the queue, but will drop the message if the kernel cannot respond in time. UNet/MM, like FM and VNTP, does not support receiver-managed communication, because the receiving application cannot control the destination address of a message. Instead, it obtains the address from the communication system and then copies the data to its final destination.

Pinning pages in the messaging critical path is difficult. The key problem is that the overhead to interrupt the host processor to pin the page, possibly after fetching it from the backing store, and to install the page mapping into the network interface is too high to guarantee that the network interface will not drop the message due to insufficient local memory. The limited memory available on the network interface cannot sustain large delays to the data-transfer pipeline, especially if the target page is not resident. For example, incoming data at 1-Gb/s consumes 8-MB of network-interface memory in only 6.3ms, which is less than the page-in time even for a single page and negligible compared to the time for the worst-case scenario to fetch 8MB of data, or 2048 4-KB pages, from the backing store. Pinning *resident* pages, however, requires a much lower time (on the order of microseconds per page), because the page mapping already exists in the host VM page table. Therefore, it may be possible to pin resident pages in the critical path and provide the translation to the network interface in time. However, this minimum time is not guaranteed, because other host interrupts and operating system activity may delay this time-critical pinning operation sufficiently to force the network interface, such as in UNet/MM, to abort waiting for the translation and drop the message.

The network interfaces in PM and VNTP drop the message if they cannot locate the target page in host memory. The receiving application is responsible for pinning the pages before the message arrives from the network. VNTP sends a NAK to the sending network interface if the receiving endpoint is currently paged out, PM sends a NAK if the target page is not pinned. Thus, they both require a messaging flow-control mechanism to handle data retransmissions.

VMMC-2 redirects the message to a *default* buffer if the network interface cannot locate the target page in host memory. Its UTLB [13] mechanism maintains a per-process translation table in host memory, which the network interface can directly access and cache. The application pins the target pages on demand and updates the kernel-protected table in host memory. The network interface redirects the data into the default buffer if it receives a message without a

matching posted receive. When the application finally posts the receive operation, the user library copies the redirected data from the default buffer to the target address. A credit-based flow-control scheme prevents the default buffer from overflowing. If the network interface receives a message with a matching posted receive, it looks up its cached translation table to locate and transfer the data directly to the target page in host memory via DMA. The lookup may result in a cache miss. In this case, the network interface fetches the updated mappings, via DMA, from the kernel-protected table in host memory, which the application *must* have previously updated by pinning the pages before posting the receive operation. The fixed cache-miss-handling time to fetch the mapping is small and thus the network interface will never need to drop the message.

2.2.3 NetVM per-page locking

NetVM does not need to statically or dynamically pin application buffers. Instead, it relies on the host VM system to track all resident pages that the applications export for communication. The host kernel uses programmed-IO to update a shadow page table, which is stored on the network interface, whenever it maps or unmaps an exported page in host memory. As long as the required page is resident, the network interface simply sets its lock bit in the shadow page table before accessing it with a DMA transfer. This brief locking operation incurs a low overhead on the network interface. To ensure translation consistency, the host kernel synchronizes with the network interface when it unmaps a page from the shadow page table.

The VM system in NetVM performs a similar function to the cache controller in the pinned-page cache approach. In this case, it treats the entire host memory as one large cache. The network interface can access any exported page as long as it is resident in host memory. This situation is equivalent to a hit in the pinned-page cache, except that NetVM does not require any resident pages to be pinned, which is necessary in the pinned-page cache, and the VM system can thus replace them at any time. Because the VM system naturally retains pages that are frequently accessed locally or remotely in host memory, NetVM applications do not need to explicitly manage pages for communication.

The network interface relies on a one-copy scheme if the target page is not resident. In this case, the network interface avoids dropping the message by redirecting the message into a shared pinned system buffer and interrupting the kernel. The kernel fetches the required page and copies the data from the system buffer to its final destination. NetVM implements a credit-based flow-control protocol to prevent the shared system buffer from overflowing.

2.3 Address-translation mechanisms

Address Translation	Description
physical address	application/kernel hands physical address to NI
segment table	NI maintains a segment translation table
translation lookaside buffer (TLB)	NI maintains a TLB data structure

Table 2. Three general address naming and translation mechanisms.

For the network interface to access application pages in host memory using DMA, it must translate the name of a user buffer into its corresponding physical address. This name can be as simple as the actual physical address of the buffer, or in the form of a named segment identifying the buffer, or even be the virtual address of the buffer in the application address space. Table 2 summarizes the three general address-translation alternatives.

The first alternative is for the application to hand the physical addresses over to the network interface after obtaining the mapping from the operating system. BIP, FM and LFC applications call a kernel driver to pin the requested pages and return the corresponding physical addresses, which applications pass to the network interface for DMA transfers. This approach, although fast and simple, is only suitable for trusted applications, because the network interface usually cannot verify the physical addresses that it received from the applications.

The second alternative is for the network interface to store a translation table to map a named region of host memory to its corresponding physical address. Many communication systems, such as Hamlyn, VIA, SCI, GM, Trapeze and VMMC-1, use this approach. The named region may or may not be physically contiguous in host memory. The key difference is that, for physically contiguous regions, the network interface stores only the base physical address and size of the region. Otherwise, it has to store a mapping entry for each page in the named region. In either case, *every* page in the named region must be pinned in host memory and mapped into the network interface.

The third alternative is for the network interface to store a translation-lookaside buffer (TLB) data structure, which maps an application's virtual addresses into their corresponding physical addresses. Network coprocessors, such as the Stanford FLASH [43], Sun DVMA [73], ELAN (Meiko CS-2 and Quadrics QNet) [50] implement the TLB in hardware. Communications systems using programmable network interfaces, such as UTLB (VMMC-2), UNet/MM and Pin-down Cache (PM), implement the TLB in software. In either case, the TLB cannot store the mapping of every virtual page belonging to every communicating application, therefore these systems cache the

mappings and require a miss-handling mechanism to lookup a mapping and update the TLB if the network interface cannot locate the required entry in the TLB.

NetVM maintains a shadow page table on the network interface to store page mappings for resident pages that applications export. Unlike the TLB approach, NetVM never suffers from capacity misses because the shadow page table is large enough to accommodate the mapping of every *physical*, not virtual, exported page on the host. However, it does suffer rarely from conflict misses because individual buckets in the page table, which is a hash table, can overflow when inserting a new page mapping. Finally, compulsory misses occur with the required page is not resident in host memory. Thus, NetVM handles these misses by redirecting the transfer to the kernel without dropping the message.

Miss Handling	Description
interrupt	interrupt host to install page mapping (UNet/MM, FLASH, DVMA)
fetch entries from host	fetch page mapping from host data structure (UTLB)
drop message	abort the transfer (PM, VNTP)

Table 3. Miss handling on NI-based page tables.

Table 3 shows the three general alternatives for the network interface to handle TLB misses. In the first alternative, the network interface interrupts the host operating system to install the mapping into the TLB. The operating system looks up its page table to locate the physical address of the required page to update the TLB. A problem with this alternative is that the high latency required to interrupt the host processor and update the TLB may cause the network interface to drop the message if it cannot obtain the translation in time.

NetVM uses a variation of the first alternative. If the network interface cannot locate the translation in its shadow page table, it redirects the transfer to a system bounce buffer and interrupts the kernel. The kernel first fetches the required page and installs the page mapping into the shadow page table. Thus, the network interface will find the translation for the same page in subsequent lookups. The kernel also completes the transfer by copying the data from the bounce buffer to the final destination. The bounce buffer ensures that NetVM does not need to drop the message due to translation miss on the shadow page table.

In the second alternative, the network interface autonomously fetches the updated entry from a pre-defined page-table data structure in host memory. This alternative requires that the operating system and the network interface agree on the location and data structure format of

the reference page table in host memory. It also requires that the system has currently pinned *all* pages mapped by the reference page table.

In the third alternative, the network interface simply drops the message if it cannot find the mapping in the TLB. UNet/MM and BIP also drop the message if the receiving application is too late in replenishing the free queue or posting the receive operation and providing a target physical address to the network interface respectively. VNTP drops the message if the destination endpoint is not resident.

Schoinas et al. [69] describe different address-translation mechanisms based on hardware-based and software-based approaches.

2.4 Synchronization

Synchronization between concurrent threads is a basic problem in multithreaded programming. Since Dijkstra's 1968 paper on concurrent programming [22], only a small number of basic synchronization mechanisms has been recognized as fundamentally essential to multithreading. They include mutexes [79], condition variables [36], monitors [37], semaphores [23], eventcounts [67], barriers [46] and rendezvous [2]. Each mechanism is suited for a particular synchronization pattern; no one mechanism provides the ideal solution for all patterns. The tradeoff between them is usually the level of abstraction that they provide and the efficiency with which they are implemented.

2.4.1 Notification

A basic aspect of synchronization is notification. An application notifies a remote application by signaling an event that the remote application detects. NetVM's notification mechanism is similar to eventcounts and monotonic counters [76]. An eventcount is an object that keeps a count of the number of events in a particular class that have occurred during program execution (for example, in producer-consumer problems). It has a monotonically increasing integer variable with three operations on the variable. **advance**(eventcount) signals the occurrence of an event associated with a particular eventcount, by atomically increasing the variable by one. Thus, the eventcount variable tracks the total number of **advance** operations performed on it. **await**(eventcount,wakeup) and **read**(eventcount) respectively blocks on and polls the eventcount variable. **await** blocks until the eventcount variable is greater than or equal to the specified *wakeup* argument. **read** simply returns the current value of the eventcount variable. In NetVM, applications signal an event by including a notification number, which specifies the

event, in a remote write operation. Only the receiving application that owns the notification counters can poll, wait for, or handle the incoming event. Thus, the NetVM notification mechanism is limited to between a set of senders and a *single* receiver. The semantics of eventcount and monotonic counters do not impose this single-receiver restriction. Thornley et al. [76] describe monotonic counters as the mechanism to support a range of synchronization patterns, including barriers, mutual-exclusion and single-writer–multiple-reader problems. In each pattern, participating processes wait on, and signal, monotonic counters to synchronize with each other.

In a shared memory multiprocessor environment, an application detects events either by busy waiting or by block waiting for the event to occur. On the one hand, busy waiting achieves lower latency at the cost of increased processor degradation [4] because all busy-waiting processes consume host processor time while waiting for the event. On the other hand, block waiting allows other processes to perform useful work while waiting for the event at the cost of increased latency to schedule the process when the event occurs. The usual compromise is to busy wait for the period equal to the scheduling overhead before falling back to block waiting [20]. The polling watchdog [48] mechanism combines polling and interrupts for efficient message handling.

The IVY [44, 45] and IVY II DSM (Distributed Shared Memory) systems use eventcounts for synchronization, which the underlying Aegis operating system natively supports. The original IVY system accessed eventcounts using remote procedure calls; incrementing and checking an eventcount required a blocking request-response operation directed to the kernel managing the eventcount to update the variable in its local memory. IVY II switched to a shared memory update approach. In this case, processes can busy wait on a cached eventcount variable for a signaling process to increment it. Li showed that this newer alternative provides cleaner semantics and is more efficient than the RPC version, especially if there are multiple processes spinning on the eventcount variable that is cached on the same node.

The Tripwire [68] synchronization mechanism allows applications to detect when a remote application accesses its local memory. Each tripwire is associated with a local or remote memory location and fires when that location is read from, or written to, by the network interface, which monitors all incoming and outgoing packets. When a tripwire is activated, the network interface can either interrupt the host processor to invoke the device driver, update a bitmap in host memory, or increment an event counter in host memory. Thus, this mechanism allows applications to busy wait or block wait for tripwire activations.

2.4.2 Mutual exclusion

Mutual-exclusion algorithms allow concurrent asynchronous processes to resolve conflicting accesses to shared resources. In the mutual exclusion problem, a process accesses a shared resource only by executing a *critical section* of code. Critical sections are exclusive; at most only one process executes its critical section at any time.

Mutual exclusion in send-receive messaging systems requires that applications send a request message to server processes running on the host. These servers may be centralized or distributed throughout the network. In both cases, a request message typically interrupts the host processor, which schedules the server process to handle the message. In contrast, mutual-exclusion operations in globally addressable memory systems use only remote memory operations on shared data structures. These individual remote memory operations are usually significantly faster than the request-reply operations. Many algorithms that apply to non-cache-coherent DSMs also apply to RDMA systems; RDMA essentially provides non-cache-coherent global addressable memory with the weak-ordering [1] memory consistency model.

Using mutual-exclusion algorithms that busy wait locally is essential for non-cache-coherent shared memory architectures. In these algorithms, processes busy wait only on local variables without causing an interconnect traversal. Remote processes may share synchronization variables on cache-coherent machines, because these machines allow all processes to spin on the local cached copy of the variable while ensuring consistency during an update. However, spinning on shared synchronization variables on non-cache-coherent machines is not practical, because each access generates a network transaction. Thus, spinning will quickly generate hot-spots and saturate the network. Craig [18] characterized various spinlock schemes based on five traits (memory architecture, atomic instruction set, operation order, interconnect load, and data structure size) and compared various spinlock algorithms using these traits.

A classic paper by Crummey et al. [51] concluded that software synchronization algorithms can exploit the memory hierarchy in shared memory multiprocessors to implement scalable synchronization operations, without the need for specialized synchronization hardware support. Their MCS lock mechanism, which is the basis for the NetVM distributed lock, executes in O(1) network transactions for both acquire and release operations, requires O(1) storage space per application, and supports FIFO ordering of requests. Motivated from this work, other researchers have investigated other techniques to reduce the number of remote memory accesses needed to implement synchronization primitives on multiprocessor systems [30, 39].

Specialized hardware support for synchronization is an active topic of research especially in the area of interconnection networks. On the one hand, the key concern with using hardware-based synchronization is the high implementation cost and thus state-of-the-art shared memory multiprocessor systems tend to support only standard atomic operation primitives and implement the synchronization operations in software. On the other hand, using hardware-based synchronization can significantly reduce the latency for these time-sensitive operations. Furthermore, implementing these operations on programmable network hardware comes at a much lower cost than on host processors or fixed-function network coprocessors.

The Queue on Lock Bit (QOLB) mechanism [32] is a hardware version of a list-based queuing lock with local busy waiting. The hardware customization allows the processor that holds the lock to transfer it to another processor with only a single cache-to-cache transfer. As a result, QOLB requires an absolute minimum number of remote messages for an acquire-release pair of operations. However, it also has a high implementation cost and complexity, because it requires significant modifications to the processor's instruction set, cache controller and cache-coherence protocol [41].

Bradford et al. [11] presented a hardware-based semaphore synchronization scheme. Their design added two additional semaphore instructions and a hardware semaphore unit to the processor. The semaphore unit maintains the status of blocked threads waiting on a semaphore. On a semaphore P operation, the processor decrements the semaphore value and blocks the requesting thread if the result is negative, by storing the semaphore address and the necessary information about the thread in the semaphore unit. On a semaphore V operation, the processor checks the semaphore unit for any blocked threads on the requested semaphore and unblocks the first thread blocked on it. Using the hardware-based processor modifications and semaphore unit significantly reduce the overhead for thread management. However, a semaphore unit can only support a single semaphore and scaling this approach to support a large number of semaphores is expensive.

Nikolopoulos et al. [57] described a methodology to implement fast synchronization operations on multiprocessor machines that support both cache-coherent and non-cache-coherent atomic read-modify-write operations. They observed that efficient software synchronization primitives fail to scale well on cache-coherent multiprocessors due to high latency for atomic operations on cached memory in the critical path of the synchronization operation. These atomic operations interfere with the directory-based cache-coherence protocols and place undue overheads on the network, especially under heavy contention for the synchronization variables. They presented a systematic methodology for transforming any software synchronization primitive into

a hybrid one that uses both cached and uncached atomic operations. Thus, it exploits uncached accesses to reduce latency during the arbitration phase, when many processors try to modify the synchronization variable, and exploits cached access to reduce the network traffic with fewer remote memory accesses.

NetVM supports fine-grained synchronization operations by implementing a standard and extended set of atomic operations on the programmable network interface. This approach does not require any hardware modification to the host, but only to the firmware on the network interface. Programmability provides the opportunity to implement and evaluate different synchronization schemes with little turnaround overhead. In this case, NetVM implements the common set of atomic operation primitives to support traditional higher-level synchronization operations. In addition, NetVM also implements new atomic operation primitives that directly support wait queues and counting semaphores. Unlike QOLB and hardware semaphore units, NetVM's approach requires only firmware changes to the network interface.

2.5 Summary

User-level network interfaces and system-area networks are increasingly essential to supporting demanding applications that require high-performance communication. Two traditional memory-management techniques to support zero-copy DMA transfers are static and dynamic pinning. NetVM introduced a new page-locking approach that does not require pinning. To enable DMA transfers, the network interface also has to translate the address of the user buffer into its physical address. Existing techniques typically name a buffer either by its physical address, by its segment name, or by its virtual address. The first two techniques are straightforward; the third requires a page table or TLB on the network interface. NetVM implements a shadow page table to store the physical page mappings for all exported pages that are resident on the host.

Synchronization is a basic problem in concurrent programming. Applications typically detect events either by busy waiting or by block waiting. The tradeoff between the two is a lower response latency for busy waiting against a lower host-processor overhead for block waiting. NetVM's notification mechanism is based on eventcounts. Non-cache-coherent remote memory systems typically implement synchronization operations using local-spin algorithms implemented with atomic read-modify-write operations. Some systems augment the host processor, cache or memory controller to optimize for these fine-grained operations. NetVM implements new atomic operation primitives on the programmable network interface to directly support wait queues and counting semaphores.

3 Overview

This chapter provides an overview of NetVM. It first lists the major design goals in four areas: data transfer, control transfer, synchronization and protection. It then summarizes the key operations available to applications. It also describes a high-level view of the system architecture. Finally, it introduces the Myrinet programmable network interface.

3.1 Design goals

NetVM has nine key design objectives in four areas.

3.1.1 Data-transfer objectives

- *Write to, and read from, virtually-addressed memory.* An application can name unpinned source and destination memory using virtual addresses or using named segments imported from remote memories. To directly evaluate NetVM's page-locking approach against a static pinning approach, NetVM also provides an alternative for applications to name pinned memory using physical addresses.
- *Guarantee reliable delivery.* Assuming that the underlying network hardware is reliable, NetVM guarantees reliable data transfer without data buffering for retransmission. This guarantee holds even if the source or destination memory is not resident.

3.1.2 Control-transfer objectives

- *Optionally notify a remote application.* Using RDMA writes means that a receiver will never know when a sender updates its memory. NetVM provides a notification mechanism for a sender to signal a control transfer when delivering the data.
- *Selectively detect notifications.* An application can selectively detect notifications only when necessary and can detect them with different latency-overhead tradeoffs. It can synchronously busy wait or block wait for the notification or it can register a user-defined handler to execute whenever the notification arrives.

- *Support out-of-order delivery networks.* NetVM ensures that a remote application receives a notification only after all previous updates from the same sender have completed, regardless of the order in which they arrive at the remote node.

3.1.3 Synchronization objectives

- *Provide standard atomic operations.* NetVM implements a set of standard low-latency atomic operations that applications can use to build higher-level synchronization operations (e.g., the MCS distributed lock).
- *Natively support wait queues and semaphores.* NetVM provides an extended set of atomic operations to implement wait queues and semaphores with low latency, by reducing the number of required network transactions.

3.1.4 Protection objectives

- *Enforce local access protection.* NetVM uses VM-based protection to prevent local applications from interfering with other local applications, with the kernel, or with the network interface.
- *Enforce remote access protection.* NetVM uses capability-based protection to prevent applications from interfering with remote memory. Assuming the network is physically secure, an application can only access remote memory if it has obtained the proper authorization.

3.2 Key operations

Type	Operation	Description
initialization and segment setup	register export/unexport import/unimport	initialize with NetVM declare/revoke a local segment for data transfer declare/revoke a remote segment for data transfer
remote memory access	write[F][N] read[F] flush[R][W]	transfer to remote segment [F=fenced][N=notification] transfer from remote segment [F=fenced] wait for transfer completion on local node [R=read][W=write]
notification	notfTest/notfSpin/notfWait notfArm/notfDisarm	poll/spin-wait/block-wait for a notification event register/deregister notification handler
atomic and synchronization operations	swap/cswap/testandset/incr/decr acquire/release insertQ/waitQ/removeQ mbegin/mend/cvwait/cvsignal Swait/Ssignal	standard atomic operations acquire/release mutual-exclusive lock enqueue-onto/wait-on/dequeue-from wait queue begin/end monitor, wait-on/signal condition variable wait-on/signal semaphore

Table 4. Key operations in the NetVM API.

Table 4 shows the key operations in the NI Application Programming Interface (API). NetVM exports these operations to applications through a user-level library. The table has four sections.

The first section describes operations that applications use to register with NetVM and to declare local and remote memory regions for communication. The second section describes the remote memory access operations that write to, or read from, remote memory. The third section describes the notification operations that applications use to detect, handle and acknowledge incoming notification signals. The final section describes the NetVM synchronization operations.

3.2.1 Initialization and segment export/import

The first section of Table 4 describes operations that applications use to register with NetVM and to declare local and remote memory regions for communication. An application initializes with NetVM by calling **register**, specifying an available NetVM port number and the size of a notification queue. This port number together with the node number uniquely identifies the application within the network. The notification queue supports the notification mechanism, which Section 6.1 describes.

An application calls **export** to make a contiguous virtual address range in local memory accessible to NetVM as a source or destination for network data-transfer operations. The application specifies the address range and assigns a name for the segment in the export call. NetVM assigns an export handle and a random 64-bit protection key for each exported segment. The application revokes an export by calling **unexport** with the export handle, which immediately disables remote access to that segment. NetVM will also abort an active transfer in progress involving any memory that has just been unexported.

An application binds to a remote segment by calling **import**, supplying the remote node, port and name of the exported segment. The importing application sends an import message to the operating system of the remote node, which matches the segment name and responds with the virtual address range and protection key of the requested segment. The user library in the requesting application assigns an import handle for each imported segment. The application revokes an import by calling **unimport** with the import handle, which immediately disables its own access to the remote segment.

The application can name a remote address in two ways. The first way is by its *import handle-offset pair*, which comprises the handle and offset into the imported segment. The second way is by its *full remote memory address*, which comprises the remote node and port number, and the virtual address in the remote exported segment.

3.2.2 Remote memory access

The second section of Table 4 describes the remote memory access operations that write to, or read from, remote memory. An application accesses remote memory by issuing **write** or **read** calls directly to the network interface. Both sender-managed data-transfer calls must specify the local virtual address and the remote address in the form of an import handle-offset pair. Alternatively, a helper function converts a full remote memory address into its corresponding import handle-offset pair. **write** transfers data from the local source address to the remote destination address, **read** transfers data in the opposite direction.

An application detects the completion of a write or read operation using the **writeF**, **readF** and **flush[R][W]** calls. **writeF**, or *fenced* write, extends **write** and returns to the calling application only after the network interface has processed the current and all preceding write calls. **writeF** provides source-side fencing; a return only implies that the application can safely reuse and overwrite all previously transmitted buffers on the source node, it does not imply that the data has arrived at the destination memory. **readF**, or *fenced* read, extends **read** and returns to the calling application only after the current and all preceding read operations have completed. **flush[R][W]** busy waits until all active **read** (R), **write** (W), or both (RW), calls have completed on the local node.

A local application can optionally transfer control to a remote application by including a notification identifier using the **writeN** call. The remote application has four alternatives to detect and handle the notification, which the following section will describe.

3.2.3 Detecting notifications

The third section of Table 4 describes the notification operations that applications use to detect, handle and acknowledge incoming notification signals. An application has four alternatives to detect notifications from a remote application. It can call **notfTest** to poll the notification counters and determine if there are any pending signals, it can call **notfSpin** to busy wait for the signal, it can call **notfWait** to block wait for the signal, or it can call **notfArm** to register a notification handler procedure, which will automatically execute whenever a relevant signal-carrying message arrives. **notfDisarm** deregisters the notification handler procedure from NetVM.

3.2.4 Atomic and synchronization operations

The final section of Table 4 describes the synchronization operations, which are divided into five groups. The first group consists of a set of five standard atomic operations that an application can access. **swap** atomically exchanges a local operand with a remote 32-bit word. **cswap** takes in a *compare* and a *replace* argument. It matches the remote word with the *compare* argument and overwrites it with the *replace* argument if they match. It also returns the original value of the remote word regardless of the outcome. **incr** and **decr** atomically increments and decrements a remote 32-bit integer respectively. **testandset** conditionally sets a remote word to a nonzero value if it was previously zero and returns the result of the test in either case.

The second group is a set of two operations for an MCS-lock-based distributed mutual-exclusion lock. An application calls **acquire** to gain mutual-exclusive access to a locked resource. It calls **release** to return and release the lock, which also wakes up the next application waiting to acquire the lock.

The third group is a set of three operations for a NetVM distributed wait queue. An application calls **insertQ** to insert itself into a NetVM wait queue. It then calls **waitQ** to busy or block wait for a signal on the wait queue. A remote application calls **removeQ** to signal the wait queue, which wakes up the front application waiting in the queue.

The fourth group is a set of four operations for Mesa monitors implemented using NetVM distributed locks and wait queues. An application calls **mbegin** and **mend** to enter and to exit the monitor respectively. Within a monitor, it calls **cvwait** on a condition variable to place itself on the wait queue for that condition variable. A signaling application calls **cvsignal** to signal the first, if any, application waiting on the specified condition variable.

The final group is a set of two operations for NetVM counting semaphores implemented by extending the wait queue design. An application calls **Swait** to wait on a semaphore, which blocks if the semaphore count is previously zero. It calls **Ssignal** to signal a semaphore, which wakes the first blocked application, if any, waiting on the semaphore.

3.3 System architecture

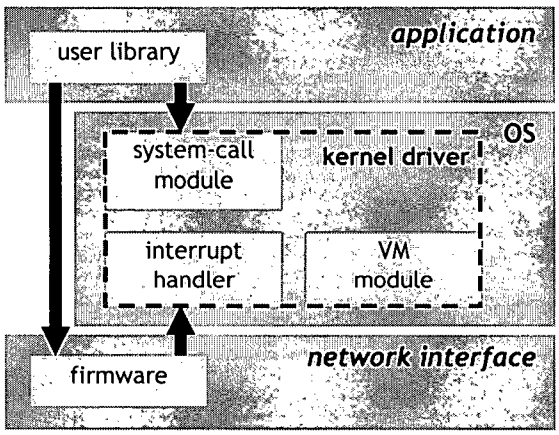

Figure 2. NetVM module block diagram.

Figure 2 shows the basic architecture of NetVM. At the top of the diagram, an application links against the *user library* to access the NetVM API. The user library parses application requests into either system calls for the NetVM kernel driver or command descriptors for the network interface firmware. The FreeBSD [29] kernel driver has three modules. First, the *system-call* module handles application requests to initialize, export and import memory segments. Second, the *VM module* extends the kernel VM system to provide memory-management support for the network interface. Third, the *interrupt handler* services interrupts from the network interface. At the bottom of the diagram, the firmware for the Myrinet network interface exports a set of operations that applications invoke, through the user library, for accessing remote memory.

3.4 Myrinet programmable network interface

Figure 3 on the following page shows the general architecture of the Myrinet programmable network interface [10, 54]. The network interface connects to the host computer through a 64-bit 66-MHz PCI IO bus and to the network through a 1.28+1.28Gb/s link interface. The main components onboard are a network processor and local memory to execute host programmable firmware, and three DMA engines to transfer data between host and local memory, and between local memory and the network.

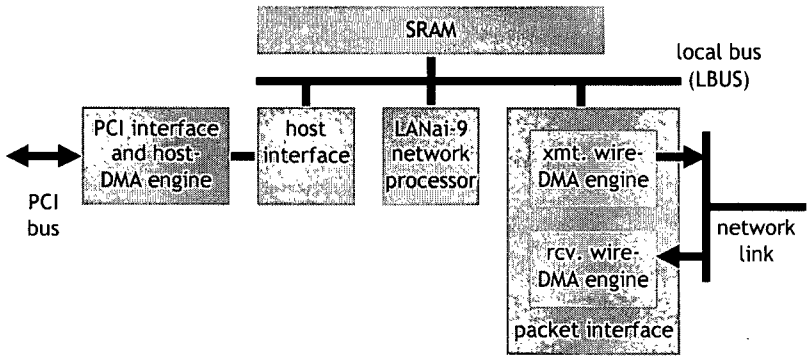

Figure 3. General architecture of the Myrinet LANai-9 network interface.

At the core of the network interface is a 132-MHz *LANai-9.2 RISC processor* [53]. It executes a Myrinet control program that the operating system downloads into the onboard *SRAM*. The 8-MB 64-bit SRAM connects to the processor through a local data bus (LBUS), which operates at twice the clock rate of the network processor. The network interface arbitrates and interleaves LBUS access to the local memory among the network processor and DMA engines with the following decreasing priority order: host DMA, receive-wire DMA, transmit-wire DMA and network processor. Therefore, it is possible for active DMA operations to stall the network processor trying to access local memory.

The network interface connects to the PCI bus through a *PCI interface* and a *host-DMA engine*. The theoretical peak DMA bandwidth over a 64-bit 66-MHz PCI bus is 528MB/s. However, most systems cannot achieve this peak bandwidth due to overheads in the interconnection between system memory and the PCI bus on the host. The *host interface* module connects the DMA engine to the SRAM, which completes the data path between host memory and local memory. The network processor initiates the host-DMA engine to transfer data from any physical address on the host to any SRAM address on the network interface.

The PCI interface supports memory-mapped IO. An operating system can map a region of the host PCI addressing space onto the local memory and register area on the network interface. Thus, the host processor can directly access the entire SRAM, using programmed IO, through this memory-mapped window. Similarly, the operating system can also allow an application to map a restricted window into the interface so that the application can directly interact with the network controller without relying on a system call into the kernel to communicate with the controller.

The PCI interface also supports a *doorbell* mechanism, which is a FIFO queue in the network interface. This mechanism allows multiple applications on the host to safely multiplex control

words into the network interface. The host processor executing the application writes the control word anywhere within the shared PCI address range allocated to the doorbell, the network interface redirects the address and data of the programmed-IO write operation into a predetermined queue in local memory. In this way, the network processor needs only to check a single location for new control words from any application on the host.

The network interface connects to the network link through a packet interface. This packet interface comprises two independent wire-DMA engines: one for transmitting data to the network link and one for receiving data from the link. Each engine operates at 1.28Gb/s.

3.4.1 Programmed-IO—host-DMA tradeoff

Two mechanisms to transfer data between host memory and the network interface are programmed IO and host DMA [71]. With programmed IO, the host processor accesses network interface memory by directly accessing the memory-mapped addresses. With DMA, the host or network processor programs the DMA engine on the network interface to transfer data in bulk. DMA transfers have much higher bandwidth than corresponding programmed-IO transfers. However, they also require a setup time to program the transfer parameters for source and destination addresses and data size into the DMA engine registers. As a result, programmed-IO transfers are better suited for host-processor-initiated small transfers, whereas DMA transfers are better suited for large transfers, or are required when the network interface initiates transfers without any host-processor intervention. The LANai-9 network processor can only access host memory through its DMA engine and thus will always incur the DMA setup overhead whenever it initiates a transfer between host memory and the network interface.

Figure 4 on the following page shows the measured latency and throughput for various transfer sizes using programmed-IO and host-DMA transfers on the existing hardware prototype. In both host-to-network (H2N) and network-to-host (N2H) transfers, the latency and throughput of host-DMA operations are significantly better than programmed-IO operations for most transfers. Specifically, the programmed-IO transfers are only better than host-DMA transfers when the transfer size is less than 112 bytes for host-to-network (H2N) transfers and less than 8 bytes for network-to-host (N2H) transfers. NetVM uses this result to optimize the latency for small messages, by adaptively switching between programmed-IO and DMA transfers depending on the message size.

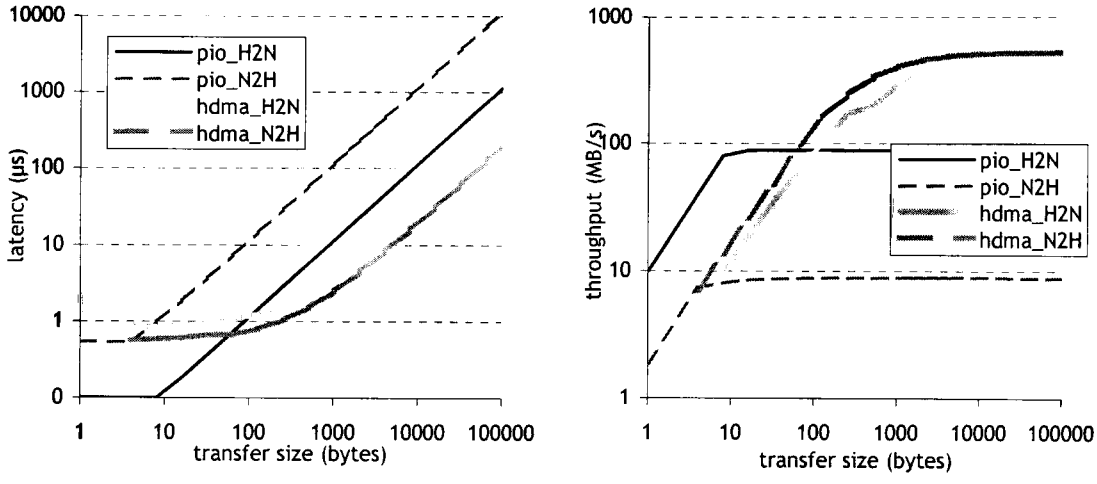

Figure 4. Latency and throughput of programmed-IO vs. host-DMA transfers.

3.5 Summary

NetVM allows applications to access data in virtually-addressed remote memories in a reliable and protected way. In addition, an application can optionally notify a remote application during a data transfer. The remote application detects these notifications either by busy waiting, by block waiting, or by triggering a handler to process them. NetVM delivers notifications in issue order even over an out-of-order delivery network. Finally, NetVM directly supports efficient wait queues and counting semaphores by implementing new atomic operations that reduce the number of network transactions required for synchronization operations.

The NetVM architecture consists of three key modules. First, the application links against the *user library* to access the NetVM API, which provides the interface to the functionality summed up in the preceding paragraph. Second, the *kernel device driver* handles application system call requests through the API, integrates with the VM system to manage the shadow page table, and implements the interrupt handler for the network interface. Finally, the *firmware* on the network interface handles direct application requests through the API and messages from the network.

NetVM is implemented with the Myrinet programmable network interface. This interface consists of the LANai-9 processor, on-board SRAM and three DMA engines to access host memory and a bidirectional network link. On this particular interface, DMA is more efficient for bulk data transfers across the PCI IO bus, but programmed IO is more suited for small transfers due to the high setup overhead for the DMA engine.

4 Memory management

A central feature of NetVM is that applications can directly access unpinned virtual memory. To do this, NetVM shadows portions of the host page table on the network interface by intercepting key host VM-system operations. The VM module maintains this shadow page table for all pages that local NetVM applications export. The NI uses this table to lock and obtain the physical address of a page before accessing it with a DMA transfer. Both host and NI synchronize on individual page table entries to ensure translation consistency. The host VM system delays removing a page mapping if the NI has current locked the same page for a DMA transfer.

4.1 Import-Export memory segments

NetVM transfers data only between local memory segments that an application exports and memory segments that it imports from remote applications. The user library provides an import-export API to manage these segments, it maintains the import map and an export map required to support the data-transfer operations. The kernel driver also maintains a copy of the export map required to update the shadow page table and to handle remote import requests.

4.1.1 Import-Export API

Type	Operation	Return	Description
export	export(name, addr, size)	seg	export local virtual-address range
	unexport(seg)	-	remove exported segment
	export_lookup(addr)	(seg, offset)	convert local virtual address to segment-offset
import	import(node, port, name)	seg	bind remote exported segment
	unimport(seg)	-	remove imported segment
	import_lookup(node, port, addr)	(seg, offset)	convert remote virtual address to segment-offset

Table 5. NetVM import-export operations.

An application exports a virtual address range, or memory segment, to allow remote applications to access it, and to allow the network interface to transfer data to or from it via host

DMA. The application imports a remote exported segment before accessing that memory. Table 5 shows the import-export API that the NetVM user library provides to applications in two sections. The first section lists the export operations. The **export** system call specifies the base address, size and a 64-bit name of a virtual memory segment and returns an handle for the newly exported segment. **unexport** unmaps an exported segment and immediately prevents NetVM, and remote applications, from accessing it. **export_lookup** maps a local virtual address into its associated segment-offset pair, which is the exported segment handle and the offset into that segment.

The second section of the table lists the import operations. An application calls **import** to bind to a named remotely exported segment. A successful **import** returns a local handle that the application uses to refer to the imported segment. **unimport** unbinds an existing import and immediately prevents the application from accessing the remote segment. **import_lookup** maps a remote application address, specified by the node, port and virtual address, into its associated import segment-offset pair. NetVM data-transfer API calls require the application to specify the segment-offset pair to name remote memory. **import_lookup** provides a convenient alternative to map a global remote application virtual address into the required arguments for those calls.

4.1.2 Import-Export maps

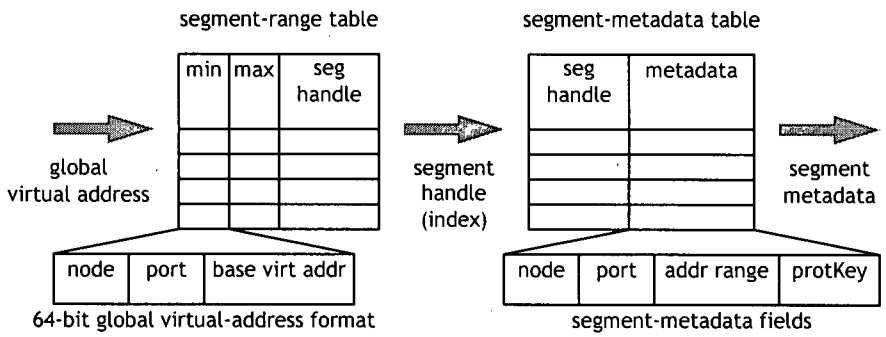

Figure 5. NetVM import and export maps.

Figure 5 shows the data structures that NetVM uses to maintain the application imports and exports. Each application stores an *import map* and an *export map*, which have identical formats. The user library uses the maps to obtain the required location and protection information and construct the data-transfer commands for the network interface. The kernel also stores a

copy of the export map for each application. It uses the map to handle remote import requests and to handle the shadow page-table updates during page faults. Both import and export maps have identical formats, they respectively store only remote or local mappings.

The import or export map consists of a *segment-range table* and a *segment-metadata table*. The segment-range table is a binary-search table (BST) that maps non-overlapping global virtual-address ranges into their corresponding segment handles. The system-wide unique 64-bit global virtual address is the concatenation of the 16-bit node number, 16-bit port number, and 32-bit base (or limit) virtual address of a segment. The segment handle is also the index into the segment-metadata table. This table stores attributes about each segment, including its location in the network, virtual address range and protection key.

Using a BST efficiently obtains the segment metadata from the unique global virtual address in O(logS) time where S is the number of segments. This fast look up is important because it is in the critical path of a data-transfer or page-fault operation. Using a hash table is insufficient, because the table would need to store each individual page address as a hash key to fully map a segment.

4.1.3 Import-Export operations

An application exports a segment by calling the NetVM **export** system call with the virtual address range and name of the segment. The system-call module first verifies that the address range is not overlapping with existing exported segments. It then generates a random 64-bit protection key for the segment and inserts a new entry into the kernel's copy of the export map. Finally, it returns the local handle of the newly exported segment, which is the index into the segment-metadata table, together with the generated protection key to the application. The application uses the returned results to update its own identical copy of the export map.

The NetVM **unexport** system call revokes a local exported segment. In addition to deleting the segment entry from the export map, the kernel module also scans the set of resident pages and removes any mapped pages that belong to the segment from the shadow page table, so that NetVM cannot access them from the network interface. The kernel synchronizes with the network interface to remove page mappings from the shadow page table to ensure translation consistency.

The NetVM kernel module intercepts the process-exit code to clean up after a NetVM application that either crashed or exited without properly closing the NetVM port. The module calls

unexport for each exported segment in the exiting application's export map, which removes the page mappings in the shadow page table for any resident pages in the exported segment.

An application imports a segment by specifying a node, port and the segment name in an **import** request message to the remote kernel module. The kernel module on the remote node uses the NetVM port and segment name to locate and look up the kernel copy of the requested application's export map. If the named entry exists, it replies to the importing application with the attributes, including the virtual address range and protection key, of the exported segment. The importing application updates its import map with the returned attributes of the newly imported segment. **unimport** simply removes the entry from the import map.

4.2 Host VM-system integration

NetVM intercepts key VM system operations to update the shadow page table on the network interface. The NetVM VM module provides three operations for the host to manage the page table: insert a page mapping whenever the host pages in a NetVM application page; remove the mapping whenever the host pages out or frees the page; and detect if the network interface has modified a mapped page. NetVM requires only four simple changes to the host VM system, each change calls one or more of the three exported operations.

To support the page-mapping operations, the NetVM VM module maintains a system-wide process mapping table and a copy of the export map for each application. During a page fault, these tables translate the operating system's page-fault parameters, available only to the host VM system, into the arguments required by the NetVM VM module to update the shadow page table.

4.2.1 FreeBSD VM-system operation

Figure 6 on the following page shows the key operations on the FreeBSD VM system that are pertinent to NetVM. The host VM system manages the host pages in VM page queues using an approximate LRU policy. Pages on the host are in one of five possible states: active, inactive, cache, free and wired. With the exception of wired, the state of a page determines the queue that stores it. Initially, all pages belong to the free list.

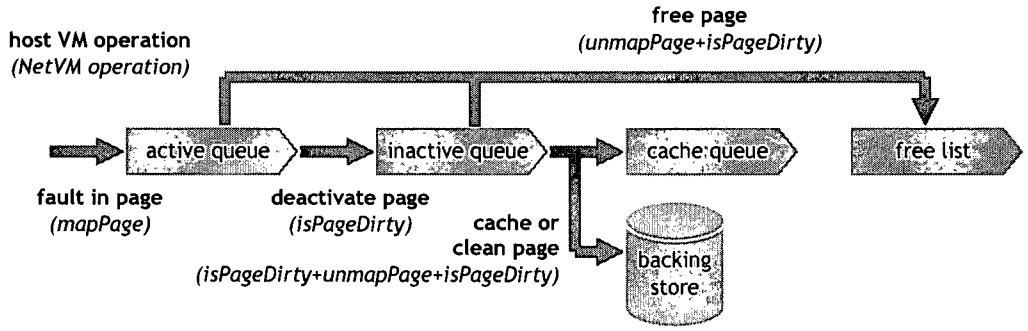

Figure 6. FreeBSD VM-System page queues.

When an application faults on a nonresident page, the *page-fault handler* allocates a free page for the application and places it into the active queue. To balance the size of the queues, the *page-deactivation* routine deactivates dormant pages in the active queue and moves them into the inactive queue. If the system has insufficient available pages, the *page cleaner* scans the inactive queue and either frees unused pages, transfers clean pages into the cache queue, or flushes modified pages to the backing store. An exiting application releases all its allocated pages back to the free list.

NetVM intercepts key VM operations to maintain the shadow page table on the network interface. These changes call corresponding NetVM operations shown in parenthesis in Figure 6. The remaining sections describe the NetVM interface and the required modifications to the host VM system.

4.2.2 NetVM VM-System interface

Operation	Return	Description
mapPage(VPN, vmpage, protKey)	-	insert a page mapping
unmapPage(vmpage)	SUCCESS or LOCKED	conditionally remove a page mapping
isPageDirty(vmpage)	DIRTY or CLEAN	test if NI has modified a page and reset its state

Table 6. Host page-mapping operations.

Table 6 lists the operations that NetVM exports to the host VM system to manage the shadow page table in the network interface. **mapPage** inserts a new page mapping of an application virtual page number (VPN), which is a node-wide unique page identifier, into the page table. The *vmpage* argument is an operating system metadata structure that describes a host page including its physical address and modification state. **unmapPage** conditionally removes a page

mapping, this operation may fail if the NI has currently locked that page. **isPageDirty** returns DIRTY and updates the operating system state for an NI-dirtied page, which the NI has modified since the previous **isPageDirty** call for the same page. As a side effect, it also resets the modification state for that page to CLEAN.

4.2.3 Host VM-system modifications

NetVM requires four simple changes to the host VM system. Also shown in Figure 6, these changes correspond to the page-state transitions, by the VM system, of all NetVM exported pages. Each change amounted to adding a single block statement to call the appropriate stub function in the NetVM kernel driver and to redirect the VM operation when necessary. The first change is that the page-fault handler calls **mapPage** to insert a new page mapping whenever it pages in an exported page into host memory.

The second change is that the page-deactivation routine in the pageout daemon calls **isPageDirty,** to test if the NI has modified an exported page, just before deactivating an exported page. If the NI has modified the page, the daemon skips this page and selects another one to deactivate. Otherwise, it deactivates the page by moving it to the inactive queue.

The third change is that the page cleaner calls **isPageDirty** and, if the NI has modified the page, *reactivates* the page by moving it to the active queue. If, however, the page was clean, the cleaner calls **unmapPage** to remove the page mapping from the network interface in preparation for moving it out of the inactive queue. **unmapPage** may fail because the NI has locked the page. In this case, the cleaner also simply reactivates the page, which acknowledges that the page is still active. Immediately after successfully calling **unmapPage**, the cleaner calls **isPageDirty**, a second time, to update the dirty state in the host page-table entry. Using this updated state, the cleaner either flushes a modified page to the backing store, or moves a clean page into the cache queue. This second **isPageDirty** call is necessary because the NI may have modified the page between the first **isPageDirty** and **unmapPage** calls. NetVM must reflect this modification by updating the host page-table entry with the second **isPageDirty** call.

The final change is that the page-free routine calls **unmapPage** and **isPageDirty** to remove its page mapping and to reset its modification state. If **unmapPage** fails, the kernel spins, retrying the operation, until it succeeds in forcibly removing the page mapping from the network interface. This spinning is not detrimental because it occurs only when the NI is currently performing DMA on the same page. In any case, the failed call will succeed as soon as the DMA operation completes, which is a delay of at most 9.4μs on the current hardware prototype.

4.2.4 Translating OS page-fault parameters into NetVM mapPage arguments

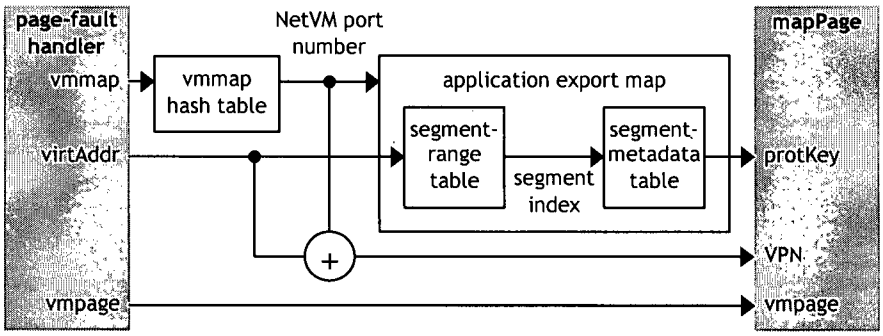

Figure 7. Translating application page-fault parameters into NetVM mapPage arguments.

The **mapPage** operation requires the VPN, physical address and protection key of a faulting page to insert its mapping into the network interface. However, the VM system page-fault handler has access to only the process vmmap, the faulting page's virtual address and its corresponding physical address. The host obtains the required arguments for **mapPage** through a series of lookup operations shown in Figure 7.

The NetVM system-call module maintains a system-wide vmmap hash table for each node and an export map for each application on the node. The vmmap table maps a registered process's vmmap pointer to its NetVM port number. The system-call module updates this table whenever an application opens or closes a NetVM port. It also maintains an export map, described in Section 4.1, to store the metadata describing each segment that the application exports.

The page-fault handler obtains the protection key for a faulting page in three steps. First, it uses the faulting process's vmmap pointer to search the vmmap hash table and obtain its associated NetVM port number. Second, the handler uses the faulting virtual address to search the application export map, identified by the port number, for the index of the segment containing the faulting page. Finally, it uses the index to look up the segment-metadata table and obtain the protection key for the faulting page in the exported segment.

The page-fault handler concatenates the process's NetVM port number with the faulting virtual page number to compute the node-wide unique VPN. This VPN, together with the protection key and the original vmpage parameter, make up the required arguments for the **mapPage** operation.

4.3 Network-interface page table and physical map

The shadow page table on the network interface consists of two data structures: a page table (PT), which stores mappings from NetVM virtual page numbers (VPNs) to physical page numbers (PPNs), and a physical map (PMAP), which stores NetVM metadata including the reverse mapping for all resident pages that NetVM applications export on the host.

The design of the data structures faces two important constraints. First, they must be compact to fit in the limited memory on the network interface. The data structure sizes on the network interface should scale with available memory on the host, assuming that available network-interface memory grows proportionally with available host memory. Second, they must allow concurrent access by the host and network interface processors.

4.3.1 Page table

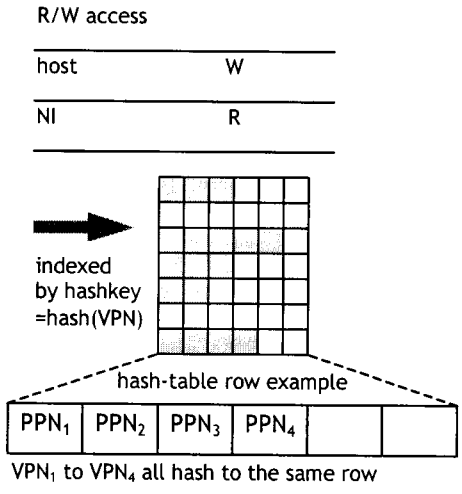

Figure 8. Page-table organization on the network interface.

Figure 8 shows the page table on the network interface. It is a fixed hash table organized as a 2D array. The index into the hash table is the VPN hash key, which the host computes by hashing the VPN. Each hash row stores a fixed number of contiguous 32-bit PPN entries terminated by a zero value. Requiring contiguous entries in the row speeds up the lookup operation by the network interface because the NI can retrieve all valid row entries without scanning the entire row. The VPN itself is not included in the table to conserve space; instead, it is stored in the PMAP as a reverse mapping.

Only the host modifies the page table, using programmed IO, to insert and remove page mappings. The network interface reads from, but does not modify, the table to translate addresses. This asymmetry is necessary to allow concurrent access to the page table by the host and network interface without needing each other to synchronize. It is still possible for the network interface to read an invalid mapping if the host is simultaneously changing that mapping. NetVM uses the PMAP to detect this error and discard the mapping.

The **mapPage** operation inserts a new entry for a page into the page table by first hashing its VPN to locate the hash row and then inserting its PPN at the end of the row. To eliminate the cost of linear probing with programmed IO on network interface memory, NetVM maintains an array of counters in host memory that records the number of entries in each hash row. In the rare chance that the row is already full, **mapPage** replaces a randomly selected victim, from the same row, with the new entry. If the NI later tries to access but cannot find the victim page in the page table, it redirects the operation to the host kernel, which will reinsert the old page mapping and, if necessary, select another victim entry.

The **unmapPage** operation removes a page-table entry by simply replacing it with the last entry of the hash row to maintain the invariant of contiguous row entries. Because only the physical address PPN is available as an argument, **unmapPage** uses the reverse mapping in the PMAP to first obtain the VPN and compute the hash row. It then linearly scans the row, searching for the matching PPN entry, to remove it.

4.3.1.1 Hash-table analysis

There is a probability that the NI cannot locate a previously inserted entry in the hash table. When the host inserts a new entry into a full row, it evicts and replaces a random victim in the same row with the new entry. The NI will not be able to locate the evicted entry in the hash table and has to redirect its operation to the host kernel. The following paragraphs present a simple hash-table analysis to determine the likelihood of this event occurring based on the hash-table parameters and load.

For the analysis, assume that the hash function that maps the VPN into the hash-table index has a uniform distribution. Assume also that the NI is equally likely to look up any of the previously inserted entries, but will not look up uninserted entries.

The probability that the NI cannot locate a hash-table entry due to eviction is

$$P(evicted) = \sum_{i=d+1}^{n} \binom{n}{i} \left(\frac{1}{m}\right)^{i} \left(1-\frac{1}{m}\right)^{n-i} \left(\frac{i-d}{i}\right).$$

where

m is the number of hash-table rows,

d is the number of entries in a hash-table row, and

n is the number of entries inserted into the hash table.

Proof:

Consider a row k.

The probability that a single entry hashes to row k is $p = \frac{1}{m}$. For a sequence of n insertions into the hash table, the probability that i entries hash to row k is $p^i(1-p)^{n-i}$. Because the number of possible sequences that lead to the same outcome is $\binom{n}{i}$, the probability of i insertions occurring in row k after n insertions into the hash table is $\binom{n}{i} p^i (1-p)^{n-i}$.

If $i \le d$, the row stores all i entries and thus the NI can always locate the required entry. If $d < i \le n$, the NI has an equal probability to look for any entry in the row, whether it is evicted or not. Therefore, the probability that the NI cannot locate the required entry in that row in this case is bounded by $\frac{i-d}{i}$ (it is zero only if the required entry is the most recent inserted entry in the row).

Hence, after n insertions into the hash table, the probability that NI cannot locate the required entry in a row that has exactly i insertions is

$$P(evicted_i) = \begin{cases} 0 & (i \le d) \\ \binom{n}{i} p^i (1-p)^{n-i} \left(\frac{i-d}{i}\right) & (d < i \le n) \end{cases}.$$

Summing up for all i between $d+1$ and n inclusive, the probability that the NI cannot locate the required entry in a row due to eviction is $P(evicted) = \sum_{i=d+1}^{n} \binom{n}{i} p^i (1-p)^{n-i} \left(\frac{i-d}{i}\right)$.

Because the NI is equally likely to index into any of the m rows during a lookup (uniform hash distribution), therefore the probability that the NI cannot locate a hash-table entry due to eviction is the same as the probability for a single row, which is

$$P(evicted) = \sum_{i=d+1}^{n} \binom{n}{i} \left(\frac{1}{m}\right)^i \left(1-\frac{1}{m}\right)^{n-i} \left(\frac{i-d}{i}\right).$$

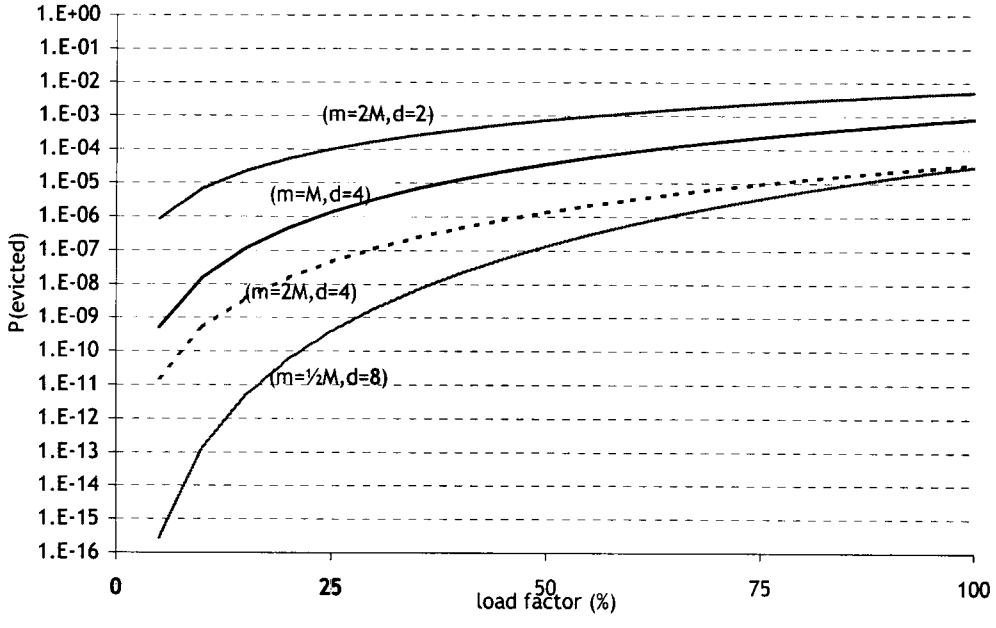

Figure 9. Probability that NI lookup hits an evicted entry in the hash table.

Figure 9 shows the computed probability, for different hash-table configurations, that the NI lookup will hit an evicted entry in the hash table for a range of load factors. The load factor is the ratio of n to M, where M is the total number of host memory pages. To recall, n is the number of inserted entries, m is the number of hash-table rows and d is the size of a hash-table row.

The three solid lines show the results for three hash-table configurations that occupy the same total storage space of $4M$ ($md=4M$ in each case). A higher row depth, with a correspondingly lower row count, results in a smaller probability of looking up an evicted entry. However, the same trend also results in a higher average number of entries per row, which increases the lookup time. NetVM uses ($m=M$, $d=4$) as a compromise, because the estimated cost of redirecting the operation to the host, due to eviction, is less than 10^3 times the cost for the NI to complete the operation. In this configuration, the average number of entries per row is 1 and the probability of a not finding a previously mapped entry in a fully loaded hash table is less than 10^{-3}. For a half-loaded hash table, this probability drops to less than 4×10^{-5}.

4.3.2 Physical map

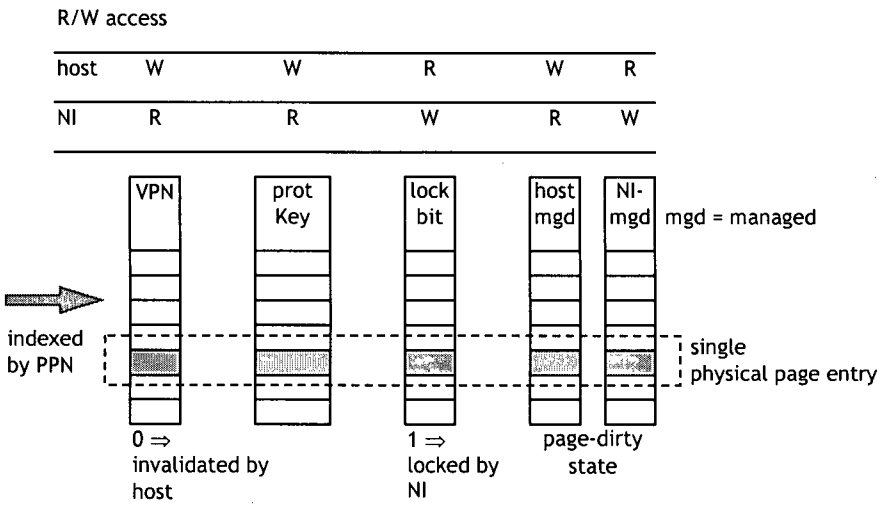

Figure 10. PMAP organization on the network interface.

Figure 10 shows the PMAP data structure on the network interface. The PMAP stores the metadata for physical pages on the host that NetVM has currently mapped into the network interface.

The PMAP consists of five separate arrays all indexed by the PPN. The VPN array stores the 32-bit reverse mappings for the page table. Each physical page has only one VPN and, therefore, has only one reverse mapping. The protection key (protKey) array stores the 64-bit protection keys that the NI uses to verify that a remote application has the right to access the requested pages by matching them with the keys carried in the request message. The lock-bit array stores one flag for each page, which the NI uses to indicate its intent to lock a page in host memory.

Finally, the page-dirty state table actually consists of two bit arrays: host-managed and NI-managed. Section 4.3.2.1 describes host how these two arrays work together to maintain the list of NI-modified pages. The size of the PMAP on the network interface is proportional to the physical memory on the host, each physical page entry requires only 99 bits: 32 bits for the VPN, 64 bits for the protection key and 1 bit each for the page-lock flag, host-managed and NI-managed page-dirty states.

The shadow page table currently supports only a 1:1 mapping between an application VPN and PPN. On the one hand, this approach limits the data structure sizes so that they scale linearly with the amount of physical memory on the host. On the other hand, NetVM applications cannot share access to physical pages on the host. Extending the design to support an M:1 mapping, for a small fixed value of M, is straightforward and requires only two changes to the data structures. The first change is to increase the size of the page table by up to M times to support more virtual-to-physical page mappings. The second change is to increase the number of reverse mappings and protection keys by including M, instead of one, VPN and protKey arrays in the PMAP.

Also shown in Figure 10, either the host or the NI, but not both, modifies each array in the PMAP. Ensuring a single writer for each data structure allows concurrent access by the host and NI. However, both the host and NI need to synchronize with each other when the host wants to remove a mapping for a page that the NI wants to lock. Section 4.4 describes this host-NI synchronization in detail.

4.3.2.1 Page-dirty state table

Figure 11 on the following page shows the operation of the page-dirty state table on the network interface. It consists of two bit arrays: host-managed and NI-managed. Only the host writes to the host-managed bit array and, likewise, only the NI writes to the NI-managed bit array. With this scheme, corresponding bits with equal value (both 0 or both 1) in the two arrays indicate that the NI has modified that page. The NetVM driver initially sets all the bits in the host-managed array and clears all the bits in the NI-managed array. The NI **unlockPageDirty** marks a page modified, after a DMA transfer to the host page, by copying the host-managed bit to the NI-managed bit. The host tests and marks a page clean, within **isPageDirty**, by copying the complement of the NI-managed bit to the host-managed bit.

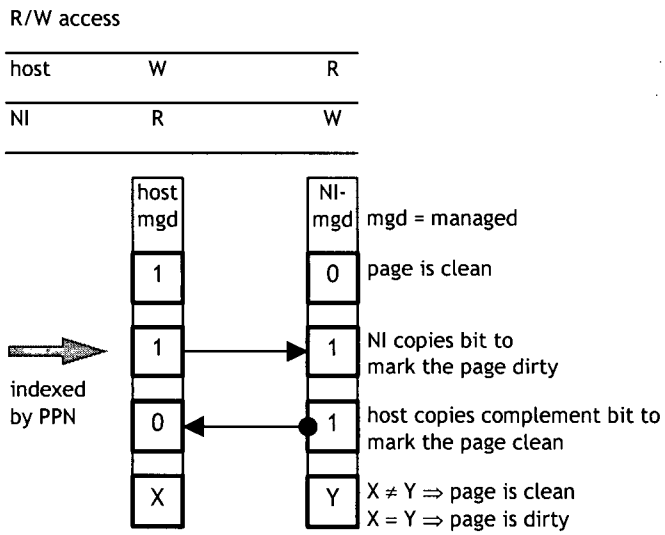

Figure 11. Page-dirty state bit arrays.

A possible race condition occurs when the host tests and marks a page clean at the same time that the NI marks a page modified. This race does not affect the correct operation of the page-dirty state table because the host test will never detect a modified page as clean. However, the final state of a modified page may still be dirty even after the host marks it clean, depending on the relative ordering between the host and NI operations. Recall from Figure 6 on page 47 that the host VM system calls **isPageDirty** at four places: once each in the page-free and page-deactivation routines, and twice in the page cleaner. In the page-free routine and after **unmapPage** in the page cleaner, the host calls **isPageDirty** only *after* it has unmapped the page. Therefore, the NI cannot mark a page modification during that time because it cannot acquire the unmapped page. For the remaining two places, the host will reactivate the page if it detects a modified page. The page's modification state may remain dirty if the host calls **isPageDirty** in a race. This situation is not detrimental because, in the worst case, the page-deactivation routine unnecessarily reactivates the page when it calls **isPageDirty** the next time, even if the NI has not modified the page.

NetVM uses two separate bit arrays to store the page-dirty state to conserve limited memory space in the network interface while allowing concurrent updates by the host and NI. Reasonably, each page requires only one bit to store the modification state. However, NetVM cannot simply store all the modification flags in a single packed bit array on the network interface because both host and NI processors cannot atomically, and concurrently, update individual bits in the same memory word. Neither processor can perform a read-modify-write operation on a single bit in network-interface memory and the smallest data unit that both processors can

modify is a 32-bit word. An alternative to storing the page-dirty table, which uses more memory space, is to allocate an entire 32-bit word to store the modification state for each page. This word granularity ensures that both processors can update the states for different pages without possibly interfering with each other. A better alternative is to use two separate bit arrays and ensuring there is only one writer for each array. This double-array approach reduces the storage on the network interface by a factor of 16, from 32 bits to only two bits per page.

4.3.3 Address translation and page locking

Operation	Return	Description
lockPage(VPN, hash_key)	PPN or fail	translate address and lock page
unlockPage(PPN)	-	unlock page
unlockPageDirty(PPN)	-	unlock page and set modified bit

Table 7. NI page-locking operations.

Table 7 lists the operations for the network interface to access the shadow page table. **lockPage** translates the virtual address of a page into its physical address in host memory and locks the page to prevent the host from unmapping it. The NI calls **lockPage**, in the critical path of any NetVM data-transfer operation, to acquire the page before accessing it using DMA. Therefore, minimizing the latency of **lockPage** is important, especially for small data transfers. The NI calls **unlockPage** to release the page lock after it completes a DMA *read* transfer from the page in host memory. **unlockPageDirty** is similar to **unlockPage** except that, in addition to unlocking the page, it also sets the modification state for that page after a DMA *write* transfer to the page in host memory.

Figure 12 on the following page shows the **lockPage** operation by the network interface. This procedure requires the VPN and VPN hash key of the page. It will return the PPN of the page in host memory, which the DMA transfer requires, if the operation was successful, or it will return NOT_FOUND if the page is not resident in host memory. **lockPage** first uses the VPN hash key to locate the page-table hash row containing all the PPN entries that potentially correspond to the requested VPN. For each PPN, **lockPage** attempts to lock the page by first setting the lock flag for the page in the PMAP and then verifying that the requested VPN matches the reverse mapping also in the PMAP. The NI successfully acquires the page if the VPNs match and returns the PPN value. If the VPNs do not match, the NI resets the lock flag for the page and tries the next PPN in the hash row. When there are no more entries to try, **lockPage** returns NOT_FOUND to indicate that it cannot locate the page in host memory.

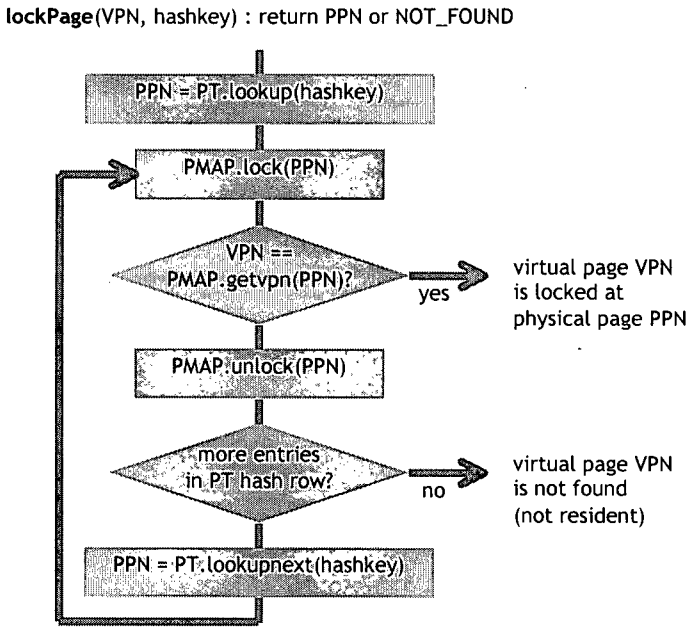

Figure 12. Address translation and page locking on the network interface.

4.4 Host-NI synchronization

Both host and NI cooperate with each other when updating a PMAP page entry to ensure PMAP consistency. The host **mapPage** operation does not require explicit synchronization because it orders the updates by writing the PPN into the page table last, after updating the PMAP fields. As a result, any PPN that the NI reads from the page table will already have a corresponding updated, and coherent, VPN in the PMAP.

The host **unmapPage** and NI **lockPage** operations need to synchronize with each other. A race condition arises if the host is removing the page mapping of a page that the NI is locking at the same time. If both operations had succeeded, the NI would perform a DMA transfer using a stale virtual-to-physical mapping. NetVM prevents this situation by using a mutual exclusion algorithm, based on Peterson [62], to ensure at most only one operation succeeds in a race over the same page. NetVM uses the two-word Peterson-based approach instead of a single-word test-and-set lock because PCI [70] does not support atomic read-modify-write operations across the IO bus.

HOST unmapPPN(PPN)	NI lockPPN(PPN, VPN$_{REQUIRED}$)
save and clear PMAP.VPN[PPN] if PMAP.lock[PPN] is TRUE restore PMAP.VPN[PPN] return PAGE_LOCKED return UNMAPPED_PAGE	set PMAP.lock[PPN] = TRUE if PMAP.VPN[PPN] ≠ VPN$_{REQUIRED}$ restore PMAP.lock[PPN] = FALSE return PAGE_NOT_MAPPED return LOCKED_PAGE

Table 8. Host unmapPPN and NI lockPPN synchronization implementation.

Table 8 shows the synchronization operations **unmapPPN** and **lockPPN** on the host and NI respectively. **unmapPage** calls **unmapPPN** to remove a physical page mapping by first clearing its VPN field and then finding its lock flag clear. It restores the VPN and returns PAGE_LOCKED if the lock flag is set. Correspondingly, **lockPage** calls **lockPPN** to lock a physical page by first setting its lock flag and then finding a valid VPN for the page, as described in Section 4.3.3. It clears the lock flag and returns PAGE_NOT_MAPPED, indicating that this page is not resident, if it found an invalid VPN.

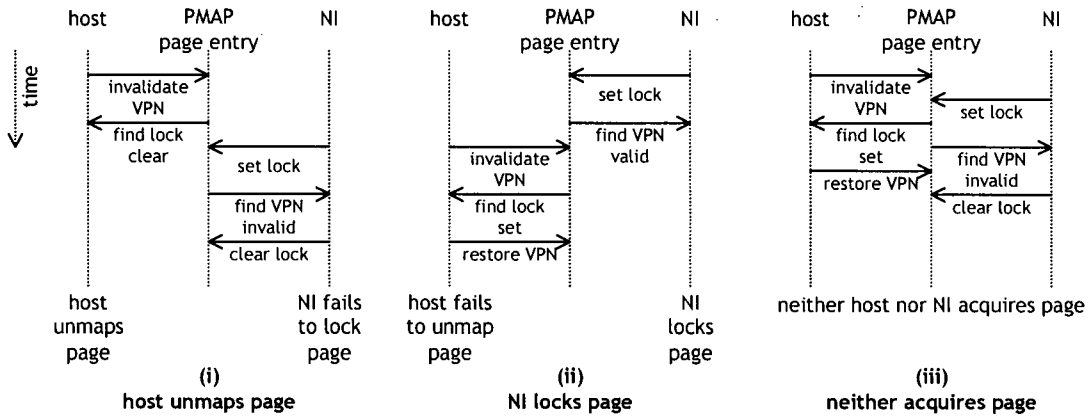

Figure 13. Three possible outcomes when the host and NI synchronize over the same page.

Figure 13 shows the three possible outcomes when the host and NI both try to acquire the same physical page simultaneously. In the first outcome (i), the host successfully removes a page mapping by invalidating the VPN and finding a clear lock flag *before* the NI can set it. The NI interprets this page as nonresident, because of its invalid VPN, and continues to scan the page table for additional PPN entries to try. The NI redirects the transfer to the bounce buffer only when there are no more entries in the page-table hash row.

In the second outcome (ii), the NI successfully locks the page by setting the lock flag and matching the required VPN in the PMAP *before* the host can invalidate the VPN. The host has two options depending on the VM operation that called **unmapPage**. In the first option, the

page-deactivation routine correctly treats this page as active and simply leaves it in the active queue before selecting another one to deactivate. In the second option, the page-free routine forcibly removes the page mapping by retrying the **unmapPage** operation until it succeeds. To do this, it leaves the VPN invalidated, preventing the NI from reacquiring the page, and spins, reading the lock flag, until the DMA transfer completes and the NI unlocks the page by clearing the flag.

In the third outcome (iii), neither the host nor the NI successfully acquires the page. In this rare case, the host invalidates the VPN but finds a set lock flag and the NI sets the lock flag but find an invalid VPN. Both host and NI take the same recovery actions described in the previous two outcomes. This third outcome is not detrimental because the NI simply redirects the transfer to the bounce buffer, which results in a less efficient one-copy operation instead of a zero-copy transfer even though the page is already mapped on network interface.

4.5 Summary

Integrating memory management between the host operating system and the network interface is a key function in NetVM. NetVM transfers data only between memory segments that an application exports and imports. The kernel module integrates with the host operating system to update and maintain a shadow page table on the network interface by intercepting four key VM system operations. The network interface stores a hash table that tracks virtual-to-physical page mappings and a PMAP that tracks reverse page mappings and page metadata. It uses these data structures to lock and obtain the physical address of a required page before accessing it for DMA transfer. Both host and NI synchronize on individual page entries to ensure translation consistency. The host delays removing a page mapping if the NI has currently locked it for DMA transfer.

5 Data-transfer operations

A NetVM application writes to, and reads from, remote application memory by issuing **write** and **read** commands directly to the network interface. The user library exports a read-write API and converts application requests into command descriptors for the network interface. The network interfaces on the local and remote nodes coordinate with each other to process the transfer operation. They locate and lock resident host pages through the shadow page table and directly transfer the requested data using a zero-copy scheme with the host DMA. If the required page is not resident, the NI redirects the transfer to a system bounce buffer on the host. The kernel driver fetches the page from the backing store and completes the transfer using a single host-to-host memory copy operation.

5.1 Data-transfer API

Type	Operation	Return	Description
write	write(seg, offset, addr, size)	-	write to remote memory
	writeF(seg, offset, addr, size)	-	fenced-write[*] to remote memory
	writeN(seg, offset, addr, size, notf)	-	notifying write to remote memory
	writeFN(seg, offset, addr, size, notf)	-	notifying fenced-write[*] to remote memory
read	read(seg, offset, addr, size)	-	read from remote memory
	readF(seg, offset, addr, size)	-	fenced-read from remote memory
fence	flushR()	-	spin until all reads complete
	flushW()	-	spin until all writes complete locally[*]
	flushRW()	-	spin until all reads and writes[*] complete locally
map	import_lookup(node, port, addr)	(seg, offset)	convert remote virtual address to segment-offset

Table 9. RDMA write, read and flush API.
[*] source-side fencing only.

Table 9 shows the data-transfer API that the NetVM user library exports to applications in four sections. The first section shows four different **write*** calls. Each **write** call requires the imported segment (*seg*) and *offset* into the destination remote segment, and the address (*addr*) and *size* of the source data in local memory. The basic **write** call posts the command into the network interface and returns immediately. **writeF**, or fenced write, is similar to **write** except that it returns only after the current and all previous writes from the sender have completed

on the source network interface, thus allowing the application to safely reuse the source buffers. **writeN**, or notifying write, attaches a notification signal *notf* with the message, which notifies the remote application once all current and all previous writes from the sender have arrived at the remote application. **writeFN**, or notifying-fenced write, combines **writeF** and **writeN** into a single API call.

The second section of the table shows two different **read*** calls. Both **read** calls require the imported segment and offset of the source remote memory location, and the address and size of the destination in local memory. The basic **read** call posts the command into the network interface and returns immediately. The network interfaces will complete the **read** operation without any intervention by, or indication to, the requesting application. **readF**, or fenced read, returns only after the current and all previous read requests have completed.

The third section of the table shows three different **flush*** calls. These calls provide an alternative to the *fenced* modifier in the **writeF**, **writeFN** and **readF** calls. **flushR** and **flushW** spin until all active read and write operations have completed respectively. **flushRW** combines **flushR** and **flushW**. For example, an application can issue a series of asynchronous **read** and **write** calls, followed by a **flushRW** before continuing, to ensure that all requested reads have completed and that it can reuse all source buffers.

Finally, the fourth section of the table repeats the **import_lookup** API from Table 5 on page 43. This operation maps a remote address, using the node, port and virtual address in the remote application, into the import segment-offset pair that the **read*** and **write*** calls require for naming remote memory.

5.2 User-library operations

write-command Descriptor	
Myrinet wormhole route (8 bytes)	
Myrinet message tag (NETVM)	(padding)
sending node	sending node sequence number
source VPN	source VPN hash key
source protection key (64 bits)	
source page offset and length	destination node
destination VPN	destination VPN hash key
destination protection key (64 bits)	
destination page offset and length	notification
command (WRITE)	(padding)
small-message payload (up to 64 bytes)	

Table 10. Format of the write-command descriptor.

The NetVM user library handles the read-write API calls by constructing command descriptors and issuing them into the network interface. Table 10 shows the **write**-command descriptor format. Each field is a 32-bit word except for the Myrinet route and protection keys, which are 64 bits each. The user library assigns only the shaded fields, which are specific to the write operation; the network interface assigns the unshaded system-dependent fields.

The user library splits a large application request into multiple smaller transfer commands for the network interface. The read-write API calls specify local and remote virtual address ranges, which may span multiple pages and may be unaligned. However, the network interface handles only up to page-sized page-aligned transfers, no transfer can cross either a local or remote page boundary. The user library fragments a large unaligned transfer into multiple smaller page-granular units and constructs a command descriptor for each unit. It computes the subsequent VPN, VPN hash key, page offset and length fields of the source and destination pages in each descriptor.

Each descriptor also specifies the protection keys for the source and destination pages so that the NI can access those host pages. To obtain the keys, the user library looks up its export map to retrieve the key for the local page and looks up its import map to retrieve the key for the remote page. Section 4.1 on page 43 described the import-export maps in detail.

The **write**-command descriptor includes a 96-byte small-message payload. The user library copies the data for small transfers, using programmed IO, directly into the payload field instead of relying on the host-DMA mechanism on the network interface. The network-processor overhead for setting up the host-DMA operation for a small data transfer, including address translation,

page locking and DMA-engine setup, exceeds the host-processor overhead of writing the same data directly into network-interface memory. The **read**-command descriptor does not include a payload field, because it is the network interface, and not the application, on the remote node that transfers the data from host memory into network interface memory.

5.3 Application command queuing and dispatching

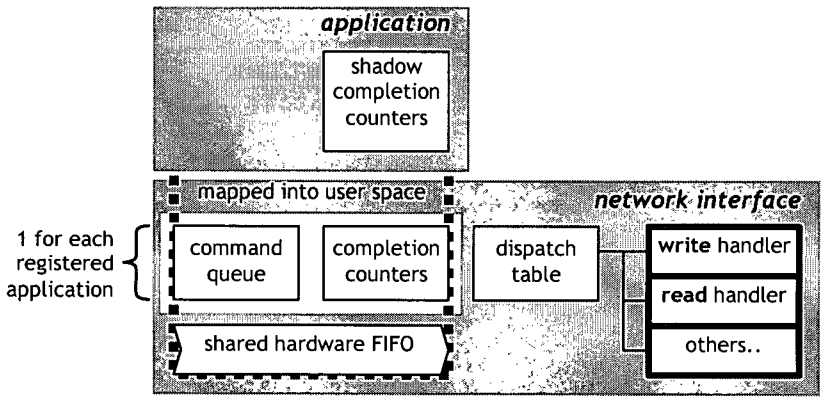

Figure 14. Application-NI interface and command dispatching.

Figure 14 shows the key data structures for applications to issue command descriptors directly into the network interface, and for the firmware to dispatch them to the respective command handlers. NetVM allocates a private page-aligned region of network interface memory for each application to store the command queue and a set of completion counters. It also maps shared the write-only hardware FIFO that all applications use to signal the NI whenever they insert an entry into their private command queues. The dispatch table stores function pointers to the command handlers, which allows the NI to quickly dispatch a command descriptor to its respective handler. The application also stores a set of shadow completion counters, which the NI directly updates, to enable flow control over the command queue.

An application issues a command to the network interface by first inserting the descriptor into its command queue and then writing its NetVM port number into the shared hardware FIFO. The NI polls this FIFO to determine the command queue that has a ready descriptor. Without this FIFO, the NI has to repeatedly scan the queues of all registered applications, which is slow when there are many NetVM processes on the node. The NI uses the command field in the descriptor as an index into the dispatch table to obtain the address to its associated handler. Finally, the NI calls the handler to process the command descriptor.

5.3.1 Flow control over the command queue

NetVM maintains a stepping window, using the completion counters, to provide flow control over the 64-entry command queue. Each time a handler completes a command in the queue, it increments a completion counter associated with that queue on the network interface. After every 32 completions, the NI transfers the completion counter value, via host DMA, to a shadow counter in the application host memory, thus freeing up 32 entries in the queue. The application tracks the total number of commands it has issued and polls this shadow counter to determine if the queue is full. It spins on the shadow counter, if necessary, until entries are available. This flow-control scheme allows an application to issue at least 32, and up to 64, commands into the queue whenever the NI updates the shadow counter. Using a step size of 32, instead of one, amortizes the cost of the host-DMA updates while freeing up sufficient entries in the queue on each update.

The application spins on the shadow completion counter in host memory because it is inefficient to spin on the counter in network-interface memory. The host processor requires an expensive IO-bus transaction for each poll on network-interface memory. The network processor also stores its working data and executes its firmware code off the same memory. Using the host processor to spin on network-interface memory ties up the IO bus and slows down the network processor due to increased memory-bus contention on the network interface. Having the host processor spin on host memory avoids tying up either the IO or the host memory bus. Host-DMA transfers from the network interface are cache coherent. The host processor spins on a cached copy of the shadow counter until the NI invalidates the associated cache line by updating the counter in host memory via host DMA. The next read after the update causes the memory controller to fetch the new value from the host memory.

5.4 Write operation on the network interfaces

Figure 15 on the following page shows the **write** operation on the source network interface. The NI **write**-command handler uses the VPN and VPN hash key fields in the descriptor to look up the source page in the shadow page table and obtain its physical address (PPN) and protection key, and to lock the page in host memory. Section 4.3.3 described this address translation and page locking operation in detail. After matching the protection key from the PMAP with the key in the descriptor, the NI initiates a host-DMA transfer from the source page at host physical address PPN into the DMA staging buffer. At the same, it also transfers the **write** message header, by wire-DMA, onto the wire after updating the system-dependent fields (e.g. route and sequence number) in the header. When the host-DMA transfer completes, the NI appends the

message header with the source data by transferring it, also by wire DMA, from the staging area onto the wire. Finally, it unlocks the page in the shadow page table and, if necessary, notifies the application of the local completion of this **write** command.

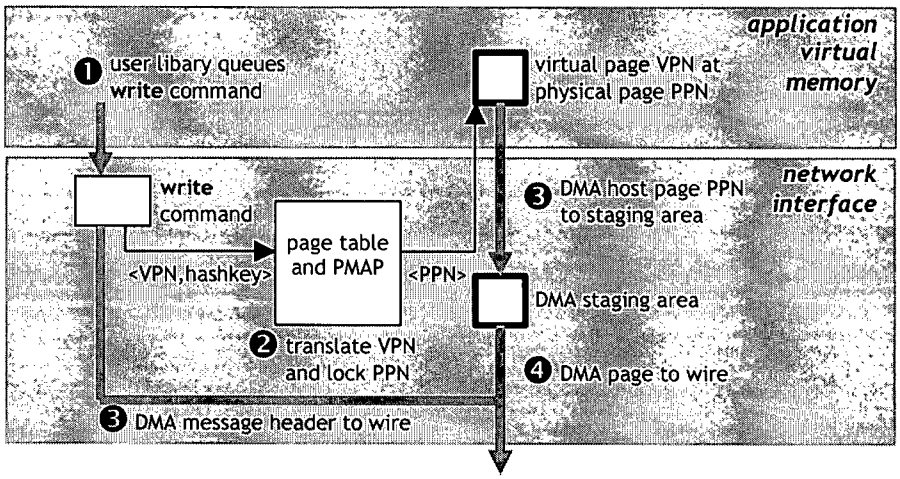

Figure 15. Operation of RDMA-write on source network interface.

The NI interrupts the kernel driver if cannot locate the source page mapping in the page table. The kernel driver responds by first fetching the nonresident source page from the backing store and inserting its page mapping into the shadow page table. It then restarts the **write** operation on the network interface. This case is rare, because the application touches each source page, causing a page fault if necessary, immediately before issuing the **write** command for it.

After the **write** operation completes, the NI increments the write-completion counter in the network interface and, if requested by the application, updates a shadow counter in host memory via host DMA. The application spins on this shadow counter to determine when the operation completes and, more importantly, when it can reuse the source buffer. The application requests for this notification by incrementing a private write-completion counter and writing the required count into the network interface. The NI notifies the application only when the counters match. For a multi-page **write** operation, the application only needs to request the notification for the last **write** command in the group, because NetVM processes a command queue in FIFO order. This **write**-completion notification mechanism therefore provides source-side write fencing because the completion of **write** *j*, with respect to the sending application, also implies the completion of **write** *i* for all $i < j$ in time.

NetVM uses the small-message-payload field in the descriptor for small messages. For messages that are less or equal to 96 bytes, the user library copies the data, using programmed IO, di-

rectly into the payload field. Doing so, especially for small transfers, avoids the higher overhead from the address translation, page locking and host-DMA setup operations required on the source network interface. Section 3.4.1 on page 41 shows this trade off between using programmed IO and host DMA for transferring data for various sizes, between 4 bytes and 4KB, from host memory into the network interface.

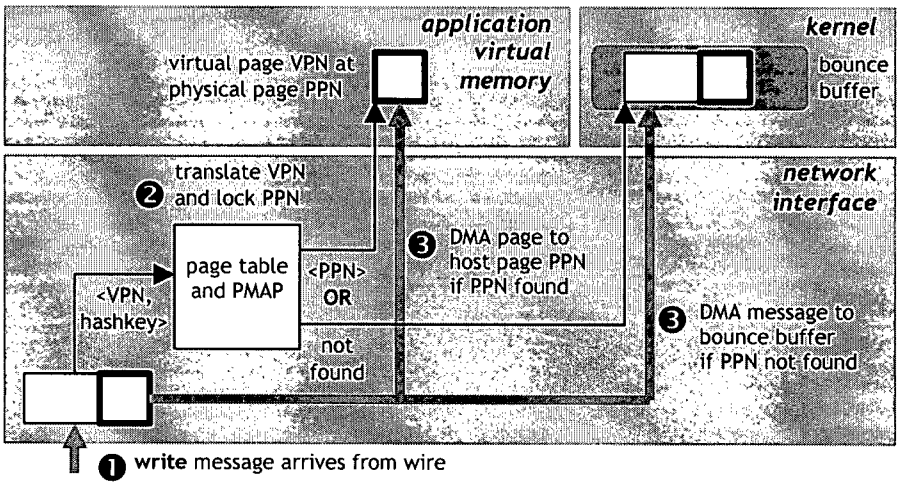

Figure 16. Operation of write on destination network interface.

Figure 16 shows the **write** operation on the destination network interface. The NI starts examining the message header once it receives sufficient data from the wire. It uses the VPN and VPN hash-key fields in the header to translate the address of and lock the page in host memory. After matching the protection keys and waiting for the message payload to completely transfer into the network interface from the wire, the NI initiates a host-DMA transfer from the network interface into the host page. When the DMA transfer completes, the NI marks the page as dirty and unlocks it in the shadow page table. If the **write** operation requires notifying the receiving application, the NI transfers a notification record to the application notification queue and optionally interrupts the host to signal the application. Section 6.1 describes this notification mechanism in detail.

The NI transfers the entire message into the bounce buffer and interrupts the kernel if it cannot locate the destination page mapping in the page table. The kernel driver fetches the non-resident destination page from the backing store and transfers the data, on behalf of the NI, from the bounce buffer into the application page.

The NI aborts the transfer if the protection checks fail. The **write**-command descriptor carries the protection keys for both source and destination pages. The NI compares the keys with those in the shadow page table after successfully locking and obtaining the PMAP entry for the page. These checks ensure that the application cannot inadvertently access a remote segment that it did not import. Using 64-bit wide protection keys also prevents an application from easily forging a key in the command descriptor and thereby gaining unauthorized access to a remote memory segment.

5.5 Read operation on the network interfaces

read-command Descriptor	
Myrinet wormhole route (8 bytes)	
Myrinet message tag (NETVM)	(padding)
sending node	sending node sequence number
destination port	destination node
destination VPN	destination VPN hash key
destination protection key (64 bits)	
destination page offset and length	source node
source VPN	source VPN hash key
source protection key (64 bits)	
source page offset and length	(padding)
command (READ)	(padding)

Table 11. Format of the read-command descriptor.

Table 11 shows the RDMA-read command descriptor format. The **read** operation transfers data from the remote source memory location to the local destination address. Like the **write** operation, the application initiates a **read** request by constructing the **read**-command descriptor, inserting it into the command queue, and signaling the shared hardware FIFO.

The **read** operation is similar to **write**. The requesting NI first forwards the entire **read** message to the remote source node. The source NI processes the message and replies to the destination node with the requested data, similar to the **write** operation on the source network interface, without any host intervention as long as the target page is resident. If the page not resident, the NI redirects the **read** request to the kernel, which will fetch the page and restart the reply transfer. When the reply message arrives at the destination node, the NI transfers the data directly to the destination page, similar to the **write** operation on the destination network interface. Again, if the page is not resident, the NI redirects the data to the bounce buffer and interrupts the kernel, which will fetch the page and complete the transfer.

After the **read** operation completes on the destination node, the NI increments the read-completion counter in the network interface and, if requested by the application, also updates a shadow counter in host memory. This read-fencing mechanism allows the application to spin on the shadow counter to determine when all active **read** operations have completed, even if they complete out of order. The application requests this notification by incrementing a private completion counter whenever it issues a **read** command and updating the requested count value into the network interface after the final issue. The NI updates the application shadow counter only when the requested count value matches read-completion counter.

The **read** operation provides the same protection mechanisms as **write** to prevent inadvertent and unauthorized access to a remote memory segment.

5.6 Bounce buffer

The NetVM kernel driver in each node maintains a system bounce buffer, which is a physically contiguous and pinned circular queue in host memory. The NI redirects **write** and **read** operations that refer to nonresident application pages to the bounce buffer as described in the previous two sections. Therefore, assuming reliable network hardware, NetVM avoids the need to buffer messages for retransmission due to overruns on the receiving network interface. A credit-based flow-control mechanism protects the bounce buffer itself from buffer overruns.

During initialization, NetVM partitions its bounce buffer among all the other nodes in the system. Before sending a message that potentially uses the bounce buffer, the NI checks that it has sufficient credits for the remote node and consumes a credit unit after sending the message. Without sufficient credits, the NI delays the message until it receives new credit replenishments from the remote node. This delay does not affect other transfers from the sending node to other destination nodes that have sufficient available credits. The destination NI immediately frees the credit for a message that bypasses the bounce buffer. Otherwise, the kernel driver frees the credit, for the NI, once it processes the message in the bounce buffer. The driver interrupt handler wakes up a separate kernel paging thread to scan the bounce buffer, perform the paging, and transfer the data, so that the interrupt handler can process other nonblocking interrupt operations concurrently with the paging thread. Once the destination NI accumulates sufficient freed credits, it sends a replenishment message back to the sending NI. This credit-based scheme also integrates with the delivery-order scheme, which provides ordering guarantees for notification delivery, described in Section 6.2 on page 78.

The **read** operation checks for bounce buffer credits at two places: on the requesting NI and on the remote NI. The remote source page or the local destination page, or even both pages, may not be resident in host memory.

The requesting NI needs a bounce buffer credit for the remote node in case the remote NI has to redirect the **read** request to the bounce buffer because the source page is not resident. Similarly, the remote NI needs a bounce buffer credit for the requesting node in case the local NI has to redirect the **read** reply to bounce buffer because the destination page is not resident. The second case is rare, however, because the requesting application touches each destination page, thus setting its referenced bit in the host page-table entry, before issuing the read-command descriptor for that page. Setting the page's referenced bit marks it as recently used, which significantly increases the likelihood that the page is resident when the **read** reply arrives. If the remote NI has insufficient credits for the bounce buffer on the requesting node, it redirects the **read** reply to its kernel driver, which will defer the reply operation until the requesting node replenishes it with enough credits.

5.6.1 Scalability of the bounce-buffer mechanism

NetVM currently tracks credits only at the node level. Applications on the same node that send messages to a common remote node consume credits from a shared pool. On the one hand, managing credits at the node level limits the amount of bounce buffer space in host memory, and metadata in network memory, to $O(N)$ where N is the number of nodes in the system. On the other hand, it is possible for a single application to stall other applications that are sharing its credits if it repeatedly accesses nonresident remote pages. The remote kernel has to fetch the accessed pages from the backing store and thus requires a much longer time to complete the transfer and release the credits. Using a per-application credit allocation scheme addresses this sharing problem. However, it will also increase the memory required for the bounce buffer on each node to a nonscalable $O(NP^2)$, where P is the number of NetVM processes per node.

5.7 Summary

Data-transfer operations originate in the application. The user library exports a data-transfer API for NetVM applications to access by interacting directly with the network interface. The user library manages the import-export tables to verify address ranges and to obtain the protection keys for each transfer. It also reformats large transfer requests into page-sized fragments for the network interface. Multiple applications interact with the network through the doorbell mechanism, which multiplexes application requests into a single hardware FIFO for the

firmware to process. Each application also maintains a stepping window for flow control over its own command queue.

The network interface handles all data transfer requests directly from applications. For **write** operations, it adaptively switches between programmed IO for small transfers and DMA for larger transfers above 96 bytes. For **read** operations, it forwards the request to the remote network interface, which uses DMA to fetch data from host memory. To access a page using DMA, the NI looks up the shadow page table and the PMAP to lock the page and obtain its physical address. The NI maintains completion counters for each application to support fenced reads and source-side fenced writes.

If a destination page is not resident, the NI redirects the entire transfer into a system bounce buffer in host memory and interrupts the kernel to complete the operation. The interrupt handler fetches the required page and copies the data from the bounce buffer to the target page. NetVM implements a credit-based flow-control scheme for the bounce buffer to avoid the need to buffer messages for retransmission due to overruns on the receiving network interface.

6 Control-transfer Operations

The NetVM notification mechanism allows an application to transfer control to a remote application. The basic **write** operation transfers only data; the receiving application cannot tell when that data arrives into its memory. A sending application additionally notifies a remote application by including a nonzero notification number in a **write[F]N** operation. The receiving application detects the notification synchronously by polling, spinning on host memory, or block waiting for a notifying write to complete. Alternatively, it can arm a local procedure to automatically execute whenever a notifying write completes.

An out-of-order delivery network may deliver fragments of a large transfer in a different order that they originated from the sending node. The NetVM delivery-order semantics defines the relationship between notifying and nonnotifying transfers. The semantics guarantee that an application receives a notification for a transfer after it receives *all fragments* of the transfer, even if they arrive out of order. Furthermore, the semantics also guarantee that the application receives that notification after all previous transfers *from the same sender* have also arrived. To do this, the NI maintains a sequence window in the network interface to track message packets that have arrived from a sending node. The sending NI allocates a slot in the remote sequence window each time it transmits a packet to that node. The receiving NI marks the slot when it receives the packet, but defers any associated notification until all preceding packets have arrived. To provide flow control, the receiving NI recycles portions of the window to the sending NI once it marks and frees sufficient slots.

6.1 Notification

NetVM notifications have event-counter semantics. An application maintains a set of notification objects. Each object, which is identified by a notification number, includes a *signal* count and an *acknowledge* count. A sending application signals an event, which increments the signal count, by specifying the notification number in the **write*N** operation. A receiving application finds a pending signal if the signal count exceeds the acknowledge count. It handles and acknowledges an event by incrementing the acknowledge count.

The application detects notifications in four possible ways. First, it can directly poll the signal and acknowledge counters to compute the number of pending signals for a particular notification number. Second, it can direct the network interface to update a shadow signal counter in host memory each time it receives a notification. The application spins on this shadow counter until the notification arrives. Third, it can direct the network interface to interrupt the operating system each time it receives a notification. The application blocks waiting for a wakeup from the operating system. Finally, it can register a notification-handling procedure that will execute whenever a notifying transfer completes.

A NetVM application allocates a physically contiguous circular *notification queue* in host memory when it opens a NetVM port. This queue multiplexes signals with different notification numbers into a single pipe to the application. Thus, the user library needs to only test the head of the queue to determine which notification has arrived. NetVM bounds the size of the notification queue by coalescing multiple events with the same notification identifier into a single entry in the queue. The user library compares the *signal* and *acknowledge* counts to determine the number of pending signals for that notification identifier in the queue.

6.1.1 Notification API

Type	Operation	Return	Description
Notification queue	notfQRemove() notfQWait(timeout)	notf or EMPTY notf or TIMEOUT	dequeue notification entry wait to dequeue notification entry
Notification handlers	notfArm(notf, handler) notfDisarm(notf)	- -	register a notification handler deregister a notification handler
Synchronous operations	notfTest(notf) notfSpin(notf, timeout) notfWait(notf, timeout) notfAck(notf)	number pending SUCCESS or TIMEOUT SUCCESS or TIMEOUT -	test for pending notifications spin until notification arrives wait until notification arrives acknowledge a notification

Table 12. NetVM notification handling operations.

Table 12 shows the operations that NetVM applications use to handle notifications in three sections. The first section lists the operations for the notification queue. **notfQRemove** dequeues and returns the first available notification-number entry from the notification queue, or returns EMPTY if there are none. **notfQWait** blocks and waits until the notification queue has an entry. It returns the first notification-number entry in the queue, or TIMEOUT if the application did not receive any notifications.

The second section lists the operations to register and deregister a notification handler. **notfArm** specifies the notification number and application-defined procedure that will handle the

notification when it arrives. Once armed, NetVM calls the handler, exactly once, for each notifying data transfer with a matching notification number. If pending signals already exist during the arming operation, **notfArm** immediately calls the handler to process each of those signals. **notfDisarm** simply unbinds the registered notification-handler procedure with the given notification number.

The third section lists the operations that synchronously test for and handle notifications. **notfTest** polls the signal and acknowledge counts for the specified notification number to determine if there are pending signals. **notfSpin** spins on host memory until a new notification arrives. The application calls **notfAck** to acknowledge the notification after processing it.

6.1.2 Data structures and operations

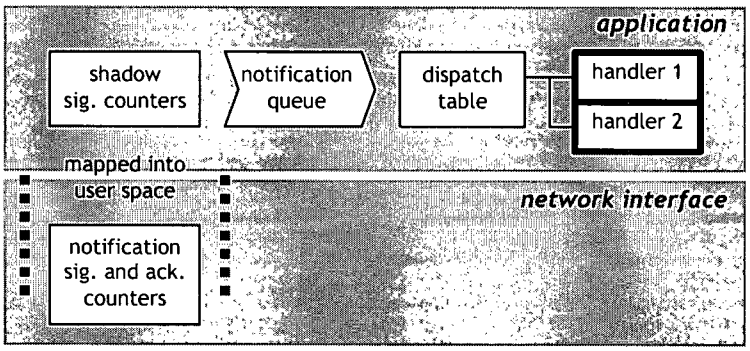

Figure 17. Data structures and function handlers for the notification mechanism.

Figure 17 shows the notification data structures in both network interface and application memory. The NI memory stores a pair of *signal (sig.)* and *acknowledge (ack.)* counters for each of the 1023 notification numbers available to each registered application on the local node. The application memory stores shadow *signal* counters and the *notification queue* that the NI can directly access. It also stores a notification-handler *dispatch table* that binds notification numbers to their registered *handler* procedures. To support concurrent access to the notification counters, only the NI updates the *signal* counters and only the application updates the *acknowledge* counters on the network interface.

NI write handler	Application notification dispatcher
notify[ID] = notify[ID] + 1 *if* notify[ID] == acknowledge[ID] + 1 *enqueue*(notfQ, ID) *interrupt host*	*while !empty*(notfQ) ID = *dequeue*(notfQ) *while* acknowledge[ID] < notify[ID] *call* handler[ID] acknowledge[ID] = acknowledge[ID] + 1

Table 13. NI and application notification dispatch implementation.

The application arms a notification by registering a user-defined handler with the dispatch table and setting an *interrupt-enable* flag for the specified notification number on the network interface. Table 13 shows the operations for the NI to dispatch a notification to the application. When the NI receives a message carrying a notification number that is armed, it increments the *signal* counter and compares it with the *acknowledge* counter. If the *signal* count exceeds the *acknowledge* count by *exactly* one, the NI adds a notification-number entry, using host DMA, into the notification queue and interrupts the kernel driver. Subsequent arriving notifications will only increment the *signal* counter, without interrupting the host, as long as there are pending signals that the application has not acknowledged. Doing so will coalesce multiple arrivals of the same notification number into a single queue entry. The kernel interrupt handler signals the application notification dispatcher, which is a SIGUSR2 signal handler that the NetVM user library registers during its initialization. This dispatcher dequeues the notification-number entry from the notification queue and uses it to look up the dispatch table. The dispatcher calls the handler exactly once for each pending signal by incrementing the acknowledge count after each call until the *signal* and *acknowledge* counters match.

A race condition exists that can cause both the application arming operation and the NI to trigger the handler for the same single notification signal. An application arms the handler by first setting the interrupt-enable flag on the network interface and then checking for any pending signal for that notification number. It immediately calls the handler if pending signals exist instead of relying on the notification dispatcher, because the NI will interrupt the host only for *new* signals, when the signal count is exactly one greater than the acknowledge count, *and* only *after* the interrupt-enable flag is set by the arming operation. However, a race occurs when the NI receives a notifying message immediately after the application sets the interrupt-enable flag, but before the application tests for pending signals. In this case, the application will find a pending signal and directly call the handler, and the NI will find a new, interrupt-enabled, notification and invoke the dispatcher to call the handler too. NetVM ensures exactly-once handler-calling semantics through two ways. First, the arming operation delays the dispatcher until it completes calling the handler for any pending events. Second, shown in Table 13, the dispatcher verifies that the *signal* count exceeds the *acknowledge* count each time before calling

the handler. Therefore, it avoids calling the handler if the arming operation has already acknowledged any pending signals.

NetVM relies on the BSD signal-handling semantics to simplify its notification mechanism. In particular, the operating system blocks, but does not drop, new signals when the application is busy executing the signal handler. There are two benefits to this behavior. First, notification handlers can be nonreentrant because the operating system will not recursively invoke a signal handler and, hence, the dispatcher can not recursively call notification handlers. Second, the signal handler will not lose any notifications from the NI. A race condition exists when the NI posts a signal just after the dispatcher finds no entry in the notification FIFO, but just before the signal handler exits. This second notification will be lost if the operating system drops the NI-posted signal. However, the operating system, with BSD signal-handling semantics, simply invokes the signal handler again to restart the dispatcher. If the operating system uses System V signal-handling semantics, the signal handler first needs to reinitialize the signal vector each time it executes to avoiding dropping new signals. It also needs to handle recursive signal invocations by ensuring that it does not recursively call the notification handlers.

The NetVM signal handler should not interrupt critical sections in nonreentrant operations that both the main application thread and notification handler use. An example is the **write** operation; a notification handler that calls **write** in the middle of an active **write** by the interrupted application thread may corrupt the command queue. A notification handler calling these nonreentrant operations does not need to synchronize because its execution is uninterruptible by the application thread. However, the application thread must synchronize with the notification handlers, by delaying the signal handling, while it is in a critical section. One approach is to temporarily disable the dispatcher by blocking SIGUSR2 signals with the **sigsetmask** system call. NetVM uses an alternative optimistic approach (assuming that signals occur much less frequently than critical sections) that avoids the system call overhead by incrementing a shared lock-count variable when the application thread enters a critical section and decrementing the variable when it leaves. If the SIGUSR2 signal handler executes while the application thread is in a critical section, it finds a nonzero lock count and defers its execution by setting a signal-pending flag and then exiting immediately. When the application thread finally exits the critical section with a zero lock-count and finds a set signal-pending flag, it explicitly restarts the signal handler.

6.1.3 Synchronous notification detection

An application synchronously detects notifications using the API calls listed in the third section of Table 12 on page 73. **notfTest** computes the number of pending signals for the specified notification number by subtracting the *acknowledge* count from the *signal* count after reading them from the network interface using programmed IO. **notfSpin** first directs the NI to update the *shadow signal* counter in application memory, via host DMA, each time it updates the *signal* counter in network interface memory. **notfSpin** then spins on this *shadow counter* until the NI updates it with the new value when a notification arrives. **notfWait** sets the interrupt-enable flag in the network interface and makes a system call to block waiting for the notification to arrive. When the NI receives the notification, it interrupts the kernel, which wakes up the application. Finally, **notfAck** increments the *acknowledge* count by one to indicate to the NI that the application has processed a single occurrence of the signal.

6.1.4 One-shot notifications

NetVM implements a second simple *one-shot* notification mechanism. This scheme utilizes the notification queue to receive ordered notifying single-word messages from a remote application. The notifying application specifies a notification number of 1024 or above in the **write** operation. The receiving NI adds the entry, for each one-shot notification that it receives, into the notification queue. Thus, multiple notifications, even with the same number, result in multiple entries in the queue. Therefore, the remote applications must implement flow control with the receiving application to ensure that they do not overflow its notification queue. An application can also control the size of its notification queue by specifying the minimum number of entries when it opens a NetVM port.

There is a tradeoff between using event-counting notifications and one-shot notifications. Event-counting notifications are useful for resource management, where applications need to track a large number of events in a specific class that have occurred. An example is the producer-consumer problem: a sending application signals an event each time it produces a data unit and the receiving application reads the event counters to determine when and how many units are ready from the sender. However, the total number of such notifications is limited, because NetVM needs $O(nP)$ storage in network interface memory, where n is the number of notifications per NetVM process and P is the number of NetVM processes per node. NetVM currently supports only 1023 event-counting notifications per application due to the limited network-interface memory. One-shot notifications do not require any storage on the network interface. However, they do require a word entry in the notification queue for every notifying write from the remote application. They are useful for applications to synchronize with each

other by signaling with named events, because the name space for one-shot notifications is large, which is from 1024 to 2^{32}-1.

6.2 Delivery order

With an out-of-order delivery network, it is insufficient for the sending node to attach the notification number to the final message of a large fragmented transfer and for the receiving NI to deliver the notification on receiving that message. The final message may arrive before the other associated messages in the same transfer group. NetVM ensures that the application receives the notification only after it receives all the fragments in the group. The application can therefore safely access the preceding sent data without the need to explicitly check that the entire group has, in fact, arrived. NetVM also ensures that the application receives notifications in the same order that the sending application issues them. This ordered-notification delivery simplifies an application's reasoning about multiple transfers in pipelined operations such as data streaming.

NetVM transfers data and control in a notifying **write** operation by including the payload, a notification number, or both, in a **write** message. The delivery semantics of NetVM specify that, if a sending application issues two notifying writes (i and j) with the same notification number to a receiving application, NetVM will deliver the notification for **write** j after it

1. delivers the data for **write** i for all $i \leq j$ and
2. delivers the notification for **write** i for all $i < j$.

In other words, *NetVM delivers the notification for a **write** only after it has delivered the notification and data for all preceding **writes** from the same sender*. The first requirement specifies that NetVM delivers a notification only after it has delivered the data from the current and all previous **write** transfers. The second requirement specifies that NetVM delivers a notification only after it has delivered all previous *same*-numbered notifications. The delivery order between two *different*-numbered notifications, even from the same sender, is undefined. Within the constraints of both requirements, NetVM does not control the delivery order for data, which depends only on the underlying network.

These semantics allow a receiving application to determine when multiple **write** transfers that belong to a group have all completed. There are three cases where ordered grouped notifications are important. The first case is during a single large transfer; an application issues a large multi-page notifying **writeN** to a remote application. The NetVM user library splits this large

transfer into page-sized fragments and sends a **write** message for all but the last fragment, and sends a **writeN** message for the last fragment. The remote NI notifies the receiving application only after delivering all the fragments for the transfer. The second case is during multiple small scattered transfers; an application issues multiple **writes** to scattered locations in the remote application and notifies it only after issuing the last **writeN**. NetVM notifies the receiving application only after all the **write** transfers have all arrived to their respective destinations, even if they arrived out of order. The third case is during multiple large transfers in a pipeline; an application issues a series of large, possibly multi-page, notifying transfers to a remote application in a pipeline. NetVM delivers each notification, after transferring the associated data, *in the same order* that the sender streams them. The receiving application relies on these ordered notifications to expect the next ready data in the pipeline whenever it receives a new notification.

6.2.1 Sequence windows

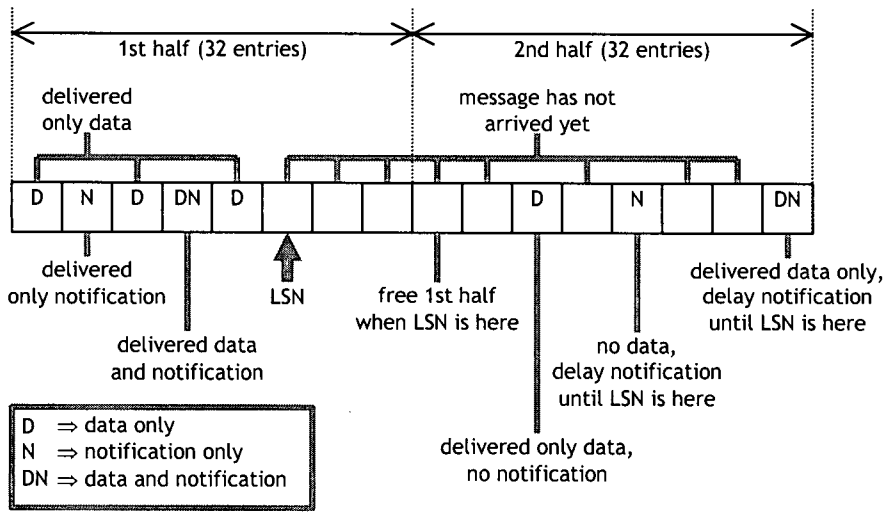

Figure 18. Sequence window example that tracks arriving messages from one remote node.

Shown in Figure 18, NetVM maintains a 64-entry circular sequence window in the network interface for each node in the system. This window tracks arriving messages from the remote node to implement the delivery-ordering mechanism. The sending node maintains a monotonically increasing sequence number, for each remote node, that corresponds to an entry in the sequence window. The receiving node uses this sequence number and the **write** message type (D for data only, N for notification only, or DN for both) to record a flag into the corresponding entry in the sequence window. A Lowest Sequence Number (LSN) cursor tracks the minimum sequence number of messages that have not yet arrived at the node. When the LSN crosses into

the second half of the window, the NI frees the first half by sending an acknowledgement to the sending node, thus allowing it to reuse the first half of the window for new messages. Similarly, the NI frees the second half of the window when the LSN wraps around into the first half.

NetVM delivers data in message arrival order but delivers notifications in increasing sequence-number order. The NI immediately delivers data in a **write** message regardless of the message's position in the sequence window. It delivers the notification in a message only if its position matches the LSN. In either case, the NI increments the LSN if it matches the received message's sequence number. If the NI redirected the message to the bounce buffer, the kernel driver updates the flags, on behalf of the NI, for that window entry once it fetches the required page and completes the transfer from the bounce buffer.

There is a tradeoff between imposing too much ordering, which slows throughput due to the flow-control overheads in buffering and acknowledgements, and imposing too little ordering, which increases complexity for applications that need to reason about the delivery order. NetVM attempts to strike a balance with three levels of ordering: enforcing total order for notifications with the same notification number, partial order for notifications with respect to data by delivering the notification after delivering the associated data, and no order for data by delivering it the moment it arrives into network interface from the wire. The current bounce buffer mechanism already avoids the need to buffer data on the sender for flow control. The ordering mechanism preserves this feature by buffering only a small amount of state information to provide ordered deliveries for notifications. It also reduces the number of acknowledgement messages for flow control, without stalling the sender, by maintaining a sufficiently large sequence window and sending an acknowledgement only after freeing half the window.

6.2.2 Notification reordering

NetVM does not order event-counting notifications that specify *different* notification numbers, even if they originate from the same application. Although the sequence-window mechanism guarantees that NetVM delivers notifications to the host in increasing LSN order, the notification queue mechanism coalesces multiple same-numbered notifications into a single entry. Therefore, if a pending notification already exists, the NI appends the new notification by simply incrementing the signal count, without adding a new entry into the queue. This notification coalescing, in effect, may allow a new notification to overtake an earlier, but different-numbered, notification waiting in the queue. The notification dispatcher repeatedly calls the handler until there are no more pending signals for the associated notification number before it examines the next entry in the queue to dispatch new notifications.

One-shot notifications do not suffer from this notification-reordering problem. NetVM does not coalesce one-shot notifications, but adds an entry into the notification queue *each time* it receives a notifying message. The dispatcher finds these notifications in the same order that they arrive into the queue. Therefore, NetVM delivers one-shot notifications in strict issue order from the sender. This one-to-one mapping between incoming one-shot notifications and entries in the notification queue imposes a constraint on remote applications; they must provide flow control with the notified application to ensure that they do not overflow its notification queue.

6.2.3 Integration with the bounce buffer

The delivery-order and bounce-buffer mechanisms are integrated. Each entry in the sequence window corresponds to a credit in the bounce buffer. Therefore, a sending node that has an available entry in the sequence window on the remote node also has a credit for that message. Consequently, an acknowledgement message freeing up half the sequence window also replenishes 32 credit units.

This integration also means that NetVM does not require more than 64 credit units of bounce-buffer space for each remote node, because the sequence window prevents the sending node from issuing more than 64 messages without receiving an acknowledgement message. Reducing the sequence window size reduces the number of bounce buffer credits needed on the host at the expense of increasing the acknowledgement rate.

6.2.4 Scalability of the delivery-ordering mechanism

The delivery-ordering mechanism operates at the node level. Applications on a node that send data to a common remote node also share the sequence number allocation. This sharing does not allow a slow sending application to affect others on the same node, because NetVM only allocates the sending sequence number after the message, already on the network interface, is ready to transfer onto the wire. However, an application can stall other local applications sending to the same remote node, due to the node-level bounce-buffer mechanism, if it repeatedly accesses nonresident remote pages. These nonresident transfers cause the kernel driver to take a longer time to fetch the page, complete the transfer, and recycle the bounce-buffer credit by updating the sequence window.

In the worst case, an application can stall other local applications sending to the same remote node if it accesses just one nonresident page even if the others are busy sending data to only resident pages. The sequence window mechanism only allows at least 32, and up to 63, trans-

fers to complete while waiting for the kernel to complete the transfer to the nonresident page. After that, the sending NI defers sending new messages to the remote node until the remote NI recycles the half window containing the sequence number for the nonresident transfer and replies with an acknowledgement message. In the meantime, it can still send a message to all other nodes that have available sequence-window entries.

Similar to the bounce-buffer sharing issue, using a per-application sequence window will decouple the delivery dependence between applications. Rather than storing flow-control state for each pair of nodes, an alternative is to require each pair of applications to store the state instead. The tradeoff in maintaining application-level state is that the sequence windows requires a nonscalable $O(NP^2)$ storage on the network interface, where N is the number of nodes in the system and P is the number of NetVM processes per node. One way to address the storage problem is to *cache* active sequence windows on the network interface, assuming that the host has sufficient memory to store every sequence window for every NetVM application in the system. The NI uses the host memory as a backing store and caches only active windows in network interface memory. The cost of this approach is that the NI may have to fetch the 256-byte sequence window from host memory in the critical path of an incoming message. Overlapping the host-DMA transfer for the miss handling with the wire-DMA transfer for a large message payload mitigates this cost.

An alternate scheme for group notification in Hamlyn [12] uses an accumulator to track the number of arriving fragments in a group. All but the last fragment in the group carry a count value of -1 and the last one carries a count value of N-1, where N is the number of fragments in the group. The receiver accumulates these count values as it receives the fragments and finds that the entire group has arrived when the accumulator sums to zero. This scheme, although efficient and elegant, has three disadvantages. First, each individual sending application has to generate these count values. Therefore, the receiving network interface has to maintain, for every local application, an accumulator for *every remote application* in the system, requiring a nonscalable $O(NP^2)$ storage space on the network interface to store these accumulators. Second, the network interface must store multiple accumulators *per remote application* if the sending application needs to consecutively issue multiple groups. In any case, it has to wait for an acknowledgement for an accumulator before reusing it for a new group. Third, it is not possible to order group notifications. A notification occurs only when all the fragments in that group arrives, independent of the arrival of fragments in other groups. Thus, an application cannot raise a sequence of ordered events at the remote destination.

6.3 Summary

NetVM uses a scheme based on eventcounts to allow an application to signal events to a remote application, by including a notification number in a data-transfer operation. The user-library API allows the remote application to detect these events by busy waiting, block waiting, or triggering a user-defined handler to process them. Synchronous notification detection is straightforward; the application spins on host memory or blocks in the kernel, waiting for the event to occur. Dispatching notifications to registered handlers, however, is slightly more complicated. The library maintains a notification queue and a SIGUSR2 dispatcher signal handler to process signals destined for registered handlers. NetVM ensures mutual exclusion between the application thread and signaled handlers by delaying notification dispatching whenever the application thread is in a critical section updating the command queue.

To support out-of-order delivery networks, NetVM implements a sequence window on the network interface to track arriving synchronized and unsynchronized transfer messages. It only delivers the notification for a **write** only after it has delivered the notification and data for all preceding **writes** from the same sender. Thus, NetVM delivers notifications in the same order that the sending application issues them. To avoid the need for data buffering, the network interface delivers data in the order that it receives them from the network, regardless of the order that they were sent.

The next chapter describes a user-level implementation for channels, which provide stream-oriented communication between two applications, using only the NetVM API.

7 Channels

A receiving application cannot determine the target location of a NetVM RDMA **write** operation. It can only determine when the transfer completes using the notification mechanism. The stream-oriented send-receive model, however, allows the receiving application to control where data should arrive. In addition, it can explicitly wait for and receive data in FIFO order from the sender.

NetVM implements *channels* to support stream-oriented communication. The receiving application exports a message queue that the sending application directly accesses and updates whenever it sends a message through the channel. NetVM channels are not receiver-managed; the receiving application cannot specify the exact location of the arriving message. Instead, it obtains the address of the message in the queue on returning from a successful receive operation. It then either processes the message in place, or copies it to a new location for post processing. NetVM uses event-count notifications to implement flow control over the channel.

Flow control over a NetVM channel is based on the producer-consumer model. Both sender and receiver track the number of free, sent (produced) and received (consumed) messages in the message queue to ensure that it does not overflow or underflow. The receiver blocks during a *receive* operation until the queue is not empty. It refills the sender each time after consuming sufficient messages in the queue. The sender blocks during a *send* operation, waiting for a refill from the receiver, if the queue is full.

NetVM implements channels at the user level. It transfers data to the application message queue using the notifying **writeN** operation and implements flow control using two event-count notifications. Figure 19 on the following page shows the key data structures required to implement channels.

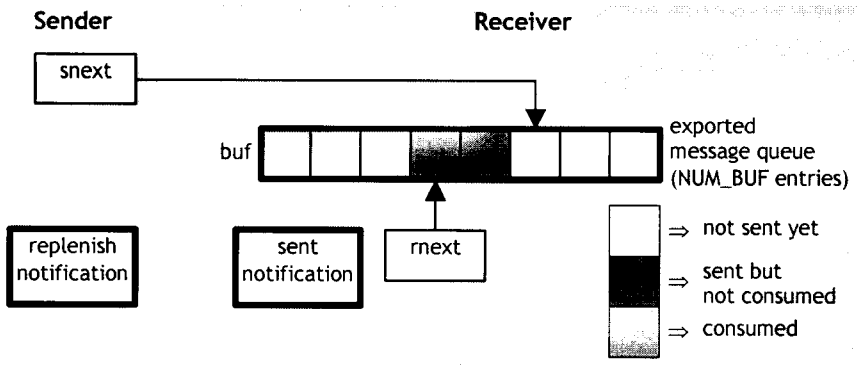

Figure 19. Channel data structures.

The receiver first exports a memory segment that stores the circular message queue containing *NUM_BUF* entries. Each fixed-sized message entry accommodates the largest message that a sender may transfer, which may span multiple pages. The receiver also allocates a *sent* notification that triggers each time the sender delivers a message. A pending signal on the *sent* notification indicates that the channel is ready with a message in the queue at the read cursor (*rnext*). The sender uses a write cursor (*snext*) to determine the destination location in the queue for the next transfer. It also allocates a *replenish* notification that triggers each time the receiver delivers a replenishment message. The receiver sends the replenishment every time it consumes and frees a sufficient (*REFILL*) number of messages.

send(msg)	**recv(&addr)**

if avail == 0
  notfWait(replenish)
  avail = avail + REFILL
avail = avail - 1
	notfSpin(sent) then **notfWait**(sent)
	freed = freed + 1
	if freed % REFILL == 0
	  writeN(sender, replenish)

writeN(buf[snext], msg, sent) snext = (snext + 1) % NUM_BUF	addr = buf[rnext] rnext = (rnext + 1) % NUM_BUF

Table 14. Channel send and recv implementation.
API calls simplified for clarity.

Table 14 shows the channel **send** and **recv** operations. On the sender, **send** first verifies that it has sufficient credits using an *avail* counter, initialized to *NUM_BUF-1*, to track the free entries in the queue. If the queue is full, **send** will wait for a replenishment notification from the receiver, which will credit it with *REFILL* entries in the queue. Once there is sufficient space, **send** consumes a credit and writes the message, together with a *sent* notification, to the message buffer indexed by *snext* into remote queue. On the receiver, **recv** first waits for a message in the queue by spin waiting, then block waiting, for a *sent* notification. The receiving NI

delivers this notification only after the message completely arrives into the destination queue. After **recv** accepts the notification, it checks if the sender requires replenishment by incrementing the *freed* counter and comparing it with the *REFILL* threshold. After every *REFILL* increments, **recv** returns credits to the sender by simply sending it a *replenish* notification. Finally, **recv** returns the address of the message, indexed by *rnext* into the queue, to the calling application.

The *REFILL* threshold should be large enough to reduce the rate of replenishment notifications, but small enough to improve the channel utilization and keep the pipeline full. On the one hand, this value ensures that the sender can send at least *REFILL*, and up to *NUM_BUF-1*, messages each time it receives a replenishment update. On the other hand, the receiver must send a replenishment notification every time it consumes *REFILL* messages. The sending and receiving applications can decide on the appropriate *REFILL* threshold.

Relying on NetVM's ordered notification delivery simplifies the channel implementation. If NetVM did not deliver notifications in total order, then **write** *j* sent after, but arriving before, **write** *i* will trigger a notification causing **recv** to incorrectly assume that **write** *i* has arrived. In this case, **recv** has to additionally check that the correct, and complete, data had indeed arrived into the queue. Fortunately, NetVM delivers ordered notifications. Thus, it will delay the notification for **write** *j* until after it has delivered both data and notification for **write** *i* and the data for **write** *j*.

send never completely fills up the remote message queue. It always reserves one available entry on the receiver because *avail* counter is initially *NUM_BUF-1* instead of *NUM_BUF*. This subtle but important behavior is due to the semantics of **recv**. **recv** returns the address of the newly received message and allows the application to process it *in place* or copy it into a private buffer. Therefore, the channel must guarantee that the sender cannot overwrite that message until the application calls **recv** for the next message, which implicitly frees the preceding message.

Supporting many-to-one communication requires a channel for each sender. Each sender-receiver pair sets up a separate channel with a unique *sent*-notification ID, which also identifies the channel. To determine which sender has sent a message, the receiver first checks the notification FIFO to obtain the identifier of the channel with a ready message. It then calls **recv** on specified channel to locate the address of the message in the queue. Alternatively, the receiver can also register a handler for each *sent*-notification. NetVM will call the correspond-

ing notification handler whenever a message belonging to any one of the registered channels arrives.

7.1 Summary

User-level channels provide one-way stream-oriented communication between a pair of communicating applications. The receiving application exports a message queue, which the sending application can directly access. Both applications track the total sent and consumed messages and implement flow control over the channel using notifications. NetVM's ordered notification delivery simplifies the flow-control protocol by guaranteeing that all previously sent data has arrived into the destination memory before delivering the notification. Channels are not receiver managed; the receiving application has to either process the received message in place or copy it to separate buffer before freeing the associated channel slot.

8 Atomic and synchronization operations

Synchronization is an important aspect for concurrent applications. In particular, efficient synchronization is essential for applications that exhibit fine-grained interaction in a SAN environment. Synchronization schemes that exploit the underlying network hardware can provide more efficient synchronization support for these applications. This section briefly examines existing implementations for three common synchronization idioms (mutual-exclusion locks, wait queues, and semaphores) and proposes a new implementation that extends the network interface to support these constructs more efficiently.

For centralized synchronization schemes, a key question is the location of the central arbitrator process. The first alternative is to assign a server process in the host to arbitrate synchronization requests from the network. Send-receive communication systems typically use this approach; the process blocks on a request queue waiting for application synchronization requests. NetVM channels, which are implemented with only RDMA data transfers and notifications, can also easily support this approach. However, this approach also has two problems. First, invoking a server process typically requires the network interface to interrupt the kernel to wake up the process, which increases the response latency for the operation and is thus undesirable for fine-grained synchronization. Second, frequently interrupting the kernel to schedule the server process affects other applications running on the same host. Coady et al. [16] showed that these fine-grained operations could slow down other unrelated CPU-intensive applications by 17%.

The second alternative is to process the synchronization requests using the network processor. This approach avoids the need to invoke the host processor and can result in lower response latency, especially if the synchronization operations are lightweight. However, this approach is also not scalable. Network processors are usually significantly slower than host processors, thus they are easy to overstress with too many requests. Network interfaces also have limited memory, thus they cannot support potentially large or many data structures in local memory.

Synchronization schemes that apply to traditional shared memory multiprocessors also apply to NetVM. These multiprocessors typically implement a standard set of atomic operation primitives to support various synchronization implementations. This approach is particularly suitable for NetVM because it already supports remote memory read/write access, which is readily extendable to also include remote memory read-modify-write access.

There are many well-known algorithms for implementing mutual-exclusion using standard atomic operations. NetVM adopts the scalable MCS distributed lock for mutual exclusion because it has four desirable properties. First, applications busy wait only on local memory and not on remote memory. Busy waiting on non-cache-coherent remote memory quickly saturates the network and causes congestion. Second, an application only requires a constant number of network transactions to acquire and then release a lock regardless of the number of participants in the network, which is important for scaling. Third, an application only requires a constant-sized data structure in its local memory to support the distributed lock, regardless of the number of participants in the network, which is also important for scaling. Finally, lock acquisitions occur in FIFO request order. MCS locks require only the standard *swap* and *compare-and-swap* atomic operations, which NetVM needs to implement on the network interface. However, the original algorithm only allows applications to busy wait to acquire the lock, modifying it to support a hybrid busy-block waiting is straightforward using NetVM notifications.

Implementing wait queues and semaphores using only standard atomic operations is also possible. Typically, a synchronization operation occurs in three phases [21]. The first phase acquires mutual-exclusive access to the shared synchronization data structure, using MCS locks described in the preceding paragraph or a simpler mechanism such as *test-and-set* operations. The second phase manipulates the data structure, which may involve signaling other processes in the network. The third phase releases mutual-exclusive access to the data structure. This conventional three-phase approach is not efficient compared to the MCS lock in terms of network transactions, operation latency and network-processor overhead. The problem is that the standard atomic operations alone are not ideal for implementing these synchronization constructs efficiently, because these operations individually cannot atomically update all the necessary state information in the shared data structure.

NetVM augments the network interface with an extended set of atomic operations to support wait queues and semaphores. With this new set of operations, wait-queue and semaphore operations can complete with fewer network transactions, thus reducing operation latency and network-processor overhead. This augmentation does not amount to implementing a centralized network-processor-based server; the network interface does not maintain any state for the

synchronization constructs. The new synchronization primitives are simply more sophisticated *fetch_and_Φ* operations, which are easy to implement and execute efficiently. As a result, NetVM can implement efficient wait queues and semaphores using the MCS-lock-inspired approach and benefit from the same efficiency, scalability and fairness properties of MCS locks.

The remainder of this section describes the implementation of the NetVM synchronization operations in four parts. The first part describes the standard atomic operations common to multiprocessor systems. The second part describes the MCS lock and its NetVM extension. The third and final parts describe the wait-queue and semaphore synchronization idioms respectively.

8.1 Standard atomic operations

Operation	Return	Description
testandset(seg, offset)	TRUE if set	test and set
incr(seg, offset)	old value	increment by one
decr(seg, offset)	old value	decrement by one
swap(seg, offset, new)	old value	swap
cswap(seg, offset, cmp, new)	old value	conditional swap

Table 15. General atomic operations.

Table 15 shows the standard atomic *fetch_and_Φ* operations that are available to NetVM applications. These operations are synchronous; an application can perform only one atomic operation at a time. In each operation, *(seg, offset)* specifies the location of the remote memory-word operand. **testandset** atomically tests a remote word and sets it to a nonzero value if it was previously zero. In either case, it returns the result of the comparison test to the caller. **incr** and **decr** respectively increment and decrement a remote 32-bit word and return its original value. **swap** atomically exchanges a remote word with the new argument and also returns its original value. Finally, **cswap** compares a remote word with the *cmp* argument and swaps it with the *new* argument if they match. In either case, **cswap** returns the original word before the comparison.

Atomic-Operation Descriptor	
Myrinet wormhole route (8 bytes)	
Myrinet message tag (NETVM)	(padding)
sending node	sending node sequence number
source protection key (64 bits)	
source port	atomic operation
argument 0	argument 1
argument 2	destination node
destination VPN	destination VPN hash key
destination protection key (64 bits)	
destination page offset and length	(padding)
command (ATOM)	(padding)

Table 16. Format of the atomic-operation descriptor.

Atomic operations are similar to small **read** operations. Table 16 shows the atomic-operation descriptor format. The requesting application first queues the atomic-operation descriptor into the command queue. The NI on the local node forwards the message, unmodified, to the remote NI. The command handler on the remote network interface first fetches the requested operand from the target application virtual memory into a local buffer in the network interface. Leaving the host page locked, it then updates the local copy of the operand depending on the specific atomic operation. Finally, it sends the reply message with the result of the operation to the requesting node. At the same time, it also transfers the updated operand to the application memory and unlocks the page after the host-DMA transfer completes. When the requesting NI receives the reply message, it transfers only the result of the atomic operation into a pinned reply buffer using host DMA. NetVM allocates this reply buffer, used only to store the result of an atomic operation, when the application registers. After issuing the request, the application spins on this buffer until the result arrives into host memory.

The **incr** and **decr** operations require endian conversion on the network interface. The Myrinet network processor is big endian but the Intel host processor is little endian. To correctly increment or decrement a 32-bit little-endian word on the host, the NI has to convert the operand to big endian before performing the actual operation and convert it back to little endian before updating the result to host memory. Unfortunately, the network processor does not have hardware support for the two endian conversions. Therefore, the NI performs them in software, through a series of four bit mask and shift operations for each conversion.

A race condition exists if the host and NI both try to perform an atomic operation on the same operand simultaneously. This condition occurs when the NI receives multiple atomic-operation messages for the same nonresident operand. The NI redirects the first operation to the bounce buffer because the requested page is not resident. The kernel driver fetches the page and completes the operation. However, the NI could receive a second message during this time and

try to update the same operand as well. To prevent this race, the kernel driver and NI implement per-application spin locks to ensure mutual exclusion during atomic operations. This approach is similar to the host-NI synchronization scheme for the shadow page table described in Section 4.4 on page 58.

8.2 Mutual-exclusion locks

An MCS lock is a scalable distributed lock mechanism that allows applications to acquire a lock in FIFO order with low latency. It is particularly suitable for NetVM remote memory access semantics because of two properties. First, the operations require at most two, but possibly one, network transactions to acquire or release a lock. These efficient transactions involve only atomic and notifying updates on remote memory. Centralized lock servers are unnecessary. Second, the operations busy wait only on local memory, which is important in a LAN environment where spinning on remote memory costs significantly more than spinning on local memory. To reduce the need for the original algorithm to consume the host processor while busy waiting on local memory, a NetVM extension allows an application to block waiting to acquire a lock after a busy-wait timeout. Thus, it will free up the processor to schedule other applications on the same host.

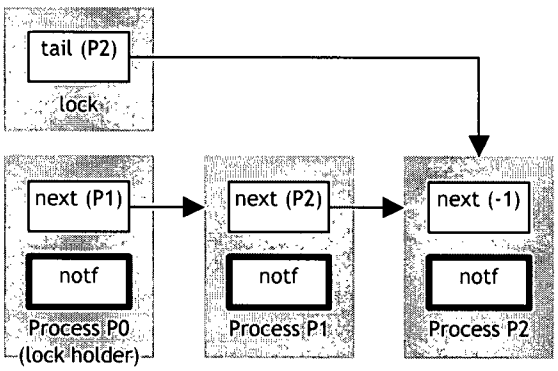

Figure 20. MCS distributed lock-chain example.

Figure 20 shows the data structures for a NetVM distributed lock-chain example. A globally addressable centralized data structure (box *lock*) stores the *tail* variable to track the current tail process (P2) of the lock chain. There are three participating processes (boxes *P0* to *P2*), each process stores a data structure in its globally addressable virtual address space that contains a *next* field, which points to its successor in the lock chain, and a *notf* notification triggered when its predecessor hands over the lock.

acquire(lock)	release(lock)
next = EMPTY tail = **swap**(lock→tail, me) if tail == EMPTY return **write**(tail→next, me) **notfSpin**(notf)/**notfWait**(notf)	if next == EMPTY tail = **cswap**(lock→tail, me, EMPTY) if tail == me return spin until next ≠ EMPTY **writeN**(next→next, notf)

Table 17. Lock acquire and release implementation.
API calls simplified for clarity.

Table 17 shows the lock acquire and release operations. Initially, the lock *tail* value is EMPTY to indicate that no process is holding the lock. To acquire the lock, an application atomically swaps the *tail* value with its own identifier (*me*). A swap result of EMPTY indicates that it is the only acquirer in the chain and, hence, it obtains the lock. If another application now tries to acquire the lock, it will find the identifier of the lock holder, instead of EMPTY, or the last of multiple requestors that have previously chained into the queue. In this case, it now knows the predecessor in the wait queue and writes its own identifier into the predecessor's *next* field to link itself into the chain. Finally, it spins, then blocks waiting after a spin timeout, for a notification from the predecessor, which will release the lock with a notifying write to it.

To release the lock, the current holder first checks its *next* field for a successor in the lock chain. If one is waiting, the holder simply hands over the lock to the successor by sending it a notification. Otherwise, it releases the lock back to the central lock data structure by conditionally swapping EMPTY with the lock *tail* only if the existing tail value is itself (*me*). A successful update indicates that no other applications have chained into the lock queue and the release operation exits. If, however, there is a race and the conditional swap operation fails, the holder now knows the new requestor that will be chaining itself into the lock queue shortly by writing into the holder's *next* field. Therefore, the holder spins until its next field is updated by that new requestor and sends a notification to release the lock to it.

The key difference between NetVM locks and MCS locks is that, with MCS locks, an acquiring process only spins on a local nonzero *locked* flag until it acquires the lock. The releasing process transfers the lock by writing a zero value into this flag on its successor in the wait queue. With NetVM, the releasing process uses a *notifying* **writeN** to transfer the lock. The acquiring process initially spins on shadow copy of the notify count in host memory for a short period before falling back to blocking mode. In this way, it will acquire the lock without rescheduling if the wait interval is small, or revert to block waiting to avoid consuming the CPU for a long period [20].

8.3 Wait Queues

A NetVM wait queue allows processes to wait in a FIFO queue for a specific event. Any process can wake up exactly one process at the head of the queue by signaling an event on the queue. Wait queues are useful to implement synchronization operations that involve a set of queuing processes waiting for events triggered externally by other processes. In particular, NetVM uses wait queues to implement condition variables in monitors.

NetVM wait queues differ from lock chains in two ways. First, *only* the lock holder can signal the process at the head of a lock chain, no other process knows the identity of the front process in the chain and, thus, cannot signal it. In contrast, *any* process can wake up the front process in the wait queue. Second, a process trying to acquire a lock cannot continue until it has successfully obtained the lock. In contrast, a process can place itself on a wait queue and defer waiting for the event associated with the queue. This deferment is necessary to correctly implement condition variables with Mesa monitor semantics described later in Section 8.3.1.

The NetVM wait-queue mechanism shares several advantages with the NetVM distributed lock. It requires only a small centralized control structure to store the state of the queue. Each participating application also needs to store only a small constant-sized data structure as part of the distributed queue. A remote queue server is unnecessary; all operations require at most three remote-memory network transactions to place or remove a process from the queue. A queued process can spin on local memory or block waiting for the wakeup event.

Type	Operation	Return	Description
NI atomic operations	enqueue(seg, offset, id) dequeue(seg, offset) newhead(seg, offset, newhead)	tail (head, count) (pend, count)	add tail entry remove head entry set new head entry
user library API	insertQ() waitQ() removeQ()	- - -	enqueue self wait for wakeup signal front process

Table 18. NI and user-library wait-queue operations.

NetVM implements the wait queue in two levels through the interfaces shown in Table 18. The network interface implements three new atomic operations for the centralized queue data structure. These new operations extend the standard atomic operations in Table 15 and include procedures that, although slightly more sophisticated than the standard operations, still execute in constant time. **enqueue** updates the tail queue so as to chain the calling process to the end of the wait queue. **dequeue** removes the front queue entry so that the signaling application can wake up the front waiting process. **newhead** updates the front queue entry with the

successor to the signaled process in the wait queue. If the wait queue received a second intervening **dequeue** request, **newhead** will indicate that the signaled process should also immediately wake up its successor as well. These three operations alone only apply to the centralized queue control structure and are still insufficient to fully implement the wait queue.

The user library completes the implementation by managing the distributed queue in application memory and the transfer of control between the signaling and waiting processes. It implements three API operations for applications to access: **insertQ** inserts the calling process identifier into the wait queue and returns immediately. **waitQ** performs the actual wait by spinning, or blocking, on the wakeup event that will eventually signal the waiting application. **removeQ** removes and signals the front process waiting in the queue. Conforming to monitor semantics, this signal is lost if the queue is currently empty. Both local and remote processes must access the centralized control structure using the user-library API. Even though the local process may have direct access to the individual fields, the network interface serializes all accesses to the control structure through the atomic operations.

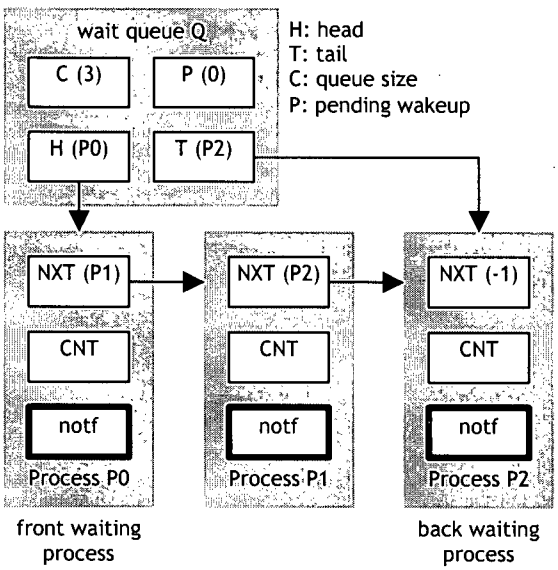

Figure 21. Distributed wait queue example.

Figure 21 shows the data structures for the distributed wait queue example. The globally addressable centralized data structure, box *Q*, contains the state of the wait queue. Specifically, it stores the head (*H*) and tail (*T*) pointers that point to the first and last processes in the wait queue respectively. It also stores a queue-size counter (*C*), which tracks the number of waiting processes in the queue, and a pending-wakeup counter (*P*), which NetVM uses to handle races

between signaling and signaled processes that may leave the queue temporarily inconsistent. In the figure, head pointer *H* points to the front process *P0*, tail pointer *T* points to the last process *P2*, queue size *C* = 3 indicates there are three waiting processes in the queue, and *P* = 0 indicates that there are no pending wakeups and the wait queue is consistent.

Each participating process (boxes *P0* to *P2*) in the wait queue stores two globally accessible variables and a notification. The next pointer (*NXT*) points to its successor in the wait queue. A new queuing process updates this pointer in its predecessor (e.g. P2 sets P1→NXT = P2) unless it is the first process in the queue. The count variable (*CNT*) indicates if a process has, or will soon have, a successor in the wait queue. The signaling process updates this variable with a notifying **writeN** that signals the waiting process. The signaled process must also update any existing, or imminent, successor as the new front process in the queue (Q→H). Finally, each process allocates a notification (notf) triggered by a signaling process.

<t> = enqueue(id)	**<h, c> = dequeue()**	**<p, c> = newhead(h)**

C = C + 1
if C = 1
  H = T = id
  reply <EMPTY>
else
  reply <T>
  T = id
	if C = 0
	  reply <EMPTY, -1>
	if H ≠ EMPTY
	  C = C - 1
	  reply <H, C>
	  H = EMPTY
	else
	  P = P + 1
	  reply <EMPTY, C>
		if P > 0
		  P = P - 1
		  C = C - 1
		  reply <TRUE, C>
		else
		  H = h
		  reply <FALSE, C>

Table 19. Wait-queue implementation on the network interface.

Table 19 shows the three NI atomic operations on the wait-queue control structure. The NI command handler processes them similarly to the standard atomic operations. It first fetches the entire 4-word control structure from application host memory into a local buffer. After completing the specific operation, it replies to the caller with the result and transfers the updated structure back to application host memory. Initially, both head (H) and tail (T) pointers contain EMPTY, and the queue-size (C) and pending-wakeup (P) counters contain zero.

The **enqueue** operation first increments the queue-size counter, which tracks the number of waiting processes in the queue. If only one process is waiting, **enqueue** sets the head and tail pointers to that process identifier. Otherwise, it replaces the existing tail pointer with the new process identifier and returns the old tail pointer (t), so that the new process can chain itself into the wait queue.

The **dequeue** operation first checks the queue size and aborts the operation on an empty queue. If the current head pointer is valid, **dequeue** invalidates it and decrements the queue-size counter to reflect the removal of the front entry. It returns the old head pointer and new queue size to the caller to wake up the front waiting process. **dequeue** finds an invalid head pointer if the previously signaled process has not yet updated the new head pointer with the successor. In this case, **dequeue** records the deficit state by incrementing the pending-wakeup counter, which tracks the number of front processes that should immediately wake up whenever the head pointer is updated.

The **newhead** operation performs this deficit check when a signaled process updates the head pointer with the successor in the wait queue. If there is a pending wakeup, **newhead** simply decrements both pending-wakeup and queue-size counters, and returns a code (TRUE) indicating that the new front process should also be immediately signaled. If there is no pending wakeup, **newhead** simply updates the head pointer with the provided value. In either case, **newhead** also returns the number of waiting processes in the queue.

insertQ	**removeQ**	**waitQ**

NXT = EMPTY
CNT = 0
t = **enqueue**(me)
if t ≠ EMPTY
  write(t→NXT, me)
	<h, c> = **dequeue**()
	if h ≠ EMPTY
	  writeN(h→CNT, c, notf)
		notfSpin(notf) then **notfWait**(notf)
		if CNT > 0
		  spin until NXT ≠ EMPTY
		  <p, c> = **newhead**(NXT)
		  if p = TRUE
		  writeN(NXT→CNT, c, notf)

Table 20. Wait-queue implementation in the user library.
API calls simplified for clarity.

Table 20 shows the three user-library operations for the wait queue. The **insertQ** operation calls **enqueue** update the calling process identifier into the wait queue. If the return tail pointer (t) is not EMPTY, **insertQ** chains the calling process into the wait queue by writing its identifier into the NXT field in its predecessor, which is stored in the tail pointer.

The **removeQ** operation calls **dequeue** to remove the head entry from the wait queue. If the returned head pointer is valid, **removeQ** performs a notifying **writeN** to the CNT field of the head process. This **writeN** serves two functions: it wakes up the front waiting process and it updates the waking process with the updated number of queue entries (zero if the waking process was the only one in the queue). **removeQ** may receive an EMPTY head pointer and hence cannot wake up the unknown process for two possible reasons: either because the wait queue was previously empty, or because the previously signaled process has not yet updated

the head pointer. In the second case, NetVM defers the wakeup until the signaled process updates the head pointer.

The **waitQ** operation first spins, and then blocks, for the notification that wakes it up. Once notified, a CNT value greater than zero means that it must update the head pointer with its successor by calling **newhead**. To do this, it spins waiting until its NXT field is updated by the successor in the wait queue before updating the head pointer. In the common case, the successor would have already updated the NXT field when it called **insertQ**. If **newhead** indicates that another **removeQ** request had already occurred (when there is at least one pending wakeup), **waitQ** also performs the deferred wakeup, on behalf of **removeQ**, with a notifying **writeN** to its successor.

NetVM relies on the pending-wakeup counter to address a race condition that can occur over the head pointer (H) between a signaling process and a signaled process during multiple wakeups. A signaling process accesses the head pointer to wake up the front waiting process; a signaled process updates the same pointer with its successor in the wait queue. However, a second signaling process will find an invalid head pointer if the previously signaled process is slow in updating the new value. The nonzero pending-wakeup counter (P) tracks this deficit condition and informs the signaled process, when it updates the head pointer, that it should also immediately wake up its successor.

8.3.1 Monitors

Mesa monitors are straightforward to implement in NetVM using the available mutual-exclusion lock and wait-queue mechanisms. A monitor requires a mutual-exclusion lock to ensure that only one process can enter a monitor at any one time. It also requires a wait queue for each condition variable associated with the monitor.

mbegin(mon)	mend(mon)	cvwait(mon, cv)	cvsignal(mon, cv)
acquire(mon)	release(mon)	insertQ(cv) release(mon) waitQ(cv) acquire(mon)	removeQ(cv)

Table 21. Monitor implementation.

Table 21 shows the operations of the NetVM monitor. A process enters the monitor after calling **mbegin** to acquire the monitor mutual exclusion lock; it calls **mend** to exit the monitor after releasing the same lock. **cvwait** waits on a condition variable associated with the monitor. It

first inserts the calling process into the wait queue. It then releases the monitor and waits for the signal on the condition variable. When the signal arrives, **cvwait** reenters the monitor by reacquiring the monitor lock. Finally, **cvsignal** signals an event on a condition variable to wake up the first waiting process waiting, or drops the signal if there is none. A process should immediately exit the monitor after calling **cvsignal**.

NetVM monitors implement Mesa semantics. A process in a monitor that signals a condition variable continues to execute until it releases the monitor lock. The signaled process must reacquire the lock before reentering the monitor. However, the signaled process may not be the first in the monitor-lock queue, a different process can enter the monitor and steal the signaled resource. Therefore, the signaled process must verify that the signaled condition is still valid once it reenters the monitor.

8.4 Semaphores

The NetVM counting semaphore is a simple extension of a wait queue. The semaphore implementation adds a *count* integer in the wait-queue data structure to store the semaphore count value. NetVM also adds two constant-time atomic operations, **semP** and **semV** on the network interface **semP** extends **enqueue** by conditionally queuing the process only if the semaphore count is zero. **semV** extends **dequeue** by conditionally dequeuing the front process of a nonempty wait queue. To complete the implementation, NetVM adds two API functions, **Swait** and **Ssignal**, in the user library.

<t, s> = semP(id)	**<h, c, s> = semV()**
s = S if S > 0 S = S - 1 else <t> = **enqueue**(id) reply <t, s>	if C > 0 <h, c> = **dequeue**() else S = S + 1 reply <h, c, S>

Table 22. Semaphore implementation in the network interface.

Table 22 shows the two semaphore atomic operations on the network interface. **semP** either decrements a positive semaphore count by one, or calls **enqueue** to place the process identifier into the wait queue if the count is zero. **semV** either calls **dequeue** to signal the first process of a nonempty wait queue, or increments the semaphore count by one. Note that, like **enqueue** and **dequeue**, **semP** and **semV** requires **newhead**, which is unmodified from the wait queue, to complete the implementation on the network interface.

<s> = Swait()	<s> = Ssignal()

<t, s> = **semP**(me)
if s > 0
  return s
if t ≠ EMPTY
  write(t→NXT, me)
waitQ()
return s
	<h, c, s> = **semV**()
	if s = 0
	  writeN(h→CNT, c, notf)
	return s

Table 23. Semaphore implementation in the user library. API calls simplified for clarity.

Table 23 shows the two user-library operations for semaphores. **Swait** calls **semP** to conditionally decrement the semaphore count and returns the pre-decremented value if it is positive. Otherwise, **Swait** chains the process to the end of the wait queue, as in **insertQ**, and then immediately calls **waitQ** to wait for the wakeup signal. **Ssignal** calls **semV** to conditionally increment the semaphore count and returns the post-incremented value if there are no processes waiting on the semaphore. Otherwise, **Ssignal** wakes up the first waiting process in the queue with a notifying write, as in **removeQ**.

8.5 Summary

The network interface implements atomic read-modify-write operations, which are similar to the read operation but with two key differences. The first difference is that the remote network interface, instead of simply replying with the requested data, also processes the atomic operation and writes the updated result back to the host memory. The second difference is that the requesting application always busy waits on a reply buffer waiting for the atomic operation to complete.

NetVM implements an MCS-based distributed lock using only two standard atomic operations: **swap** and **cswap**. The counter-based notification mechanism simplifies the implementation and, unlike the original MCS design, allows an application to block wait, in addition to busy wait, to acquire the lock.

The wait-queue mechanism requires three new atomic operations on the network interface. These operations allow NetVM applications to enqueue and dequeue themselves with a reduced number of network transactions compared to the traditional shared-memory-based implementation using standard atomic operations. With the wait queue in place to support condition variables, Mesa monitors are straightforward to implement using both distributed lock and wait-queue mechanisms.

Finally, the counting-semaphore mechanism requires two new atomic operations that extend the existing wait-queue atomic operations. NetVM implements the counting semaphore essentially as a conditional wait queue with an additional counter to track the semaphore value.

The next chapter presents an evaluation of a working prototype based on the design description in Chapters 4 to 8.

9 Evaluation

This section presents an evaluation of a complete implementation of NetVM in five sections. The first section examines the costs and benefits of using NetVM's memory-management approach. It measures the host overhead for integrating NetVM with the host VM system and the network interface overhead for page-locking operations. It also compares the page-locking approach with the traditional host-based dynamic page-pinning techniques. The second section measures the performance for small and large data transfers. It compares the cost of NetVM page locking to an efficient static pinning approach that allows applications to hand physical addresses directly to the network interface. It also examines the impact of NetVM's page fragmentation on misaligned data transfers. The third section reports the latencies and overheads for control transfers using each of the notification-detection mechanisms described in the design. They include synchronously busy waiting, block waiting, and triggering a handler to process a notification. The fourth section presents the performance for application-level channels, which provides stream-oriented data transfers. Finally, the fifth section examines the atomic and synchronization operations. In particular, it investigates the overheads of these operations over the basic remote **read**, which has a similar implementation. It also demonstrates that NetVM's synchronization operations are both scalable and add low overhead.

The experimental prototype consists of a cluster of 1-GHz Intel Pentium III PCs, each with 512MB of PC133 SDR SDRAM running the FreeBSD 4.6R operating system. The Myrinet network connecting the PCs has a peak bidirectional bandwidth of 1.28Gb/s (160MB/s) and a minimum point-to-point latency of 100ns. Each network interface contains a 132-MHz LANai 9.2 network processor and 8MB of 64-bit SRAM onboard. A 64-bit 66-MHz PCI bus connects the network interface to the host motherboard and has a peak bandwidth of 528MB/s. The measured host-DMA bandwidth of 4-KB transfers is 437MB/s from host to network interface and 492MB/s from network interface to host. With larger host-DMA transfers, the asymptotic bandwidth is 515MB/s for host to network interface and 527MB/s for network interface to host. The measured wire-DMA bandwidth is 160MB/s.

All timing measurements use a combination of the Pentium cycle counter on the host and the LANai CPU counter on the network interface. The resolutions of the host and NI counters are 1ns and 15ns respectively. The reported latency numbers represent the median of at least 1000 trials of each experiment.

9.1 Memory-management overhead

Type	Operation	Latency (µs)
kernel	mapPage	0.31
	unmapPage (success)	1.44
	unmapPage (PAGE_LOCKED)	0.73
	isPageDirty (CLEAN)	0.55
	isPageDirty (DIRTY)	0.60
NI	lockPage (success)	0.33
	lockPage (PAGE_NOT_MAPPED)	0.39
	unlockPage	0.13
	unlockPageDirty	0.27

Table 24. Host and NI page-table operations.

Table 24 shows the NetVM memory-management overhead to manage the shadow page table on the network interface in two sections. The first section shows the operations in the kernel. **mapPage** requires 0.31µs to insert a page mapping into the shadow page table. This overhead is insignificant compared to the 10–20ms required to fetch the page from the backing store. However, it is 10.61% of the 2.92µs required to map a zero-filled page into application memory for the first time. This first-time-access overhead is not detrimental, because the total number of such faults is limited to the size of the application in available physical memory on the host. For example, an application that exports 128MB of memory requires only an additional 10.16ms to map the pages into the shadow page table for the first time, 0.31µs for each of the 32768 pages.

unmapPage requires 1.44µs to successfully remove a page mapping from the shadow page table, or 0.73µs to detect that the NI has locked the page for a DMA transfer. This overhead is also insignificant compared to the cost of replacing a page in host memory, especially if the kernel has to write a modified page to the backing store. Furthermore, in the steady state, the VM system pairs the unmapping of one page with the mapping of another. Thus, the total cost of an **unmapPage-mapPage** pair is insignificant compared to the cost of writing a modified page to the disk and fetching the new one. When an application terminates, NetVM unmaps all its resident exported pages in bulk. For example, an exiting application that mapped 128MB of exported memory requires only an additional 45.88ms to unmap its pages from the shadow page table.

Finally, **isPageDirty** requires 0.55µs to detect a clean mapped page and 0.60µs to detect and clean a mapped page modified by the NI. This overhead occurs in the page-out daemon and in the page-free routine, when the kernel needs to replace or free a page.

The second section of Table 24 shows the memory-management operations on the network interface. **lockPage** requires 0.33µs to successfully translate the virtual address and lock a mapped page in the shadow page table, or 0.39µs to detect that the host has unmapped the page and reverse the locking operation. Minimizing the **lockPage** time is important because this operation lies in the critical path of a message transfer on the receiving node for small transfers and on the sending node for large transfers. For small transfers, the application copies the data from host memory into the source network interface using programmed IO, therefore the NI does not need to pin the page for a DMA transfer. For large transfers, **lockPage** is required on both sending and receiving nodes. However, because the receiving NI processes a message as soon as the message header arrives from the wire, the **lockPage** operation on the receiving node overlaps with the wire-DMA transfer of the data from the network link and thus does not lie in the message critical path when measuring latency.

unlockPage requires 0.13µs to unlock the page in the shadow page table after a host-DMA transfer from host memory into network interface memory on the source node. **unlockPageDirty** requires 0.27µs to unlock the page and mark it as modified after the host-DMA transfer from network interface memory into host memory on the destination node. Both **unlockPage** and **unlockPageDirty** do not lie in the latency critical path of the message transfer.

9.1.1 Comparison with dynamic pinning

Table 25 on the following page compares two different host-based dynamic page-pinning mechanisms to the NetVM NI-based page locking in three sections. The first two sections show the potential benefit from using a host-based approach to pin and unpin a cluster of pages (1—8 virtually contiguous pages) in a single request. Batch pinning reduces the average per-page overhead by amortizing the cost of the application system call, host interrupt and kernel page-table lookup for a cluster of pages into a single operation. However, batch pinning is effective only if there is spatial locality in the accesses, poor locality actually increases the per-page overhead by unnecessarily pinning unrelated pages. For example, the per-page cost to pin and unpin eight pages individually using an application-based approach is 4.21us. If these pages belong to the same 8-page cluster (87.5% hit rate), then the average per-page cost falls to 1.30µs. If these pages are spatially unrelated however (0% hit rate), then the per-page cost rises to 10.41µs, which is the time to pin and unpin each 8-page cluster.

Operation	Cluster Size (pages)	Latency (µs)	Per-page Overhead (µs/page)	Hit Rate (%)
application system call	1	4.21	4.21	0.0
	2	5.13	2.57	50.0
	4	7.05	1.76	75.0
	8	10.41	1.30	87.5
interrupt host	1	4.92	4.92	0.0
	2	5.37	2.69	50.0
	4	7.42	1.86	75.0
	8	10.96	1.37	87.5
NetVM	-	0.60	0.60	0.0

Table 25. Host-based dynamic page-pinning vs NetVM NI-based page-locking costs.

The first section shows the latency of the system-call-based approach used by systems such as Pin-down Cache. In this approach, the application calls the **mlock** and **munlock** system calls to manage a pinned-page cache in host memory. It then supplies the network interface with the physical address of the pages after obtaining the address translations for those pages from the operating system. To replace a pinned page in the cache, the application simply unpins a victim page (or cluster) and pins the replacement page. An application requires 4.21µs to pin and unpin a single page, but requires 10.41µs to pin and unpin an 8-page cluster. Therefore, the per-page cost drops to 1.30µs if the 8-page cluster is accessed eight times, which corresponds to an 87.5% hit rate. The reported timings include only the basic **mlock** and **munlock** system calls; they do not include any application time to obtain the physical address translations or to manage a pinned-page cache, which are part of the total cost of using this approach.

The second section shows the latency of an interrupt-based approach to page pinning using by systems such as VMMC-1 and U-Net. In this approach, the NI maintains the cache of pinned pages and interrupts the host whenever it needs to replace a page in the cache with a new page. The NI supplies the kernel with the victim and replacement pages during the interrupt. The kernel responds by unpinning the victim page (or cluster) and pinning the replacement page. It then supplies the NI with the physical address of the replacement page, which the NI uses for the DMA transfer. The reported timings only include the interrupt latency, which is 2.31µs, and the kernel-based pinning and unpinning operations; they do not include the full interrupt-handling overhead on the host or any time to manage the pinned-page cache on the network interface, which are part of the total cost of using this approach.

The third section shows the latency of the NetVM page-locking operations. It requires a fixed per-page cost of 0.60µs, which is the sum of **lockPage** and **unlockPageDirty**, to lock and unlock a page and set its modification state.

The measurements in Table 25 also show the tradeoff between using the NetVM network-interface-based page-locking approach and the host-based dynamic pinning approaches. On the one hand, NetVM is significantly more efficient than host-based dynamic pinning when pinning single pages, NetVM's latency is only 14.3% and 12.2% of the system-call-based and interrupt-based approaches respectively. On the other hand, the effectiveness of the host-based approaches largely depends on the hit rate in the pinned-page cache. For 8-page clusters with an access hit rate of 87.5%, NetVM's overhead increases to 46.2% and 43.8% of the system-call-based and interrupt-based approaches respectively. If the hit rate is sufficiently high, NetVM's pinning overhead will eventually exceed the host-based approaches, because NetVM incurs the pinning cost on *each page access*, but the host-based approaches only incur the pinning cost on *each pinned-page cache miss*.

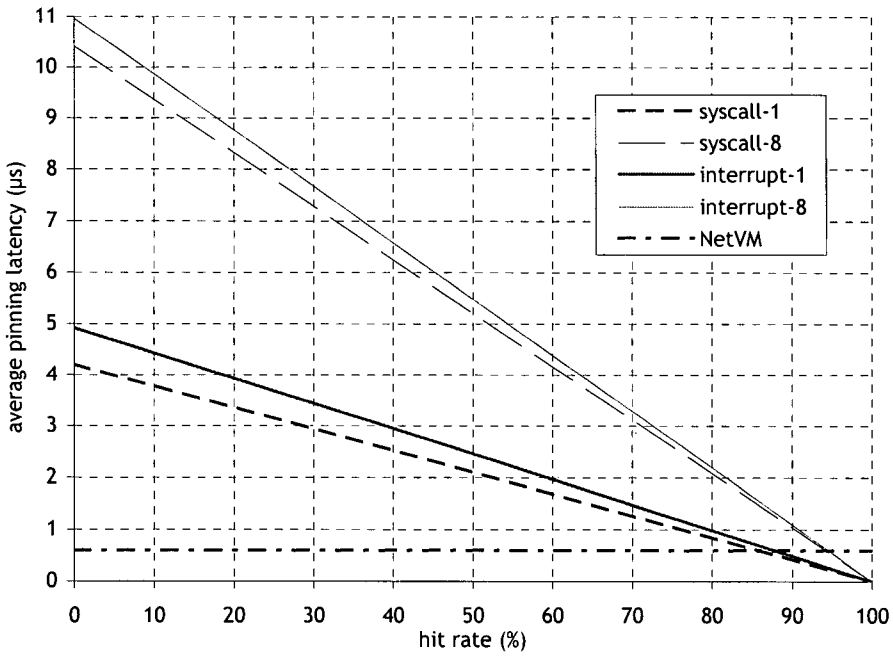

Figure 22. Average pinning latency based on pinned-page cache hit rate.

Figure 22 shows the average pinning latency based on the hit rate. Only single-page and 8-page cluster sizes are shown for the host-based approaches. A hit rate of 0% means that the page, or n-page cluster, is pinned once each time to access the page. A hit rate of 100% means that the page or cluster is pinned only once and then accessed an infinite number of times. NetVM has a fixed cost of 0.60µs independent of the hit rate, as explained in the preceding paragraph. From the figure, the system-called-based approach will require a minimum hit rate of 85.7% and 94.2% for single-page and 8-page clusters respectively to provide a lower amortized pinning

cost compared to NetVM. Similarly, the interrupt-based approach will require a minimum hit rate of 87.8% and 94.5% for single-page and 8-page clusters respectively. As a comparison, sequentially accessing a cluster of eight pages results in a hit rate of 0% with a single-page cluster size and a hit rate of 87.5% with an 8-page cluster size.

The measurements in Table 25 show a second tradeoff between using a host-based dynamic page-pinning approach and using the NetVM NI-based page-locking approach. Unlike the dynamic page-pinning operations, the page-locking operations in NetVM can always lie in the critical path of a message transfer on both sending and receiving nodes.

In the system-call-based approach, the application can pin pages in the critical path of a message transmission on the sending node, but cannot to do the same for the receiving node because it has insufficient time to detect the message arrival and still guarantee that it can pin the required page in time. A key problem is that the operating system may deschedule the application at any time, particularly at the critical moment before it can pin the page. Therefore, a pinned-page cache miss can cause the network interface to drop the message or buffer it in host memory.

Similar to the system-call approach, an NI using the interrupt-based approach also cannot guarantee that it can pin the required page in time on the receiving node. There are three reasons for this situation: First, the host interrupt latency is not always predictable. The measured value on the prototype is 2.31μs in the common case, but rises to above 10μs occasionally. Second, other higher-priority IO interrupts may intervene and prevent the critical page-pinning operations from completing in a timely fashion. Third, interrupt-based pinning for larger cluster sizes requires a significantly longer time to pin the required page (e.g., 10.96μs for an 8-page cluster). Thus, these factors can cause a detrimental delay for the NI, because it only has limited local memory to buffer messages from the network while waiting for the interrupt handler to provide the address translation. For example, a 10-Gb/s network requires only about 3μs to transfer a 4-KB page into the network interface. As a result, a relatively slow interrupt-based page-pinning rate will exhaust all available buffers on the network interface.

In contrast to the host-based dynamic page-pinning approach, NetVM only requires fixed 0.60μs per page to lock and unlock a single page. This fixed cost guarantees that, as long as the page is resident in host memory and its page mapping is found in the shadow page table, NetVM can always lock the page and access it via host DMA.

Another issue with NI-based pinned-page caching is that, like NetVM, the NI always incurs a fixed cost to look up the cache in the critical path of every message. This cost is roughly similar to the fixed page-locking cost in NetVM. If the pinned-page cache lookup time is factored in, the minimum hit rate required by the interrupt-based approach will further increase to a point where it can do no better than NetVM when its pinned-page cache management overhead exceeds NetVM's page-locking overhead of 0.60μs.

9.2 Data-transfer operations

Operation	Size (bytes)	Latency (μs)	Throughput (MB/s)	Link Utilization (% of 160MB/s)
write	4	5.56	1.26	-
writeP	4	5.28	1.51	-
read	4	9.25	0.81	-
readP	4	8.79	0.88	-
write	4K	47.79	155.46	97.2
writeP	4K	47.08	155.36	97.1
read	4K	50.05	147.09	91.9
readP	4K	49.99	147.24	92.0

Table 26. Latency and throughput for 4-byte and 4-KB transfers.

Table 26 shows the latency and throughput of 4-byte and 4-KB transfers. **write** and **read** transfer data between unpinned virtual-memory addresses, **writeP** and **readP** transfer data between pinned physical-memory addresses.

NetVM implements **writeP** and **readP** to approximate data transfer using a static pinning approach. The key difference between **writeP** and **write** is that the application calling **writeP** specifies the physical addresses of pinned source and destination memory. Thus, the NI assumes that the required pages are always resident and locked in host memory and avoids the address-translation and page-locking call to **lockPage** and the corresponding **unlockPage** operation. It directly uses the provided address for the host-DMA transfers.

9.2.1 RDMA write

The reported **write** and **writeP** timings are the one-way median latencies, which are half of the round-trip latencies measured in a ping-pong fashion using symmetrical writes on two nodes. An application on the first node writes to an application on the second node and then spins reading its local memory until it receives a corresponding update from the second node. The

remote application on the second node starts by spinning on its own local memory until it receives the update and then writes the data back to the application on the first node.

A 4-byte **write** requires 5.56μs and a 4-byte **writeP** requires only 5.28μs. The 0.28μs difference, which is 5.0% of the **write** latency, is due to the additional overhead in using unpinned virtual memory for **write** over using pre-pinned physical addresses for **writeP**. **write** has to lock and translate the address of the target page on the destination network interface. The source network interface does not incur this overhead because the application transfers the data into the network interface using programmed IO, instead of host DMA, for small writes.

A 4-KB **write** requires 47.79μs and a 4-KB **writeP** requires only 47.08us. The 1.5% difference is mainly due to the **lockPage** overhead the source network interface. For writes above 96 bytes, NetVM adaptively switches from using programmed IO to using host DMA on the source node to transfer the data from host memory into the network interface.

The throughputs for 4-byte **write** and **writeP** are 1.26MB/s and 1.51MB/s respectively. With small transfers, the overhead of processing the **write** command descriptors on the source node and **write** messages on the destination node dominates the throughput performance. This overhead is larger for **write** compared to **writeP** due to the **lockPage** operation, resulting in 19.8% lower throughput.

The throughputs for 4-KB **write** and **writeP** are 155.46MB/s and 155.36MB/s respectively. These results are virtually equal with only 0.06% difference between them. With large transfers, the 160MB/s network link is the throughput bottleneck, both **write** and **writeP** can achieve about 97% link utilization. NetVM is able keep the wire-DMA stage continuously busy by overlapping the host and wire-DMA transfer operations. The **lockPage** and **unlockPage** overheads do not significantly affect throughput because, on both source and destination network interfaces, the NI has to stall waiting for the wire-DMA engine to complete before it can proceed. This stalling effectively negates the small overheads introduced by **lockPage** and **unlockPage**.

9.2.2 RDMA read

The reported **read** and **readP** timings are the median times required to read a 4-byte or 4-KB block of remote data. An application on the first node initiates a remote read operation and then spins reading its local memory until the last byte of the requested data arrives in its memory.

A 4-byte **read** requires 9.25us. This time is 3.69μs slower than the 4-byte **write** due to two reasons. First, **read** is a round-trip operation, which requires two network messages. Second, both source and destination network interfaces require a host-DMA transfer. The local destination-node DMA is necessary because **read** transfers data into, and not out of, host memory on the requesting node. The remote source-node DMA transfers data from host memory without intervention from the host processor. A 4-byte read is thus 0.46μs, or 4.9%, slower than **readP**. A 4-KB **read** requires 50.05μs and a 4-KB **readP** requires 49.99us. These latencies are virtually equal with only 0.12% difference between them. With large transfers, the **lockPage** operation on the requesting node overlaps with the arrival of the data from the wire.

The throughputs for 4-byte **read** and **readP** are 0.81MB/s and 0.88MB/s respectively. Similar to the 4-byte **write** and **writeP** operations, small-message throughput is dominated by the network processor time to handle the command descriptor, the request and the response messages. The additional overhead in **read** over **readP** thus results in an 8.6% lower throughput. The throughputs for 4-KB **read** and **readP** are 147.09MB/s and 147.24MB/s respectively, both of which are about 92% link utilization. The values are virtually the same, because the wire-DMA transfer also dominates large-message throughput for reads.

9.2.3 Large-message latency and throughput

This section presents the latency and throughput for large transfer sizes to examine two key design tradeoffs in NetVM. Recall from Section 5.2 on page 63 that the NetVM user library fragments a large transfer request into individual page-sized page-aligned requests because the NI handles memory-management operations in page-sized units. Thus, this section only focuses on the results of transfer sizes from 4 bytes to 8-KB as the performance characteristics repeat on every 4-KB boundary. Also, recall from Section 5.4 on page 65 and Section 5.5 on page 68 that the NI transfers data through the DMA stages using a store-and-forward approach. Thus, the DMA stage of a fragment can only affect the same stage of the next immediate fragment. The following paragraphs describe how these two design choices affect NetVM performance.

Figure 23 in the following page shows the read and write latency for page-aligned transfers as a function of data size, ranging from 4 to 8K bytes. Latency increases linearly from 4 to 4096 bytes with the starting and ending values reported in the preceding section. Because the NI transfers data through the DMA stages using a store-and-forward approach, the total time spent by the data in these stages determines the gradient of this first segment of the graph, which is 10.3ns/byte. Summing the reciprocals of the measured source-host DMA, wire-DMA and desti-

nation-host DMA bandwidths results in an expected gradient of 10.6ns/byte, which is within 2.9% of the observed gradient.

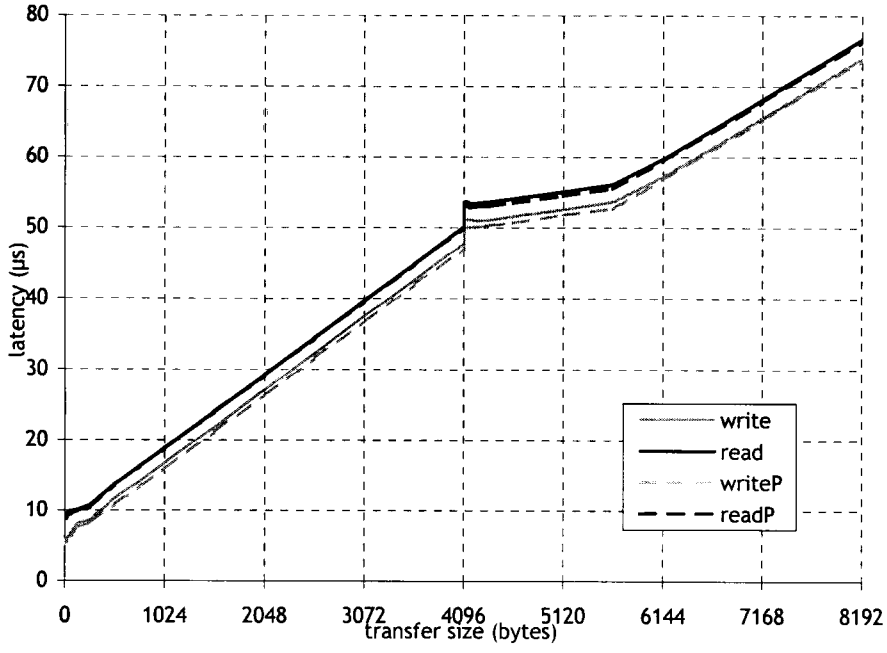

Figure 23. Latency of page-aligned write and read for 4-byte to 8-KB transfers.

Latency increases in a stepwise fashion when the transfer size transits from 4096 to 4100 bytes. The stepwise increases of 3.22µs for **write** and 3.55µs for **read** are due to the way the NetVM user library fragments the transfer. Transferring a region that spans multiple pages in either source or destination memory results in separate **write** or **read** calls. This design choice is motivated by an end-to-end argument to keep page-sized page-aligned transfers as fast as possible. However, this choice also incurs a performance cost for nonaligned transfers, which the stepwise increases show. For 4100-byte write transfers in Figure 23, the user library breaks the request into a 4-KB transfer and by a 4-byte transfer for the following page. Thus, the additional 3.22µs latency over a 4-KB transfer is due to the 5.56µs latency for the 4-byte transfer partially overlapped with the preceding 4-KB transfer.

When the transfer size is between 4100 and about 5600 bytes, latency increases only at a smaller rate compared to the first segment of the graph for two reasons. First, the NI on the source node is able to completely overlap the host-DMA transfer for the smaller second write with the wire-DMA transfer for the first 4-KB write, because the host-DMA stage is both faster and smaller than the 4-KB wire-DMA stage. It is able to send the second message out onto the

wire as soon as its wire-DMA engine completes and is ready again. Therefore, the host DMA on the source node for the second fragment adds virtually no additional latency to the transfer. Second, the reception of the data from the second **write** message also overlaps with the host DMA, into host memory, for the first 4-KB **write**. Any increase in latency is mainly due to the additional time required by the host-DMA transfer for the second **write**. The gradient in this segment of the graph is 1.7ns/byte to 1.8ns/byte, which is within 15% of the expected 2.0ns/byte by taking the reciprocal of the host-DMA bandwidth on the destination node.

Latency also increases linearly in the third segment of the graph when the transfer size is between about 5600 and 8192 bytes. In this case, the wire-DMA stage for the second write is significantly large enough that the destination-host DMA is no longer the dominant factor. The NI on the destination node has to wait for the second wire-DMA transfer to complete before it can initiate the host-DMA transfer for the data. Therefore, both wire-DMA and destination-host DMA stages dominate the transfer. The gradient of this segment is 8.2ns/byte, which corresponds to within 1.2% of the expected 8.3ns/byte derived by summing the reciprocals of the wire-DMA and destination-host DMA bandwidths.

Figure 24 on the following page shows the write and read throughput for page-aligned transfers as a function of data size, ranging from 4 to 8K bytes. The throughput for reads is lower than for writes due to the higher network processor overheads on both source and destination nodes for handling the **read** and **readP** operations.

The throughput for small writes and reads is, as expected, substantially less than the throughput for 4-KB and 8-KB transfers. This low throughput is due to the dominating overhead for processing the individual command descriptors and messages relative to the required time to transfer the actual data.

Throughput rapidly rises with increasing data size from 4-byte transfers until about 1-KB transfers and then bends sharply towards the 4-KB throughput values. The network link essentially limits throughput from the knee of the graph, at about 1KB, to 4KB.

Throughput drops sharply when the transfer size is at the 4096—4100-byte boundary, similar to the stepwise increase in latency seen in Figure 23. For example, in the worst-case 4100-byte transfers, the NetVM user library splits the transfer into a 4096-byte transfer followed by a 4-byte transfer. From the leftmost part of Figure 24, small 4-byte transfers achieve very low throughput. As a result, the throughput for 4100-byte writes is 107.47MB/s, which is 30.9% less

than for 4096-byte writes. Similarly, the throughput for 4100-byte reads is 103.68MB/s, which is 29.5% less than for 4096-byte reads.

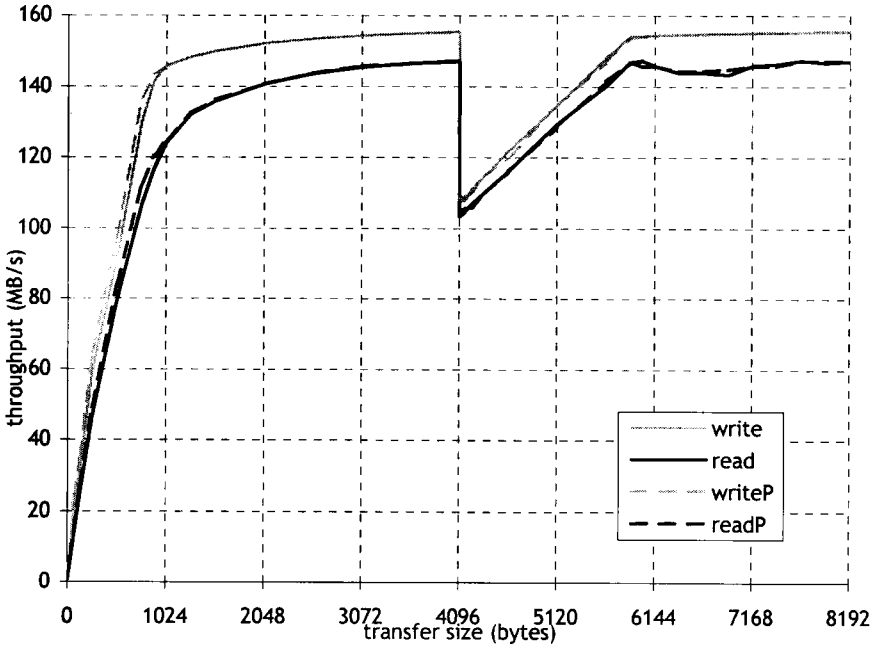

Figure 24. Throughput of page-aligned write and read for 4-byte to 8-KB transfers.

When the transfer size is between 4100 and about 5900 bytes, throughput increases *linearly* until it reaches its maximum value limited by the network link bandwidth. The linear extrapolated **write** and **read** throughput lines also intercept the origin, indicating that an incremental increase in the data size does not increase in the time required to transfer the data. Within this 4100—5900-byte range, the network-processor overhead in handling the smaller second message exceeds the wire-DMA time to receive it from the wire. Therefore, increasing the size of the second message does not significantly affect the overall time it requires to transfer the data, because the data from the second message from the wire would have completely arrived into network interface memory by the time the NI has processed its message header and is ready to initiate the host-DMA transfer.

Finally, when the transfer size is between about 5900 and 8192 bytes, the network link limits the throughput again, similar to the situation when the transfer size was between 1KB and 4KB.

9.2.4 Misaligned transfers

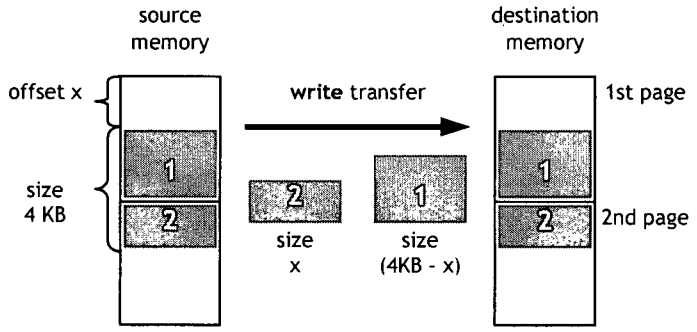

Figure 25. Transferring a 4-KB data block in two fragments.

This section presents the latency and throughput results for non-page-aligned 4-KB write transfers fragmented into two smaller transfers. If either the source or the destination data region spans a page boundary, the NetVM user library splits the transfer at the page boundary, even if the total data size is less than a single page. In this experiment, the application on the source node writes a 4-KB data block with varying page offsets into remote destination memory. In each write, NetVM splits the transfer into two fragments; the first fragment is (4KB - *offset*) bytes large at *offset* bytes from the start of the first page and the second fragment is *offset* bytes large from the start of the subsequent page, shown in Figure 25.

Figure 26 in the following page shows the measured **write** latency as a function of page offset. The horizontal axis indicates the amount of alignment offset from the start of the first page. As a reference, the horizontal broken line shows the 47.79µs latency for a 4-KB page-aligned **write**, which NetVM transfers using only one fragment instead of two.

From the figure, the 4-KB 2-fragment **write** latency is a 3-segment linear trough line. The shape of this line is due to the interaction among the DMA stages for each of the two fragments. Recall from the NI operation description in Section 5.4 on page 65 that a **write** operation for a single fragment requires three DMA stages: *source-host DMA* from source host memory to network interface, *wire DMA* from source to destination network interface, and *destination-host DMA* from destination network interface to target host memory. The time required for each stage depends on both the size of transferred fragment and the completion time of any preceding dependent stages.

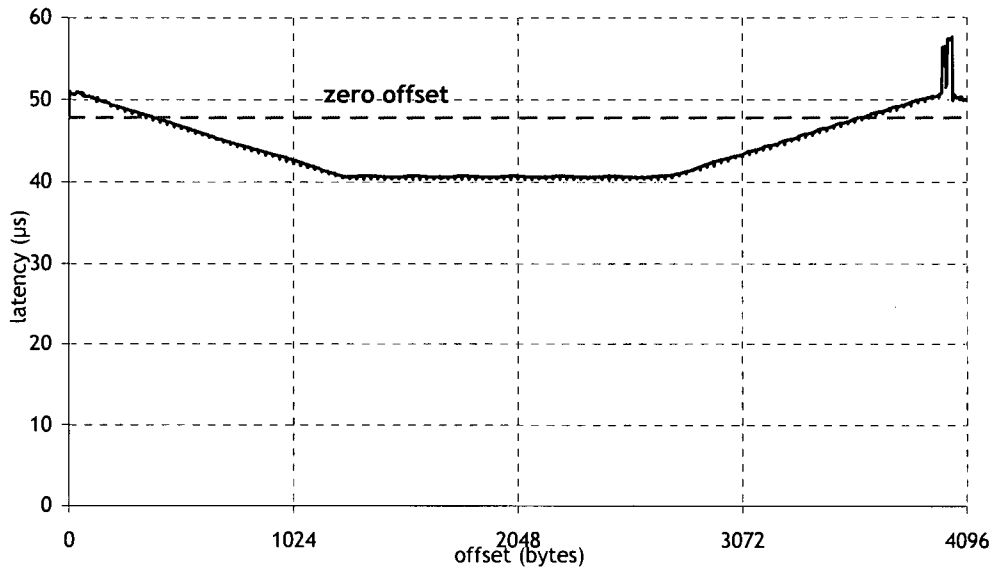

Figure 26. Latency for 4-KB write as a function of page offset.

Appendix A on page 135 describes a model that characterizes this pipelined-DMA transfer and provides a detailed analysis of the DMA engine interactions for each of the three segments. The following paragraphs briefly describe the interactions to provide a general insight into these interactions.

In the first segment of the graph when *offset* is less than 1200 bytes, latency decreases as offset increases. In this segment, the relatively large first fragment dominates latency; the critical path of the transfer passes through its three DMA stages (source-host, wire and destination-host DMA). Increasing the offset reduces the size of the first fragment but adds equally to the size of the second fragment. Thus, the critical path of the first fragment reduces, which reduces latency. The source-host and wire-DMA stages for the growing second fragment are able to completely overlap with wire and destination-host DMA stages for the first fragment respectively and, therefore, do not contribute to the total latency.

In the second segment of the graph when *offset* is between 1200 and 2800 bytes, latency remains unchanged at about 41us. Within this range, the critical path of the transfer passes through the source-host DMA stage for the first fragment, the wire-DMA stages for both fragments, and the destination-host DMA stage for the second fragment. Therefore, a lower source-host DMA time for the shrinking first fragment is negated by a higher destination-host DMA time for the growing second fragment, causing the total latency to remain the same.

In the third segment of the graph when *offset* is greater than 2800 bytes, latency increases as offset increases. In contrast with the first segment, the relatively large second fragment now dominates latency. Increasing the offset adds to the size of the second fragment and increases the critical path, and hence latency, through its DMA stages.

From the figure, the 2-fragment **write** latency when *offset* is between 400 and 3600 bytes is *lower* than the 47.79µs single-fragment **write**. The minimum latency is 40.60µs, or 15% faster than the single-fragment **write**, when *offset* ranges from 1200 to 2800 bytes. Thus, a potential improvement to the user library is that, when appropriate, the library can adaptively fragment large transfers, even if they fit within a single page, to improve the DMA engine and pipeline overlap and, hence, to reduce latency.

The cause of the spikes at the 4000-byte offset is unclear. The spikes are consistently repeatable and only occur at those offsets. The most likely cause is due to hardware idiosyncrasies in the DMA engines on the network interface.

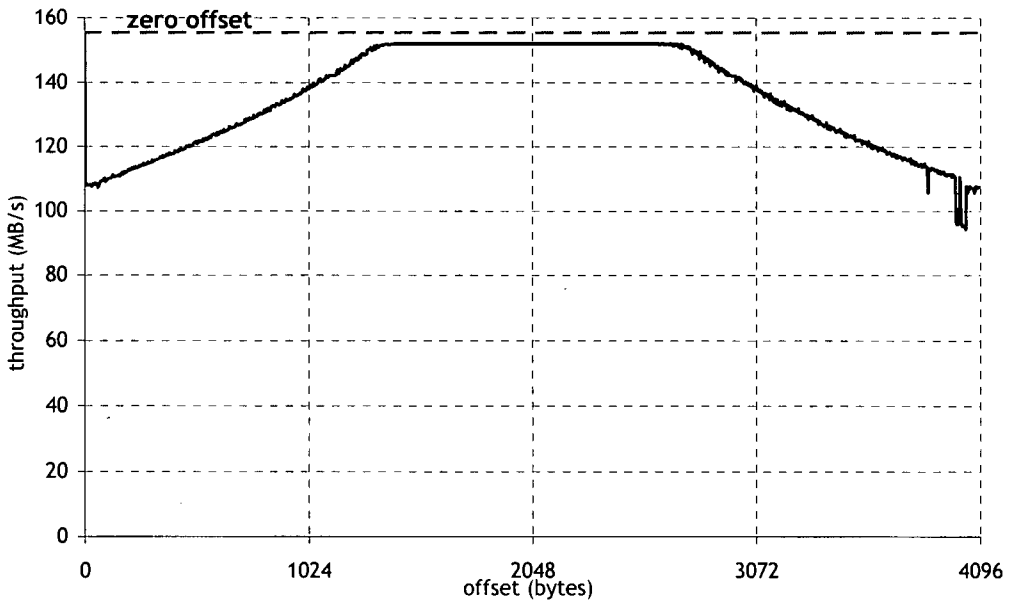

Figure 27. Throughput for 4-KB write as a function of page offset.

Figure 27 shows the measured **write** throughput as a function of page offset. As a reference, the horizontal broken line shows the 155.46MB/s throughput for a 4-KB page-aligned **write**, which NetVM transfers using only one fragment instead of two.

The shape of the throughput graph is largely determined by the DMA overlap in the transfer pipeline and the presence of any bottlenecked DMA stage. A high DMA overlap results in a high DMA pipeline utilization, which indicates that the DMA engines are more efficient in pushing data through the pipeline. In the first segment of Figure 27, increasing offset increases the DMA overlap between the two fragments and, thus, throughput. The opposite is true in the third segment of the graph; increasing offset reduces the DMA overlap and throughput. In the second segment of the graph, the wire-DMA stage is the bottleneck and throughput is limited by the network link bandwidth.

Unlike the **write** latency results, the 2-fragment **write** throughput is *never* higher than the 155.46MB/s single-fragment **write** throughput. This lower throughput is due to the additional processor overheads to handle the second fragment. These overheads introduce additional *bubbles* in the DMA-transfer pipeline compared to the single-fragment **write**, which results in a lower DMA engine and pipeline utilization and, hence, in a lower throughput.

9.3 Control-transfer operations

Operation	Detection mode	Size (bytes)	Latency (µs)	Difference over write	
				µs	%
write	data only	4	5.56	0.0	0.0
writeN	notfSpin	4	8.52	2.96	53.2
writeN	notfWait	4	16.61	11.05	198.7
writeN	notification handler	4	31.14	25.58	460.1
write	data only	4K	47.79	0.0	0.0
writeN	notfSpin	4K	50.68	2.89	6.0
writeN	notfWait	4K	58.73	10.94	22.9
writeN	notification handler	4K	73.16	25.37	53.1

Table 27. Latency for NetVM notification mechanisms.

Table 27 shows the latency for a sending application to transfer control to a remote application using three different notification mechanisms. The table has two sections and each section lists four different cases. The first case in both sections serves as the base case for comparing with the other three notification mechanisms. In this first case, the sending application issues a 4-byte or 4-KB data-only **write** to the remote memory address. The receiving application spins waiting on its local memory until it detects that all the data has arrived. In the three remaining cases, the sending application issues a 4-byte or 4-KB notifying **writeN** to the remote memory address. Specifically, the receiving application in the second case calls **notfSpin**, which directs the NI to update the shadow signal counter and spins waiting on the shadow counter for the

arriving notification. The receiving application in the third case calls the **notfWait** system call, which blocks waiting for the notification. When the notification arrives, the NI interrupts the kernel to wake up and notify the blocked application. Finally, the receiving application in the last case registers a notification handler to handle the notification. When the notification arrives, the NI interrupts the kernel to invoke the application signal handler, which executes the registered notification handler.

The first section in Table 27 reports the 4-byte **write** and **writeN** latencies for the four cases. The 5.56μs 4-byte **write** latency is taken from the results reported previously in Table 26. From the current table, an application requires 8.52μs for a 4-byte notifying **writeN** to a remote application that uses **notfSpin** to detect the arriving notification. Like **write**, the receiving NI requires 5.56μs to transfer the 4-byte data payload into the destination host memory. However, it also requires additional 2.96μs, or 53.2% more time, to update the sequence window and the shadow signal counter in host memory so that **notfSpin** can detect the new signal count.

An application requires 16.61μs for a 4-byte notifying **writeN** to a remote application that uses **notfWait** to detect the arriving notification. This latency is almost triple the 4-byte **write** latency and is almost double the 4-byte **writeN** latency using **notfSpin**. This large latency difference highlights the significant overhead in placing the critical path for control transfer through the kernel. With **notfWait**, NetVM requires additional 11.05μs over the base data-only case for the receiving NI to process the sequence window and interrupt the kernel, and for the kernel to wake up and schedule the blocked application.

Finally, an application requires 31.14μs for a 4-byte notifying **writeN** to a remote application that registers a notification handler to process incoming notifications. NetVM requires additional 25.58μs for the receiving NI to process the sequence window, update the notification queue and interrupt the kernel, for the kernel to schedule the application and invoke its signal handler, and for the notification dispatcher in the signal handler to read the notification queue and execute the notification handler.

The second section in Table 27 reports the 4-KB **write** and **writeN** latencies for the same four cases in the first section. The absolute time difference between each of the three notification mechanisms and the base case is similar for both 4-byte and 4-KB transfers, because the notification overheads are fixed regardless of the transfer size. As a result, the corresponding relative differences are significantly lower for 4-KB transfers in the second section of the table as total latency is now dominated by the 47.79μs required to transfer the 4-KB data block.

Detection mode	Overhead (µs)
notfSpin	2.29
notfWait	3.67
notification dispatcher (notfTest)	1.07
notfAck	0.69

Table 28. Overhead for NetVM notification-detection mechanisms.

Table 28 shows the host-processor overhead for detecting a pending signal using each of the three detection mechanisms. Recall from Section 6.1.2 on page 74 that the NI efficiently accumulates notifications using a signal counter on the network interface. It conditionally updates the host memory and interrupts the kernel only for the first pending notification; subsequent notifications only increment the signal count. Therefore, an application can detect pending signals without incurring the high host-interrupt and kernel-processing overheads. Hence, the overheads of **notfSpin**, **notfWait** and the main loop in the notification dispatcher determines the time required to detect these pending notifications.

From the table, **notfSpin** requires only 2.29µs to detect a pending notification; **notfWait** requires 3.67µs, or 60.3% longer, to detect the same notification. The difference is mainly due the system-call overhead into the kernel required by **notfWait**. The main loop in the notification dispatcher requires only 1.07µs to detect any pending notification. This operation is 1.22µs, or 53.3%, faster than **notfSpin** because it does not need to direct the NI to update the shadow signal counter or need to acknowledge the signal. Instead, it simply calls **notfTest** to compare the appropriate signal and acknowledge counters on the network interface. Finally, **notfSpin** and **notfWait**, but not **notfTest**, includes the call to **notfAck**, which requires 0.69µs to acknowledge a signal by incrementing the corresponding acknowledge count on the network interface by one.

9.4 Channels

Detection mode	size (bytes)	Latency (µs)	Throughput(MB/s)
spin (notfSpin)	4	9.55	1.03
block (notfWait)	4	17.34	0.79
block (notfWait)	4K	59.01	155.36

Table 29. Latency and throughput for NetVM channels.

Table 29 shows the NetVM channel latency and throughput for 4-byte and 4-KB transfers. Recall from the description of the channel operations in Table 14 on page 85 that the key data-

transfer operation is a notifying **writeN** by the sending application. As a result, NetVM requires 9.55µs to send a 4-byte packet through the channel if the receiving application is spin waiting for the data with **notfSpin**. This latency is 1.03µs more than the 8.52µs required by the corresponding 4-byte **writeN** reported in Table 27. The sending application requires this additional time for flow control over the channel to ensure that it has sufficient credits on the receiver to send the packet. Similarly, the channel requires 17.34µs to send the same 4-byte packet if the receiver is block waiting for the data with **notfWait**, which is 0.73µs more than the corresponding 4-byte **writeN**. Finally, NetVM channels require 59.01µs to transfer a 4-KB packet.

Throughputs for 4-byte data transfers through the channel are very low, as expected, at 1.03MB/s and 0.79MB/s for the **notfSpin** and **notfWait** approaches respectively. The throughput for 4-KB packets through the channel is much higher at 155.36MB/s, which is almost equal to the 155.46-MB/s 4-KB data-only **write** throughput reported in Table 26, indicating that the channel flow-control overhead has an insignificant effect on large-packet throughput.

9.5 Atomic and synchronization operations

This section presents the latency and overhead for NetVM's atomic and synchronization operations. In particular, it compares the overheads with the basic remote **read**. Recall from Section 8.1 on page 91 that the network interface extends remote reads to implement atomic operations. It fetches the operand from host memory, performs the atomic operation and writes the result to host memory, in addition to sending the reply back to the requesting node. The requesting application spins on its local reply buffer until the result returns. Thus, the decoding and execution time for an atomic operation on the network interface dominates the additional latency over the basic **read**. Furthermore, this duration together with the host-DMA operation to write the updated result to host memory dominates the additional overhead on the network interface.

This section evaluates the atomic and synchronization operations in two parts. This first part reports the latency and overhead of standard and extended atomic operations, and compares them with remote read. The second part reports the MCS lock, wait queue and semaphore synchronization operations.

9.5.1 Atomic operations

Type	Operation	Application Latency (µs)	NI Overhead (µs)
Standard	test_and_set	10.50	4.20
	incr	10.55	4.31
	decr	10.55	4.31
	swap	10.52	4.19
	cswap	10.50	4.30
Wait Queue	enqueue	10.52	4.30
	dequeue	10.74	4.30
	newhead	10.52	4.28
Semaphore	semP	10.52	4.37
	semV	10.80	4.54
Data only	4-byte **read**	9.25	2.72

Table 30. Application Latency and NI overhead for atomic operations.

Table 30 shows the application latency and the NI overhead for both standard and extended atomic operations in three sections and for the basic 4-byte **read** in the fourth section. The application latency measurement is the complete time required by an application to issue an atomic operation on a remote operand and obtain the result by spin waiting on the reply buffer in its local host memory. The NI overhead measurement is the total duration for the network processor on the remote node to fetch the operation from host memory, perform the actual operation, reply to the requesting node, and write the result, if necessary, back to host memory.

The first section lists the standard atomic operations described in Section 8.1 on page 90. The application latencies are almost equal, ranging from 10.50µs to 10.55µs, with only an insignificant 0.5% difference among them. The NI overheads are also similar, ranging from 4.19µs to 4.31µs, with only 2.9% difference among them.

More important, the atomic operations add at most 1.30µs, or only 14.1%, to the 9.25µs application latency over a comparable 4-byte application **read**. However, they add a much higher 54.0% to 58.5% to the NI **read** overhead. On the one hand, atomic and 4-byte **read** operations are similar. Both require a roundtrip network transaction, both access a 4-byte remote word operand, and both spin on local host memory waiting for the operation to complete. On the other hand, the remote NI incurs a significantly higher overhead, up to 1.59µs, for processing an atomic operation compared to processing a **read** operation.

The second and third sections list the extended NetVM wait-queue and semaphore atomic operations described in Sections 8.3 and 8.4 on pages 94 and 99 respectively. From the table,

these operations do not add significant application latency or NI overhead over standard atomic operations. The results represent the maximum timings, which traces through the longest code path for each operation. The application latencies are only between 0.2% and 2.9% slower than the fastest standard atomic operation, or are between 13.7% and 16.8% slower than the 4-byte **read**. Similarly, the NI overheads are only between 2.6% and 8.4% higher than the smallest overheads for standard atomic operations, but they are between 57.4% and 66.9% higher than the **read** overhead.

The NetVM extended atomic operations add only a small latency and overhead compared to standard atomic operations. The extended operations fetch a larger operand (up to five words instead of one) from host memory, perform a slightly more complex operation on the network interface, and return a larger result (up to three words instead of one) to the requesting application. However, they add only up to 2.9% to the application latency and up to 16.8% to the NI overhead over standard atomic operations.

9.5.2 Synchronization operations

Type	Operation	Overhead (µs)	
MCS Lock	acquire	10.53	
	release	10.55 (no successor)	0.67 (signal successor)
Wait Queue	insertQ	11.87 (first in queue)	12.47 (append to predecessor)
	removeQ	11.64	
Semaphore	Swait	10.55	
	Ssignal	10.55 (no waiter)	11.88 (signal waiter)

Table 31. Application overhead for synchronization operations.

Table 31 shows the total application overhead for the NetVM synchronization operations in three sections. These operations build on the atomic operations reported in the preceding section. If there are two numbers reported in the third column of the table, the second measurement on the right represents the total overhead including issuing the **write** or **writeN** required by the operation to synchronize with a remote process. The actual latency to transfer control depends on the notification-detection mechanism used by the waiting process, which Section 9.3 reported.

The first section of the table lists the results for MCS lock operations. **acquire** requires 10.53µs to call **swap** and acquire an available lock. **release** requires 10.55µs to call **cswap** and return

the lock back to the central lock data structure when there are no successors in the lock chain. If one is present instead, the releasing application only requires 0.67μs overhead to *issue* a notifying **writeN** to hand over the lock to the successor. Recall that **release** does not need to call **cswap** to update the lock data structure if it detects a successor in its local host memory.

The second section of the table lists the results for the wait-queue operations. **insertQ** requires 11.87μs to call **enqueue** and insert the calling application as the first process in the wait queue. If the queue is not empty, **insertQ** requires 12.47μs to additionally append the calling application to the wait queue instead. The 0.6μs difference is the additional time required by **insertQ** to issue the **write** that updates its predecessor, which is the current tail of the queue. **removeQ** requires 11.64μs to call **dequeue**, which obtains the current front process of the queue, and to issue a notifying **writeN** to wake it up.

The third section of the table lists the results for the semaphore operations. **Swait** requires 10.55μs to perform a semaphore *P* operation on a positive semaphore count. If the count was zero, **Swait** will block waiting for a wakeup notification from an **Ssignal** by another process. **Ssignal** requires 10.55μs to perform a semaphore *V* operation if there are no processes blocked on the semaphore. If there is at least one process blocked waiting on the semaphore, **Ssignal** requires 11.88μs for the NI to dequeue that process and for the signaling application to issue a notifying **writeN** to it.

Table 31 also shows the advantage of using NetVM's approach to implementing wait queues and semaphores by minimizing the total number of network transactions. Each of these operations require only one roundtrip network transaction, in the common case, and possibly an additional remote **write** or **writeN** to signal or append to a remote process. Thus, the latencies for these operations are approximately equal to the latency of a standard atomic operation. The traditional approach for implementing wait queues and semaphores on non-cache-coherent shared memory machines require at least three phases: acquire, manipulate and release a shared synchronization data structure. Each phase would normally require one, but usually more, roundtrip atomic and regular operations. Thus, the total latency for the traditional approach can be significantly higher than the NetVM approach.

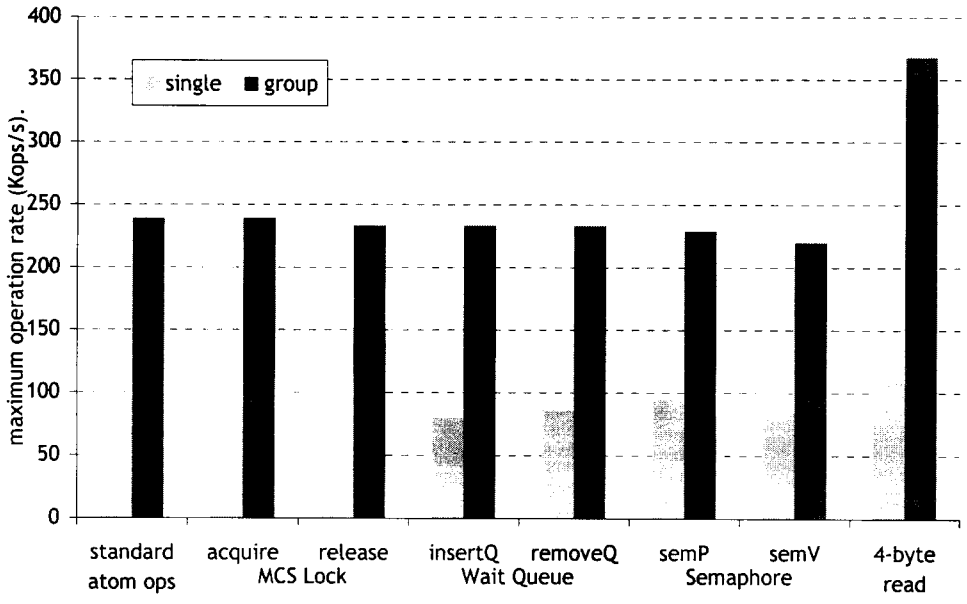

Figure 28. Maximum single- and multiple-application operation rates.

Figure 28 shows the maximum operation rate that a single application or a group of applications can generate. For a single application, the total overhead reported in Table 31 limits its maximum rate, because it can only issue a new operation once the previous one completes. Hence, the rate is computed by taking the reciprocal of the corresponding worst-case application overhead in Table 31. However, a group of applications can issue atomic operations in parallel. The network interface on the node that stores the shared synchronization data structures has to serialize all the atomic operations on that data structure. Hence, the NI overhead for sequentially processing the atomic operations limits the maximum rate, which is computed by taking reciprocal of the corresponding NI overhead in Table 30.

Figure 28 also shows that the maximum operation rates for NetVM-specific operations are approximately equal to the standard atomic operation rate and are not significantly slower than even the basic remote **read** rate. The **read** rate reflects the absolute maximum achievable rate with the current implementation and the standard atomic operation rate reflect the maximum rate if the network interface has to additionally parse, execute and update the remote operand. From the figure, the rates for standard atomic operation are slower than **read** by 12.0% and 35.1% for single and multiple application requests respectively. The rates for NetVM-specific synchronization operations, including all application-level overheads, are slower than standard atomic operations only by up to 15.8% and 7.9% for single and multiple application requests respectively. Thus, NetVM is able to support high-level synchronization operations that execute close to the maximum rate achievable by the network hardware.

10 Conclusion and future work

"i implemented, i measured, i'm done"
norm hutchinson

This chapter provides a summary of the thesis and discusses the possible research directions and future work for NetVM.

10.1 Summary

User-mode access, zero-copy transfer, and sender-managed communication are recognized as essential to high-performance communication. The main challenge for application-level DMA is to resolve the addressing mismatch between user-space applications and the network interface. Existing systems pre-pin a set of buffers in physical memory and confine both source and target data to these pinned buffers. This approach has two drawbacks: it hinders the operating system from managing memory effectively and it places a burden on the programmer to ensure network-addressable data structures are stored in pinned memory.

NetVM integrates the network interface with the host virtual memory system to provide a simple and powerful communication model with minimal host software overhead. It allows an application with the appropriate permissions to achieve zero-copy data transfer between any virtual address on the source to any virtual address on the destination, without requiring the operating system to pre-pin any of the pages. NetVM addresses the issue of pinning by maintaining a shadow page table on the network interface that the operating system updates whenever it maps or unmaps a page in host memory. The NI uses the table to briefly lock and translate the virtual address of the page whenever it accesses that page with a DMA transfer. This NI-initiated page lookup and locking mechanism is fast enough to lie in the critical path and, thus, NetVM can zero-copy transfer data to and from any *unpinned* resident page in host memory. The NI redirects transfers destined to nonresident pages through an intermediate system buffer, where the operating system completes the transfer in a one-copy scheme after paging in the target page. Thus, NetVM ensures reliable delivery without data re-transmission protocols. Integration with the host VM system required only four simple changes, these changes are easy to interject and add little overhead.

The evaluation of the prototype implementation for the Myrinet network hardware highlights the costs and benefits of using the NetVM memory-management approach. NetVM adds only 10.61% overhead to the initial application page fault on a zero-filled page. The NI-based page-locking cost is significantly lower than the host-based dynamic pinning approaches. The analysis indicates that the pinned-page cache's hit rate must be sufficiently high, up to 94.5%, before it can better NetVM's pinning cost. Compared to the static pinning approaches, NetVM adds only a modest 1.5% to 5.0% to **write** latency and adds virtually no overhead to throughput.

The basic RDMA communication model supports only data transfer. To support control transfer, NetVM implements eventcount-based notifications for applications to detect the completion of a data transfer or the arrival of a remote signal event. The sending application selectively signals a notification by including its identifier in a **write**; the receiving application detects the notification either by busy waiting, by block waiting, or by registering a handler to trigger whenever a notifying **writeN** completes. A sequence window enforces ordered notifications for synchronized writes over an out-of-order delivery network.

The range of notification-detection mechanisms allows an application to make the appropriate tradeoffs when waiting for remote signals. On the one hand, it can busy wait, which consumes host processor time, to detect notifications with low latency (2.96μs higher than a 4-byte data-only **write**). On the other hand, it can block wait, which does not consume host processor time, to detect notifications with 8.09μs higher detection latency compared to busy waiting. A third alternative is to asynchronously register a user handler, which automatically triggers when a notification arrives, with 14.53μs higher detection latency compared to block waiting.

Traditional synchronization algorithms for non-cache-coherent multiprocessors that use standard atomic operations typically require several network transactions to complete a synchronization operation. NetVM exploits the programmable network hardware by augmenting the network interface with synchronization primitives and using them in the user library to support higher-level wait-queue and counting-semaphore operations. As a result, application-level synchronization operations require a reduced number of network transactions to complete. The synchronization primitives have a low implementation and execution cost. The measurements indicate that *application latencies* for the high-level wait queue and semaphore operations are less than 18.2% slower than the low-level standard atomic operations, which in turn are only less than 14.1% slower than the basic remote **read**. Thus, NetVM significantly reduces the application latencies and network-interface overheads for these high-level synchronization constructs.

NetVM supports all the benefits of user-mode-access and zero-copy-transfer communications without the cost of pinning host memory. It provides programmers with a natural view for remote memory. It requires only minor changes, without introducing significant overheads, to the host operating system and still allows the operating system to retain full control over its host memory. NetVM also supports a range of control-transfer strategies that allow the application to tradeoff reducing signaling latency for reducing host-processor overhead. Finally, NetVM supports fine-grained high-level synchronization operations while reducing the required number of network transactions. It only requires minor extensions to the network interface, which are easy to implement and have low overheads.

10.2 Future work

Many interesting issues arose during the design and implementation of NetVM. The following paragraphs examine some potential research directions from this thesis work and some possible improvements to NetVM.

The challenge of implementing higher-level protocols on programmable network interfaces has been around for some time. There is a constant tension between implementing too much, which overloads the network processor, and implementing too little, which provides insufficient benefit to applications. Often, the tradeoffs depend on the capabilities of the network interface, the overhead of the intended operation and the cooperation between the host and NI processors to implement the operation. In the case of NetVM, the host operating system performs all the necessary work to update the shadow page table, which is completely stored and maintained on the network interface. Thus, the NI operations to access the data structures are relatively lightweight. The extended atomic operations on the network interface are also lightweight. However, they can provide substantial performance improvement because of their tight coupling with the host operations. As network hardware improves, this delicate balance between host and NI processing is likely to shift in favor to migrating more functionality to the network hardware. Thus, it is important to understand the characteristics of workloads and operations that make them suitable for deploying in the network hardware.

NetVM demonstrated that it is possible to significantly improve the latency of synchronization operations by augmenting the network interface with simple operation-specific extensions that are easy to implement, execute efficiently, and have low overhead. In particular, NetVM implements the operations to support wait queues and counting semaphores, which includes only five simple atomic operations on the network interface. Other synchronization operations can also benefit from this approach. Indeed, current research is looking into using the network in-

terface to assist group operations such as barriers, reductions, and broadcasts. However, a key issue to supporting these operations is the relatively slow processor and small memory available on the network interface. Thus, designing strategies that are both scalable and efficient on the network interface is important. The MCS-inspired approach in NetVM is only one possible strategy, investigating alternate strategies and tradeoffs can be interesting.

An application may require better access control over its exported segments. Currently, a remote application has full read-write access to an imported segment. The exporting application has no way to limit remote applications to read-only access, for example, when it wants to publish data that remote applications can look up but not modify. Adding *page-access attributes* to NetVM is relatively straightforward; the shadow page table additionally stores the page-access mode for each page, which the NI verifies before accessing the page with DMA. The host VM module updates the page table based on the access-mode arguments in the export system call. To achieve even finer-grained access control, the import-export module can export different export names, each with a different access mode, to the same virtual address range. In this way, an application can allow some remote applications to acquire read-only access and other applications to acquire read-write access over the same exported memory.

The current implementation for atomic operations requires the NI to fetch the operand from host memory, perform the operation and then write it back to host memory. The DMA transfer times to fetch and write back the operand is a significant overhead on the network interface. For frequent fine-grained synchronization operations, *caching* the operand on the network interface can significantly reduce the overhead by eliminating the two small DMA transfers for frequently accessed data structures used in atomic operations.

NetVM does not support shared virtual memory queues. Currently, the receiving application has to set up a dedicated channel for each sender to implement many-to-one communication. To implement shared queues, the network interface has to multiplex messages from different senders into a single virtual memory queue in host memory. Thus, it needs to maintain metadata for the host queue, which includes the base virtual address, size and write cursor. Because the sending applications do not know, in advance, the final addresses of the messages that are deposited into the queue, the network interface has to maintain the write cursor in order to determine the virtual, and hence physical, address to update in the queue. A key issue is dealing with the high NI-processor overhead to compute the VPN-hashkey from the write cursor. Caching the result and recomputing it only when the cursor crosses a virtual page boundary mitigates some of the overhead, especially for small messages.

Related to shared virtual-memory queues is receiver-managed queues with send-receive semantics. To implement receiver-managed queues, the network interface has to maintain a receive queue of virtual-address pointers and match them against incoming messages. The application posts a receive operation by appending the computed VPN-hashkey and the size of the buffer in the receive queue. The NI consumes an entry from the receive queue whenever a message destined for the queue arrives from the network. A key issue is dealing with underflow of the receive queue and using a redirection buffer, such as in VMMC-2, can help.

NetVM is a low-level communication layer. Although applications can directly use NetVM to communicate, many parallel and distributed applications use higher-level libraries, such as MPI or OpenMP, to simply the implementation and to improve portability. Hence, it may be useful to implement these libraries on top of NetVM. Previously, Gu implemented MPIN [33] for an earlier version of NetVM using only RDMA **write** for data transfer and busy waiting for synchronization. MPIN can be revised to take advantage of the richer API provided by the current version of NetVM.

Bibliography

1. Adve, S. V. and K. Gharachorloo, *Shared Memory Consistency Models: A Tutorial.* IEEE Computer, 1996. **29**(12): p. 66-76.
2. American National Standards Institute, *The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A.* 1983.
3. Anderson, D., et al. *Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet.* in *Usenix Technical Conference.* 1998.
4. Anderson, T. E., E. D. Lazowska, and H. M. Levy, *The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.* Performance Evaluation Review, 1989. **17**: p. 49-60.
5. Anderson, Thomas E., et al. *The Interaction of Architecture and Operating System Design.* in *4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV).* 1991.
6. Bell, Christian and Dan Bonachea. *A New DMA Registration Strategy for Pinning-Based High Performance Networks.* in *Workshop on Communication Architecture for Clusters (CAC'03).* 2003.
7. Bell, Gordon, *1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac?* Communications of the ACM, 1996. **39**(3): p. 11-15.
8. Bhoedjang, R. A. F., T. Ruhl, and H. Bal, *Design Issues for User-Level Network Interface Protocols on Myrinet.* IEEE Computer, 1998.
9. Blumrich, M., et al. *Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer.* in *International Symposium on Computer Architecture.* 1994.
10. Boden, Nanette J., et al., *Myrinet: A Gigabit-per-Second Local Area Network.* IEEE Micro, 1995. **15**(1): p. 29-36.
11. Bradford, Jeffrey P. and Seth Abraham. *Efficient Synchronization for Multithreaded Processors.* in *Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC) held in conjunction with the 4th International Symposium on High-Performance Computer Architecture (HPCA).* 1998.
12. Buzzard, Gregory D., et al. *An Implementation of the Hamlyn Sender-Managed Interface Architecture.* in *Operating Systems Design and Implementation.* 1996.
13. Chen, Yuqun, et al. *UTLB: A Mechanism for Address Translation on Network Interfaces.* in *Architectural Support for Programming Languages and Operating Systems.* 1998.
14. Chesney. *The Meiko CS-2 System Architecture.* in *ACM Symposium on Parallel Algorithms and Architectures.* 1993.

15. Chun, Brent N., A. M. Mainwaring, and D. E. Culler, *Virtual Network Transport Protocols for Myrinet*. IEEE Micro, 1998. **18**(1).
16. Coady, Yvonne, Joon Suan Ong, and Michael J. Feeley. *Using Embedded Network Processors to Implement Global Memory Management in a Workstation Cluster*. in *IEEE Symposium on High Performance Distributed Computing*. 1999.
17. Compaq, Intel and Microsoft,. *Virtual Interface Architecture Specification 1.0. http://www.viaarch.org*. 1997.
18. Craig, Travis S., *TR 93-02-02: Building FIFO and Priority-Queuing Spin Locks from Atomic Swap*. 1993, University of Washington: Seattle, Washington.
19. DAFS Collaborative. *Direct Access File System Protocol v1.0. http://dafscollaborative.org*. 2001.
20. Damianakis, Stefanos N., Yuqun Chen, and Edward W. Felten, *TR-525-96: Reducing Waiting Costs in User-Level Communication*. 1996, Princeton University: New Jersey.
21. Denning, Peter J., T. Don Dennis, and Jeffery A. Brumfield, *Low Contention Semaphores and Ready Lists*. Communications of the ACM, 1981. **24**(10): p. 687-699.
22. Dijkstra, E. W., *Co-operating Sequential Processes*, in *Programming Languages*, F. Genuys, Editor. 1968, Academic Press: New York. p. 43-112.
23. Dijkstra, E. W., *The Structure of the "THE" Multiprogramming System*. Communications of the ACM, 1968. **11**(5): p. 341-346.
24. Dubnicki, C., et al. *Design and Implementation of Virtual Memory-Mapped Communication on Myrinet*. in *11th International Parallel Processing Symposium*. 1997.
25. Dunning, D., et al., *The Virtual Interface Architecture*. IEEE Micro, 1998. **18**(2).
26. Eicken, T. von, et al. *Active Messages: A Mechanism for Integrated Communication and Computation*. in *International Symposium on Computer Architecture*. 1992.
27. Feeley, Michael J., et al. *Implementing Global Memory Management in a Workstation Cluster*. in *Symposium on Operating Systems Principles*. 1996.
28. Fillo, Marco and Richard B. Gillet, *Architecture and Implementation of MEMORY CHANNEL 2*. Digital Technical Journal, 1997. **9**(1): p. 27-41.
29. FreeBSD. *http://www.freebsd.org*.
30. Fu, S. and N. Tzeng, *A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors*. IEEE Transactions on Parallel and Distributed Systems, 1997. **8**(6): p. 628-639.
31. Gillett, Richard B., *Memory Channel Network for PCI*. IEEE Micro, 1996. **16**(1): p. 12-18.
32. Goodman, J., M. Vernon, and P. Woest. *Efficient Synchronization Primitives for Large-Scale Cache-Coherent Shared-Memory Multiprocessors*. in *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 1989.

33. Gu, YanPing, *The MPI Implementation on NetVM. MSc Thesis*, in *Department of Computer Science*. 2000, University of British Columbia: Vancouver.
34. Gustavson, David B., *The Scalable Coherent Interface and Related Standards Projects*. IEEE Micro, 1992. **12**(1): p. 10-22.
35. Gustavson, David B. and Qiang Li. *Local-Area MultiProcessor: the Scalable Coherent Interface*. in *SCIzzL, The Association of SCI Local-Area MultiProcessor Users, Developers, and Manufacturers (1994)*. 1994.
36. Hansen, Per Brinch, *Operating System Principles*. 1973, Englewood Cliffs, New Jersey: Prentice-Hall.
37. Hoare, C. A. R., *Monitors: An Operating System Structuring Concept*. Communications of the ACM, 1974. **17**(10): p. 549-557.
38. Homewood, M. and M. McLaren. *Meiko CS-2 interconnect Elan Elite design*. in *IEEE Hot Interconnects Symposium*. 1993.
39. Huang, T. *Fast and Fair Mutual Exclusion for Shared Memory Systems*. in *19th IEEE Conference on Distributed Computing Systems*. 1999.
40. InfiniBand Trade Association, *InfiniBand Architecture Specification Volume 1, Release 1.0.a*. 2001.
41. Kägi, A. and J. Goodman. *Efficient Synchronization: Let Them Eat QOLB*. in *24th International Symposium on Computer Architecture*. 1997.
42. Kay, Jonathan and Joseph Pasquale. *The Importance of Non Data Touching Processing Overheads in TCP/IP*. in *SIGCOMM93*. 1993.
43. Kuskin, Jeffrey, et al. *The Stanford FLASH Multiprocessor*. in *21st International Symposium on Computer Architecture*. 1994.
44. Li, Kai, *Shared Virtual Memory on Loosely Coupled Multiprocessors. PhD Thesis*. 1986, Yale University.
45. Li, Kai. *A Shared Virtual Memory System for Parallel Computing*. in *Proceedings of the International Conference on Parallel Processing*. 1988.
46. Lubachevsky, B., *An Approach to Automating the Verification of Compact Parallel Coordination Programs*. Acta Informatica, 1984. **14**: p. 125-169.
47. Mainwaring, Alan M. and David E. Culler. *Design Challenges of Virtual Networks: Fast General-Purpose Communication*. in *7th Symposium on Principles and Practices of Parallel Programming*. 1999.
48. Maquelin, Olivier, et al. *Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling*. in *International Symposium on Computer Architecture*. 1996.
49. Meiko Scientific, *Meiko CS-2 Communications Processor*. *http://www.meiko,com/info/CommsProcessor/CommsProcessor.html*.
50. Meiko Scientific, *ELAN Communications Processor Reference Manual*. 1993.

51. Mellor-Crummey, J. and M. Scott, *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.* ACM Transactions on Computer Systems, 1991. **9**(1): p. 21-65.

52. Mukherjee, Shubhendu S. and Mark D. Hill, *TR #1340: A Survey of User-Level Network Interfaces for System Area Networks.* 1997, Computer Science Department, University of Wisconsin-Madison. p. 26.

53. Myricom, *Myrinet Lanai9 Programmer's Documentation.* http://www.myri.com/vlsi/LANai9.pdf, 2000.

54. Myricom, *Myrinet PCI64 Programmer's Documentation.* http://www.myri.com/myrinet/PCI64/PCI64-programming.pdf, 2001.

55. Myricom. *The GM Message Passing System.* 2002.

56. Nieplocha, J., V. Tipparaju, and D. Panda. *Protocols and strategies for optimizing performance of remote memory operations on clusters.* in *Workshop Communication Architecture for Clusters (CAC02) of IPDPS'02.* 2002.

57. Nikolopoulos, Dimitrios S. and Theodore S. Papatheodorou. *Fast Synchronization on Scalable Cache-Coherent Multiprocessors using Hybrid Primitives.* in *14th International Conference on Parallel and Distributed Processing Symposium.* 2000.

58. Osterhout, John K. *Why Aren't Operating Systems Getting Faster as Fast as Hardware?* in *USENIX Summer Conference.* 1990.

59. Pakin, S., M. Lauria, and A. Chien. *High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet.* in *Supercomputing.* 1995.

60. Pakin, S., M. Lauria, and A. Chien. *The Fast Messages (FM) 2.0 Streaming Interface.* in *Usenix'97.* 1996.

61. Pakin, Scott, Vijay Karamcheti, and Andrew Chien, *Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs.* IEEE Concurrency, 1997. **5**: p. 60-73.

62. Peterson, G. L., *Myths about the Mutual Exclusion Problem.* Information Processing Letters, 1981. **12**(3): p. 115-116.

63. Petrini, Fabrizio, et al. *The Quadrics Network: High-Performance Clustering Technology.* in *Hot Interconnects 9.* 2001.

64. Petrini, Fabrizio, et al., *The Quadrics Network: High-Performance Clustering Technology.* IEEE Micro, 2002. **22**(1): p. 46-57.

65. Pfister, Gregory F., *In Search of Clusters.* 2nd ed. 1998, New Jersey: Prentice Hall.

66. Prylli, L. and B. Tourancheau, *BIP: A New Protocol Designed for High Performance Networking on Myrinet.* Lecture Notes in Computer Science, 1998. **1388**: p. 472-485.

67. Reed, David P. and Rajendra K. Konadia, *Synchronization with Eventcounts and Sequencers.* Communications of the ACM, 1979. **22**(2): p. 115-123.

68. Riddoch, David, et al., *Tripwire: A Synchronization Primitive for Virtual Memory Mapped Communication.* Journal of Interconnection Networks, 2001. **2**(3): p. 345-364.

69. Schoinas, Ioannis and Mark D. Hill. *Address Translation Mechanisms In Network Interfaces.* in *4th International Symposium on High Performance Computer Architecture.* 1998.

70. Shanley, Tom and Don Anderson, *PCI System Architecture.* 3rd ed. 1995: Addison-Wesley.

71. Steenkiste, P.A., *A Systematic Approach to Host Interface Design for High-Speed Networks.* IEEE Computer, 1994. **27**(3): p. 47-57.

72. Stunkel, Craig, *The SP2 High-Performance Switch.* IBM System Journal, 1995. **34**(2).

73. Sun Microsystems, Inc, *SBus UltraSPARC Port Architecture to SBus Interface Chip (U2S) Manual. http://www.sun.com/embedded/databook/pdf/manuals/805-0168-01.pdf.* 1997.

74. Tezuka, H., et al. *Pin-Down Cache: A Virtual Memory Management Technique for Zero-Copy Communication.* in *12th International Parallel Processing Symposium.* 1998.

75. Thekkath, Chandramohan A., Henry M. Levy, and Edward D. Lazowska. *Separating Data and Control Transfer in Distributed Operating Systems.* in *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems.* 1994.

76. Thornley, John and K. Mani Chandy. *Monotonic Counters: A New Mechanism for Thread Synchronization.* in *14th International Parallel and Distributed Processing Symposium.* 2000.

77. Wang, R. Y., et al. *Modeling and Optimizing Communication Pipelines.* in *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.* 1998.

78. Welsh, M., A. Basu, and T. von Eicken. *Incorporating Memory Management into User-Level Network Interfaces.* in *Hot Interconnects V.* 1997.

79. Wirth, N., *On Multiprogramming, Machine Coding, and Computer Organization.* Communications of the ACM, 1969. **12**(9): p. 489-498.

80. Yocum, Ken, et al. *Cut-Through Delivery in Trapeze: An Exercise in Low-Latency Messaging.* in *High-Performance Distributed Computing.* 1997.

Appendix A
Pipelined DMA transfer model

This section presents the description and results of a simple pipelined DMA-transfer model that characterizes a NetVM 2-fragment RDMA **write**. The goal of this model is to analyze the costs and benefits of fragmenting a transfer, which was evaluated in Section 9.2.4 on page 114. Recall, from Section 5.2 on page 63, that the NetVM user library splits a large transfer into multiple page-aligned fragments. A 4096-byte **write** that spans the page boundary in the source or destination memory will divide into two, or possibly three, fragments. Also recall from the NI operation description in Section 5.4 on page 65 that, in general, a **write** operation for a single fragment requires three DMA stages: source-host DMA from source host memory into the network interface, wire-DMA transfer from source to destination network interface, and destination-host DMA transfer from the destination network interface to target host memory. For small fragments less than or equal to 96 bytes, a programmed-IO transfer from source host memory into network interface memory replaces the source-host DMA transfer. Wang et al. [77] provide a comprehensive analysis for modeling and optimizing general communication pipelines.

This model makes three assumptions. First, it assumes that the host and network processors are infinitely fast and require zero time to handle the command descriptor and network message. The DMA transfer stages dominate the total latency of a large **write**; the combined processor overheads add only a small constant time to the latency. Furthermore, some of these overheads overlap with the DMA transfer of preceding stages and thus do not contribute to the total latency.

Second, the model assumes that the first stage is always a source-host DMA transfer, instead of adaptively switching between programmed IO and host DMA depending on the fragment size. This assumption simplifies the computation required for that stage and results in only a small

time difference based on the programmed-IO–host-DMA comparison in Section 3.4.1 on page 41.

Third, the model assumes that the network link is not congested and that the fragments do not reorder in transit. A fragment that leaves the source network interface immediately arrives into the destination network interface, ignoring the negligible network propagation delay.

A.1 Model Representation

$$
\begin{aligned}
hdma1(f1) &= \frac{size(f1)}{tp(hdma1)} \\
wdma(f1) &= hdma1(f1) + \frac{size(f1)}{tp(wdma)} \\
hdma2(f1) &= wdma(f1) + \frac{size(f1)}{tp(hdma2)} \\
hdma1(f2) &= hdma1(f1) + \frac{size(f2)}{tp(hdma1)} \\
wdma(f2) &= \max\left(hdma1(f2), wdma(f1)\right) + \frac{size(f2)}{tp(wdma)} \\
hdma2(f2) &= \max\left(wdma(f2), hdma2(f1)\right) + \frac{size(f2)}{tp(hdma2)}
\end{aligned}
$$

Figure 29. 2-fragment pipelined DMA-transfer model.

Figure 29 shows the 2-fragment pipelined DMA-transfer model. The model computes the completion times for each DMA stage for the two fragments. **size**(*F*) is the size of fragment *F*, f1 for the first and f2 for the second. **tp**(*S*) is the measured throughput for DMA stage *S*, the source-host DMA stage is **hdma1**, the wire-DMA stage is **wdma**, and the destination-host DMA stage is **hdma2**. The ratio of **size**(*F*) over **tp**(*S*) is simply the duration required for a particular DMA stage *S* to transfer fragment *F*.

Each line in Figure 29 computes the completion time of a DMA-transfer stage. For example, **hdma1**(f1) is the completion time of the source-host DMA transfer for first fragment. Similarly, **wdma**(f2) is the computed completion time of the wire-DMA transfer for the second fragment, relative to the start of the source-host DMA for the first fragment. The total latency of the complete transfer is **hdma2**(f2), which is the completion time for the destination-host transfer of the second fragment into the destination host memory.

There are two key DMA-transfer constraints represented in the model. First, the wire-DMA stage for the second fragment, **wdma**(f2), cannot begin until its preceding source-host DMA

stage completes at time **hdma1**(f2) *and* the wire-DMA engine is available after transferring the first fragment onto the wire at time **wdma**(f1). The second to last equation in Figure 29 represents this constraint; it uses the slower of the two dependent preceding stages, **hdma1**(f2) and **wdma**(f1), as the starting time for the **wdma**(f2) stage.

Second, the destination-host DMA stage for the second fragment, **hdma2**(f2), cannot begin until its preceding wire-DMA stage completes at time **wdma**(f2) *and* the host-DMA engine on the destination network interface is available after transferring the first fragment to host memory at time **hdma2**(f1). The last equation in the same figure represents this constraint, this time it uses the slower of the two dependent preceding stages, **wdma**(f2) and **hdma2**(f1), as the starting time for the **hdma2**(f2) stage.

Parameter	Value
size(f1)	(4096 - *offset*) bytes
size(f2)	*offset* bytes
tp(hdma1)	437 MB/s
tp(wdma)	160 MB/s
tp(hdma2)	492 MB/s

Table 32. Pipelined DMA-transfer model parameters.

Table 32 lists the input parameters to the model. **size**(f1) and **size**(f2) are the sizes of the first and second fragments respectively. The input sizes are based on the misaligned-large-transfers experiment performed in Section 9.2.4 on page 114. In that experiment, the application on the source node writes a 4-KB data block with varying page offsets into remote destination memory. In each **write**, NetVM splits the transfer into two fragments; the first fragment is (4096 - *offset*) bytes large at *offset* bytes from the start of the first page and the second fragment is *offset* bytes large from the start of the subsequent page.

tp(hdma1), **tp**(wdma) and **tp**(hdma2) are the measured 4096-byte throughput of the source-host DMA, the wire DMA, and the destination-host DMA stages respectively. For simplicity, the model uses only the best-case throughput for these parameters, which underestimates the actual DMA-setup time for smaller transfers.

A.2 Model Results

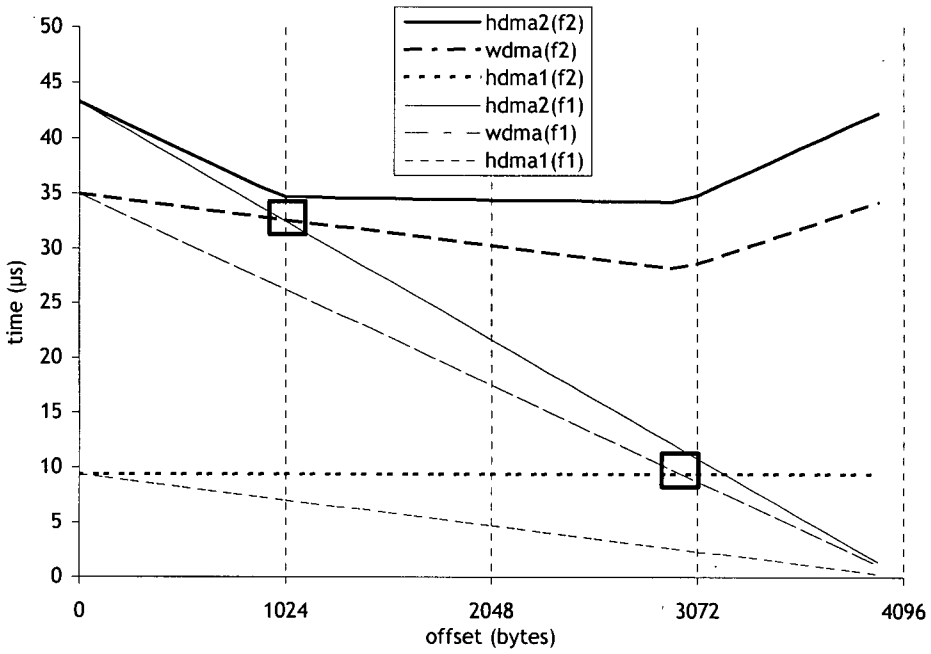

Figure 30. Model results for the latency of each pipelined-DMA stage.

Figure 30 shows the results of the model with the input page offset ranging from 0 to 4096 bytes. Each line in the figure represents the computed completion time of a DMA stage. The topmost line, which is **hdma2**(f2), represents the total latency of the 2-fragment **write** transfer. This three-segment trough line has a similar shape to the measured latency, shown in Figure 26 on page 115, of the real system with the same *offset* input.

The three segments of the latency line represent three distinct configurations in the overlap among the DMA stages. The left box in Figure 30 shows the crossover condition from the first segment to the second segment, when **wdma**(f2) exceeds **hdma1**(f1). Similarly, the right box in the same figure shows the crossover condition from the second segment to the third segment, when **hdma1**(f2) exceeds **wdma**(f1). The following paragraphs describe the DMA pipeline interaction in each of these three segments.

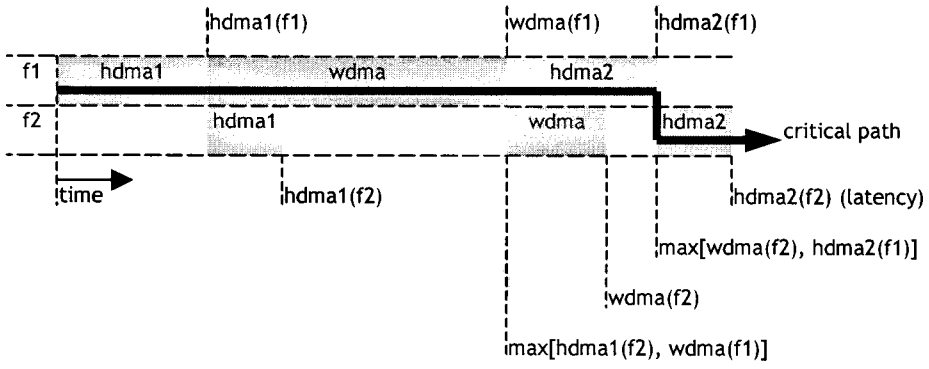

Figure 31. DMA operation timeline in the first segment (e.g. offset = 500).

Figure 31 shows the DMA-transfer timeline in the first segment of the **hdma2**(f2) line in Figure 30. The timeline shows the DMA transfer pipeline for each of the two fragments, f1 and f2. Each pipeline has three stages (**hdma1**, **wdma** and **hdma2**) and each stage can only transfer one fragment at any one time. A DMA stage can only begin when its dependent preceding stage for the same fragment completes and when the DMA engine is available. The thick solid line indicates the latency critical path of the transfer, terminating at time **hdma2**(f2).

From Figure 31, NetVM transfers a large first fragment followed by a smaller second fragment. For a fixed 4096-byte write, increasing the offset reduces the size of the first fragment but adds to the size of the second fragment by the same amount. A smaller first fragment shortens the entire f1 pipeline. As a result, the **hdma1**(f2) and **wdma**(f2) stages begin earlier and overlap more with the **wdma**(f1) and **hdma2**(f1) stages respectively. The **hdma2**(f2) stage also begins earlier but requires a slightly longer time to complete because of an increasing fragment size. However, the contracting DMA pipeline for the first fragment still dominates the latency critical path. Hence, the overall effect is a reduction in total latency and an increase in DMA overlap between the two fragments.

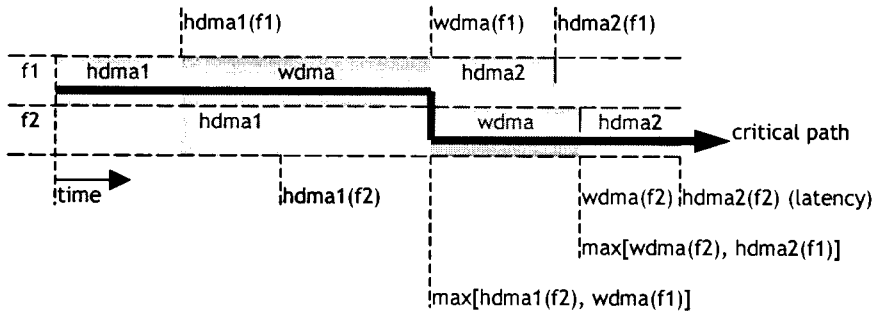

Figure 32. DMA operation timeline in the second segment (e.g. offset = 2000).

Figure 32 shows the DMA-transfer timeline in the second segment of the **hdma2**(f2) line in Figure 30. At the transition between the first and second segments, the completion time **wdma**(f2) exceeds **hdma2**(f1), indicated by the left box in Figure 30. Past this point, the NI on the destination node now has to wait for the entire second fragment to arrive from the wire, instead of previously waiting for the destination-host DMA for the first fragment to complete, before it can initiate the destination-host DMA transfer for the second fragment.

Increasing the offset does not significantly affect the latency in this segment. The latency critical path now passes through **hdma1**(f1), **wdma**(f1), **wdma**(f2) and **hdma2**(f2) stages. The total time in the wire-DMA stages for both fragments is fixed. Increasing the offset reduces the first fragment size, which further shortens the f1 pipeline. However, any latency savings from a shorter **hdma1**(f1) stage is negated by an equally lengthening **hdma2**(f2) stage. Therefore, the net effect is that latency remains about the same throughout the second segment. Interestingly, the model predicts that total latency should actually decrease slightly as offset increases because the destination-host DMA has a slightly higher throughput than the source-host DMA. However, the actual experiment did not detect this small effect, probably because the model overestimated the actual throughput difference between the two host-DMA stages.

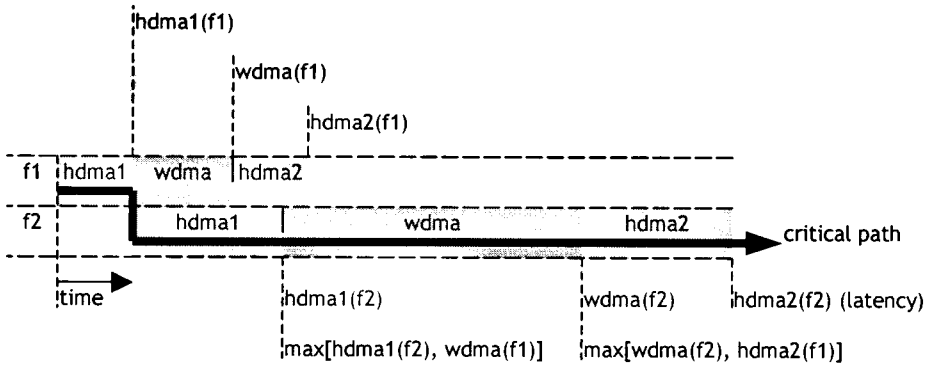

Figure 33. DMA operation timeline in the third segment (e.g. offset = 3500).

Figure 33 shows the DMA-transfer timeline in the third segment of the **hdma2**(f2) line in Figure 30. At the transition between the second and third segments, the completion time **hdma1**(f2) exceeds **wdma**(f1), indicated by the right box in Figure 30. Past this point, the NI now has to wait for the entire second fragment to arrive from host memory, instead of previously waiting for the wire DMA for the first fragment to complete, before it can initiate the wire-DMA transfer for the second fragment. Increasing the offset increases the total latency in this segment. The expanding DMA pipeline for the second fragment now dominates the latency critical path. The net effect of increasing offset is an increase in total latency and a decrease in DMA overlap between the two fragments.

Although the modeled-system latency in Figure 30 is similar in shape to the measured real-system latency in Figure 26 on page 115, there are two key differences in the graphs. First, the modeled latency is about 6μs lower than the real system, because the model ignores all the processor overheads required to handle the command descriptor and network message. Second, the slopes of the first and third segments of the modeled-system trough line is steeper compared to the real system, because the model underestimates the DMA-setup time by using only the best-case throughput as the model input parameter for each DMA stage. As a result, the model predicts a greater incremental latency improvement in the first segment of the graph and a greater incremental latency increase in the third segment compared to the measured real system.