Appears in the 4th International Symposium on High Performance Computer Architecture

Address Translation Mechanisms in Network Interfaces

Ioannis Schoinas and Mark D. Hill
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706-1685, USA
{schoinas,markhill}@cs.wisc.edu

Abstract

Good network hardware performance is often squandered by overheads for accessing the network interface (NI) within a host. NIs that support user-level messaging avoid frequent operating system (OS) action yet unnecessary copying can still result in low performance. We explore improving application messaging performance by eliminating all unnecessary copies (minimal messaging). For minimal messaging, NIs must support address translation and must do so more richly than has been done in the past. NI address translation should flexibly support higher-level abstractions, map all user space, exploit translation locality, and degrade gracefully when locality is poor. We classify NI address translation implementations based on where the lookup and the miss handling are performed (CPU or NI). We present alternative designs and we consider how they interact with the OS.

We provide simulation results that evaluate the alternative design points and we demonstrate feasibility with a real implementation using Myrinet. We find: (a) NIs need not have hardware lookup structures, as software schemes are fast enough; (b) it is difficult for an NI to handle its own translation misses unless commercial operating systems are substantially modified to view an NI as CPU peer; (c) in the conventional situation where the operating system views the NI as a device, minimal messaging should be used only when the translation is present, while a single-copy protocol is used when it is not and (d) alternatively, one can currently get acceptable performance when the CPU handle misses if the kernel provides very fast trap interfaces but microprocessor and operating system trends may make this alternative less viable in the long run.

1 Introduction

Modern networking applications place a demand for high throughput and low latency on the network subsystem. Data intensive applications like multimedia depend on high throughput to stream large amounts of data through the network. Client-server and parallel computing applications depend on low latency for fast response times. Network performance will become even more important as system area networks [3] are used in clustered servers. While the network hardware has been able to achieve high throughput and low latency, it has not proven easy to deliver this performance to the application. A key obstacle to reaching the hardware limits has been overheads associated with message processing and message delivery within the host, especially when the operating system (OS) must be involved in every message transfer.

For this reason, several research efforts have sought to provide protected user-level access to the network interface (NI) so that the OS need not be invoked in the common case *(user-level messaging).* Typically, the OS maps the device registers and/or device memory in the user address space. Thereafter, the application can initiate message operations communicating directly with the device using loads and stores to send and receive messages. Examples of such designs include the Arizona Application Device Channels (ADCs) [14], Cornell U-Net [54], HP Hamlyn [56], Princeton SHRIMP [4]. The result of this research has been commercial designs like Myricom Myrinet [6], Fore 200 [9], Mitsubishi DART [36], DEC Memory Channel [18].

When the OS is removed from the critical path, the memory subsystem emerges as a major hurdle for delivering the network performance to the application. In the last ten years, memory speeds and memory bus bandwidth have failed to keep up with networks and the trend is likely to accelerate in the future [21,38,33]. This disparity has led some to argue that we are on the verge of a major paradigm shift that will transform the entire structure of information technology [17]. Even today, studies have shown that network protocols spend a significant amount of time simply copying data [49]. Therefore, many designs have attempted to avoid redundant copying at the application interface [13,42,56], the OS [25], and the network interface [36,11,26,1].

To push the envelope of possibilities, we ask whether it is possible to efficiently implement messaging with no extra copying where message data are only copied out of sender's data structures into the sender's NI and from the receiver's NI to the receiver's data structures (the data should travel through a node's memory bus only once). We refer to this operation as "*minimal messaging*" instead of the commonly used term *"zero-copy protocols*" which has been used inconsistently in literature. Minimal messaging reduces the critical paths and decouples the CPU and the NI allowing the overlap of activities within the node. Moreover, it avoids second order effects like cache pollution due to messaging. Bringing data into the CPU cache makes even less sense when the data originate or are destined for a device in the node (e.g., framebuffer, disk).

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, National Science Foundation with grants MIP-9225097, MIPS-9625558, and CDA-9623632, a Wisconsin Romnes Fellowship, and donations from Sun Microsystems. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright nota tion thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

Minimal messaging also reduces the resource demands on the NI since data quickly move to their final destination. Thus, in many cases, minimal messaging leads to faster messaging.

For minimal messaging, the NI must determine the location of the application data structures in main memory. The application accesses data using virtual addresses, which can be passed to the NI when the message operation is initiated. However, the NI is a device and therefore, it accesses memory using physical addresses. Consequently, the application virtual addresses must be translated to physical addresses usable by the NI. Therefore, the key requirement for minimal messaging is an address translation mechanism. Such mechanisms are the focus in this paper.

Surprisingly, minimal messaging has proven to be an elusive target. For example, in some designs that support minimal messaging there are restrictions on the size or location of the message buffers that can be directly accessed by the NI [5,18,8,22]. While the application can incorporate such message buffers within its data structures, in practice the complexity and overhead of managing them can be significant and they may end up being used as intermediate buffers. We can trace the cause of such limitations to the NI address translation mechanisms and their properties. In many cases, the abstraction that introduces the NI address translation is not appropriate to support minimal messaging. Therefore, we argue that the NI must incorporate an address translation mechanism that provides a flexible interface and is able to cover all the user address space. In other cases, the performance characteristics of the address translation mechanism do not favor minimal messaging. Therefore, we argue that the NI address translation mechanism must take advantage of the locality in the source and destination data addresses to reduce the translation overhead and must degrade gracefully when hardware or software limits are exceeded.

The address translation structures in NIs can be characterized by where the lookup is performed and where the misses are handled. For these questions, the answer can be either the NI or the CPU. Alternative design points differ in their requirements for OS support. Designs in which the NI handles misses, require extensive support by the OS (e.g., custom page tables). In contrast, handling misses in the CPU can be supported through standard kernel interfaces that wire pages in memory. Unfortunately, these interfaces have not been optimized for speed and therefore, such designs fail to degrade gracefully. Consequently, we discuss techniques to address this problem and offer graceful degradation in the absence of appropriate OS interfaces.

In this paper, we make the following four contributions:

- We present a classification of NI address translation mechanisms based on where the lookup and the miss handling are performed. We discuss alternative designs including a novel one that allows user software to load mappings via a device driver and therefore, it enables custom user-level control without sacrificing protection.
- We consider the OS support that alternative designs require and we propose techniques to provide graceful degradation in the absence of appropriate OS interfaces including a novel one that gracefully degrades to single-copy when a mapping is not available and therefore, it makes fast miss handling less important.
- We provide performance data from simulations which demonstrate that the proposed techniques allow the designs to gracefully degrade. Moreover, the simulation results show that for the performance of the address translation structures more attention must be given to how misses are handled than to how the lookup is performed. For the lookup, software based schemes can give acceptable performance. For the miss handling, low overhead mechanisms are required or else extra care must be taken to avoid thrashing the translation structures.
- We demonstrate the feasibility of the approach by presenting experimental results from an implementation on real hardware (Myrinet) where minimal messaging reduces latency for 2048-byte messages by up to 40%.

The remainder of the paper is organized as follows. Section 2 elaborates on minimal messaging and the assumptions of this study. Section 3 identifies the address translation properties for minimal messaging. Section 4 analyzes the design space for address translation mechanisms and presents representative designs that span the entire design space. Section 5 evaluates these designs in a simulation environment. Section 6 makes the feasibility case with an implementation on real hardware (Myrinet). Section 7 presents related research and commercial efforts and discusses their address translation mechanisms. Section 8 finishes the paper with the conclusions of this work.

2 Minimal Messaging

In this section, we elaborate on minimal messaging and list the requirements that the NI must satisfy to support minimal messaging. Furthermore, we discuss the assumptions that are implicit in this study and sketch the simplified system model we will be using throughout the paper.

For minimal messaging, the NI must examine the message contents, determine the data location and perform the transfer. The first step requires processing capabilities in the NI or in the I/O subsystem [48]. The second step requires a translation mechanism from virtual addresses to physical addresses. The application knows the data location by its virtual address. It can be passed to the NI when the message operation is initiated. However, the NI is a device (commonly) attached to the I/O bus. It can access memory using physical addresses. This dictates that the virtual address known by the application must be translated to a physical address usable by the NI in a protected manner. The third step requires the NI to be able to access data in main memory without the need for kernel involvement to flush the CPU cache before or after the transfer *(processor coherent direct memory access)*. Modern I/O architectures [19,30] support this feature.

Since physical addresses must be used to read data from the sender's memory and write data into the receiver's memory, the physical address must be available to the NI before the data transfer takes place. We assume that this is done with two address translations, the first at the sender and the second at the receiver using user's virtual addresses. Alternatively, the sender could perform the receiver's address translation and send messages with destination physical addresses [5]. We do not consider this case further because non-local knowledge of physical addresses makes paging, fault isolation, and security containment much more difficult.

In order to simplify the presentation, this study assumes that NI address translation mechanisms operate on virtual addresses as known by the application. In a general purpose OS, this is not sufficient when many processes in a node concurrently wish to use the network. Therefore, the mechanisms must be extended for multiple senders and receivers in one node. A straightforward way is to define message segments as areas where messages can be send or delivered. Applications can create such segments to export a region of their address space. Thereafter, the equivalent of the virtual address is the pair of <segment id, segment offset>. Subtracting the segment base from an application virtual address is sufficient to calculate the segment offset for any virtual address within the message segment. A protection mechanism can grant access for a particular process to send or receive messages to a specific message segment. Thereafter, the NI must enforce the access rights when it sends or receives messages. Equivalent models are described in Berkeley's Active Message specification [27] and Intel's Virtual Interface Architecture [15].

3 Address Translation Properties

In this section, we present the properties that the address translation mechanism should satisfy and we argue why we consider them desirable for minimal messaging. First, NI address translation mechanisms must be incorporated within an abstraction that can support minimal messaging. The first two properties, application interface requirements we have often seen violated in existing designs, belong in this category. Second, the performance characteristics of the address translation mechanisms must favor minimal messaging. The next two properties, performance requirements that allow minimal messaging to provide better performance than single-copy messaging, belong in this category.

(A) Provide a consistent, flexible interface to higher level abstractions. The address translation mechanism should allow the cut-through semantics of minimal messaging to be exposed to the application through the abstraction layers. In most cases, applications access the network through a layer of messaging abstractions. We can distinguish between low-level network access models and high-level user messaging models. Network access models such as ADCs [14], U-Net [54], Active Messages [55], Fast Messages [37], provide protected user access to the NI and serve as a consistent low-level model across NIs. Applications can use them to access the network but likely they will prefer higher level messaging models such as Fbufs [13], MPI [16] or TCP/IP.

Minimal messaging, by definition, provides a path for the message data through the abstraction layers to the application data structures. At the lowest level, incoming or outgoing messages should point to application data structures. Until message operations complete, these structures are shared between the application and the NI (*shared semantics*). In abstraction layers that offer *copy semantics*, in which outgoing or incoming messages contain copies of the application data, it is difficult to fully support minimal messaging. In general, a change in semantics introduces extra overheads [7]. Therefore, it is important that the lowest abstraction layer in the NI architecture to offer appropriate semantics or else implementations of all abstraction layers will suffer from the mismatch in semantics.

As a negative example of inflexible low-level interfaces, consider ADCs, which have been designed to optimize stream traffic. On the sender, the application enqueues the data addresses for outgoing messages. On the receiver, the data end up in incoming buffers allocated out of a queue of free buffers in the application address space. An address translation mechanism can be used to map the application addresses to physical addresses [12] but it is not sufficient to fully support minimal messaging because the abstraction does allow the NI to move incoming data directly to user data structures. Except for limited cases where the application knows the data destination for the next incoming message before the message arrives, an extra copy on the receiver is necessary.

In the designs we present in Section 4 we avoid such limitations in the application interface. The only requirement in the application interface is that the remote virtual address should be specified when the message is injected into the network. If this address is not known, the messaging library resorts to a single-copy approach. Alternatively, we could have used more aggressive application interfaces that allow the remote virtual address to become known just before the data are moved to the receiver's memory [35,42].

(B) Cover all of the user address space. If the address translation mechanism is limited in its reach or it is expensive to change which pages it maps, the available space may end up being used as intermediate buffers in a single-copy approach. While it is possible for the application to incorporate these message buffers within the application data structures, in practice the complexity and overhead in managing them can be significant, depending on the application characteristics. For example, if the application wants to use the mechanism for a memory region of size greater than the mechanism's reach, it has to make explicit calls to change the installed mappings as it sends or receives different portions of the region. This limits the portability of the applications since it exposes them to an implementation constraint. More importantly, it becomes impractical to use the mechanism for applications without a prior knowledge of the messaging pattern or without a message exchange to agree on the translations before the data transfer. Finally, it becomes difficult for higher level messaging models to expose such constraints to the application cleanly without violating the first requirement.

The reason for the limited reach in NI address translation mechanisms is often the absence of *dynamic miss handling*. If a translation is not available, no mechanisms exist to install the translation (e.g., loading the translation from device page tables or notifying the OS to do it). Designs that lack this feature are limited by the size of the NI translation structures. Our designs (Section 4) avoid such limitations because the translation structures are treated as caches that dynamically respond to the application requirements.

(C) Take advantage of locality. In any translation scheme we want to take advantage of potential locality by keeping recent mappings handy for future uses. Applications exhibit temporal and spatial locality at various levels, which include their network behavior [32]. This locality will be reflected in the application data addresses from which the applications send or receive messages. The address translation mechanism should take advantage of this behavior when it exists.

(D) Degrade gracefully when system limits are exceeded. Locality by itself should not be the only mechanism which ensures good performance. First, the NI translation structures cannot fully describe the address space. In the unlikely case, they are large enough to contain as much information as the kernel structures (i.e., page tables), performance considerations make it too expensive to maintain a copy of the kernel structures on the NI. Therefore, we shall have to deal with misses when we exceed the capacity of the NI translation structures. Second, we should not degrade the performance of transfers that do not exhibit locality, such as one-time bulk data transfers. Thus, we should strive for a translation mechanism that allows minimal messaging to be at least as good as the single-copy approach when its limits are exceeded. As we shall see in Section 5, minimal messaging can easily result in worse performance than single-copy messaging, if the design requirement for graceful degradation is overlooked.

4 Address Translation Implementation Alternatives

In this section, we first present a classification of the NI address translation mechanisms according to where the lookup and the miss handling are performed. Then, we examine the points in this design space and discuss the requirements of alternative designs for OS support. Our discussion points to the central role of the OS in providing the interfaces required to achieve graceful degradation. Accordingly, we finish this section discussing three techniques to make this property hold in the absence of appropriate OS interfaces.

Lookup		Miss Service: NI	Miss Service: CPU
NI	Hardware Structures	Network Coprocessors (Section 4.1)	Custom Finite State Machines (Section 4.2)
NI	Software Structures	Network Microcontrollers (Section 4.1)	Software TLBs (Section 4.2)
CPU		-	User-controlled mappings (Section 4.3)

TABLE 1. Classification of Address Translation Mechanisms

NI address translation structures can be viewed as caches that provide physical addresses for data sources and destinations. Two key questions in a cache design are how to do the lookup and how to service misses. There are two places, the NI and the CPU, where the lookup can be done. When the lookup is performed on the CPU, misses will be handled there. When the lookup is performed on the NI, there are two choices on where to handle misses. The first is for the NI to directly access device page tables (prepared for the NI) in main memory and handle its own misses. The second is for the NI to interrupt the CPU and ask it to service the miss.

Designs that perform the lookup and the miss handling in the NI correspond to network coprocessors or network microcontrollers [1,24]. Designs that perform the lookup in the NI and the miss handling in the CPU, correspond to software TLBs or custom hardware finite state machines [20,36]. This classification reveals another interesting design point in which both the lookup and the miss handling are performed on the CPU through an interface that allows user-level software to control the mappings that are installed in the NI translation structures. Table 1 shows the design space and places representative designs within it.

4.1 NI Lookup -- NI Miss Service

We can build a flexible NI that handles its own misses. The address translation mechanisms in this design follow the philosophy of similar mechanisms designed for processors. Modern operating systems maintain three levels of processor translations. First, *translation lookaside buffers (TLBs)* make mappings available to the CPU(s). Second, processor page tables are maintained in main memory from which CPU(s) quickly load mappings for pages resident in main memory. Third, a complete description of a process address space, including pages that have been swapped out or not accessed yet, is maintained in internal kernel data structures [53]. The OS defines public kernel interfaces to access and modify mappings in all these levels. In addition, it maintains consistency among the different translation levels. For example, when a page is swapped out to secondary storage, any mappings for that page must be flushed throughout the system.

In this design (Figure 1) the OS views the NI as a processor. Address translation structures on the NI correspond to CPU TLBs. The NI searches for mappings when it sends or receives a message in order to access application data. If a translation is not available in the NI translation structures, the NI accesses device page tables that the kernel maintains in main memory. This should be an operation of the same order as a CPU TLB miss (~ few hundred cycles). If the page has not been accessed before on the node or it has swapped out, the kernel in the host CPU should be invoked to take care of the miss. In the general case, it is not realistic to expect that the NI is able to execute kernel code. This requires a network coprocessor of the same architecture as the host CPU with the ability to access kernel data structures efficiently and communicate with devices (e.g., disks), which may be impossible for a device on the I/O bus. We do not consider such misses further because both in single-copy and minimal messaging the limiting factor is how fast the kernel can allocate new pages or swap in old pages from secondary storage.

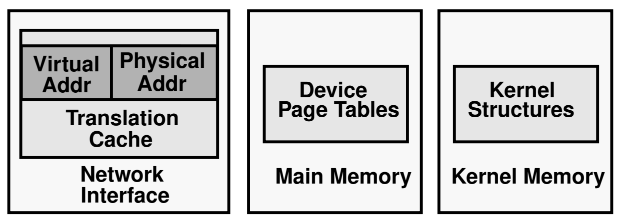

FIGURE 1. NI Lookup -- NI Miss Service

A translation cache is physically present on the NI. If a translation is not available in this cache, the NI looks up device page tables that the kernel maintains in main memory. For misses on pages that have not been accessed before on the node or that have been swapped out, kernel code must consult the OS structures that describe the application address space.

We can implement NI translation structures in software, similar to the software TLBs proposed for FLASH [20]. To implement software structures, we need an NI microcontroller that it is flexible enough to synchronize with the node's CPU to access its own page tables in main memory. Such structures have small associativity and many entries. The lookup overhead is directly proportional to the number of entries in the cache set that we must examine sequentially to find a match. Alternatively, we can consider hardware support for the lookup as in designs with a network coprocessors that include their own memory management unit and address translation hardware (TLBs) [1,26]. Such hardware structures should have high associativity with relative few entries (~ tens) and zero lookup overhead.

When a mapping is invalidated (paging activity, process termination) throughout the system, the host CPU must flush the entry out of the NI page tables, This is an operation that it is similar to TLB invalidations in multiprocessor systems. This is more complicated than in the case of a processor TLB because processor translation apply on memory accesses that are atomic with respect to other system events. NI translations, however, must be valid for the entire duration of the data transfer. This requires either that the active transfers aborted or the kernel is made to avoid invalidations for pages with active transfers.

The main disadvantage of this design is that it requires significant OS modifications since the OS must treat the NI as a processor. Commodity operating systems have not been designed to support page tables for arbitrary devices and they do not offer public kernel interfaces that provide this functionality. Even worse, this functionality cannot be implemented by standard device drivers using unsupported kernel interfaces because the virtual memory subsystem is at the heart of an OS and it cannot be easily modified without kernel rewriting. Therefore, including the appropriate support in commodity operating systems requires significant commitment from the OS groups for a specific platform. For example, in Solaris 2.4, the only device (other than processors) for which the kernel supports page tables in the Sun-4M architecture [30], is the standard SX graphics controller [31]. The code is deep inside the virtual memory subsystem and no public kernel interfaces exist to support this functionality for other devices or even other graphics controllers.

4.2 NI Lookup -- CPU Miss Service

To overcome the lack of OS interfaces that support device page tables, we can handle misses in the host CPU within the device driver's interrupt routine using standard kernel interfaces (Figure 2). The key characteristic of this design is that it supports only the NI translation structures and not special device tables in main memory. NIs with microcontrollers can implement the lookup with software TLBs [20]. Some designs however, include hardware support for the lookup [36] in the form of custom finite state machines for message processing.

Whenever a miss occurs, the device triggers an interrupt invoking the device driver's interrupt handler, which handles the misses. Using standard kernel interfaces, pages are wired down when their mappings enter the translation cache and are unwired whenever they leave it. The OS must know the reason for wiring the page down (i.e., incoming or outgoing message) to properly maintain the dirty and reference bits. Because of this, if a page is wired for an outgoing message, a second fault must be forwarded to the host when a translation is required for an incoming message.

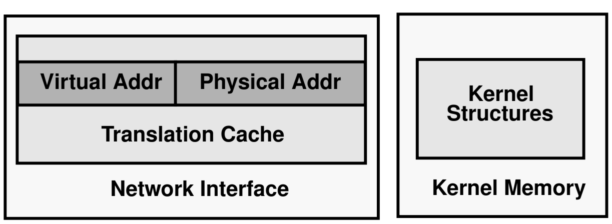

FIGURE 2. NI Lookup -- CPU Miss Service

A translation cache is present on the NI. The NI device driver uses the processor page tables (memory resident pages) or the kernel's address space structures to load mappings in the NI translation cache.

In this design, the OS treats NI translations as device translations. However, the way that the NI uses the translations does not match the way that the OS expects device translations to be used. Device translations are explicitly requested on an operation basis (e.g., when transferring data from disk to memory). The kernel defines an interface to wire pages in physical memory before the transfer and unwire it when it is completed. Since the OS sets up the translation, it is aware of incoming or outgoing transfers and it can maintain dirty and reference bits for these pages. In our case however, the NI keeps pages wired as long as they are installed in the translation cache, potentially for a long time. Since the OS assumptions about the duration that the pages remain wired do not match reality, we have an OS integration problem. Effectively, wired pages are no longer managed by the OS and this may result in underutilization of the physical memory. It is possible that these pages remain wired in memory even if the translation is not useful for communication anymore. If replacements due to conflicts are not enough to force old mappings out of the cache, we must periodically flush it.

In addition, the kernel interfaces involved for wiring and unwiring pages are not particularly fast. These calls must traverse layers of kernel software to perform the operation (~ few thousand cycles). If the translation structures cannot hold all of the translations used by the application, miss processing within the device driver is in the critical path. Therefore, the ability of the design to gracefully degrade is questionable once system limits have been exceeded and the translation structures start thrashing.

4.3 CPU Lookup -- CPU Miss Service

Traditional NIs often support minimal messaging with the kernel [25,7,51] supplying physical addresses to the NI. Involving the kernel in every message operation is unacceptable for user-level messaging. Therefore, we have devised a novel mechanism that allows us to do the lookup in user level on the host CPU (Figure 3). The key idea is to provide a protected interface through which user code can manipulate the contents of the NI translation structures. This enables custom user-level control without sacrificing protection.

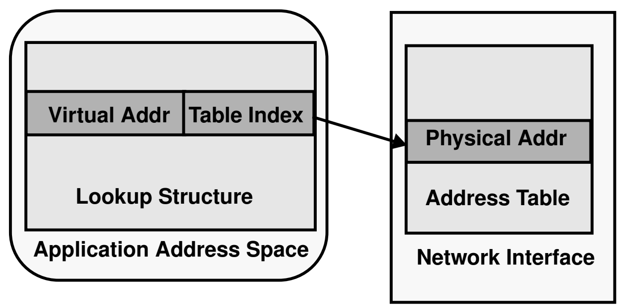

FIGURE 3. CPU Lookup -- CPU Miss Service

Data structures in the application address space are used for the lookup. The device driver installs physical addresses in the translation table per user request using the processor page tables (memory resident pages) or the kernel's address space structures.

In this design, there is a table on the NI that holds physical addresses. The user code cannot directly modify the contents of that table. However, it can request the device driver to install the physical address of a page in the user address space at a specific table index. The OS checks the validity of the user request, wires the page in memory, unwires the page previously installed there and stores the physical address in the requested table entry. Thereafter, the user code can pass to the NI the table index. The physical address stored in the table can be used as the data source of an outgoing message or the data destination of an incoming message.

For outgoing messages, the translation is done by the CPU before the message is processed by the NI. If the size of the table is greater than the maximum number of outstanding messages and we are using a *least-recently-used* (LRU) algorithm to choose replacement candidates, it is guaranteed that the translation will remain valid until the message is processed by the NI. For incoming messages, the translation is performed after the CPU examines the header of the incoming message. This results in a distinct disadvantage of this design over the design in Section 4.2. On the receiver, the CPU is still on the critical path since we cannot know the destination address for the incoming message until after the CPU examines the message. In addition, we have to pay the notification penalty for the message arrival.

This design has two advantages over the one in Section 4.2. First, the policies are set entirely at user level, so any user-level application knowledge can be readily incorporated. Second, it can support sophisticated algorithms to manage the translation cache since the host CPU typically is more powerful and flexible than any NI logic or NI processor. For example, in our user-level messaging library, we use this processing power to maintain a three-level page table in user software. On the bottom level, we store records to keep track of whether or not a translation for that page is installed in the table. Furthermore, the pages with active translations are also installed in a double-linked list that implements the LRU policy.

The design also shares the same problems with the one in Section 4.2. First, as we have discussed, the OS integration is problematic since standard kernel interfaces for wiring pages have not designed for the way we use them. Second, the design fails to degrade gracefully due to the overhead in standard kernel interfaces for wiring pages. If the physical address table cannot hold all of the translations used by the application, the device driver system call to install new mappings is in the critical path for all messaging operations once thrashing starts. The presence of an interface for user control of the replacement policy partially offsets this problem since it gives some user control to avoid thrashing. However, it is not desirable to expose this interface all the way to the application since it is implementation specific.

4.4 CPU Miss Service Optimizations

The discussion thus far favors designs where misses are handled by the NI. Since this depends on how the OS views the NI, lack of OS support can pose a significant obstacle. To overcome this problem, we can handle misses on the CPU. In this case, we can implement all the functionality in a device driver with standard kernel interfaces. However, as we have discussed, this approach fails to degrade gracefully once the translation structures start trashing. Therefore, we propose some techniques to avoid this worst case scenario.

Reduce the miss rate. We can reduce the miss rate in cache designs with better replacement policies. However, no policy can avoid thrashing when the application requirements exceed the size of the translation structures. Alternatively, we can increase the number of translation entries, either by increasing the amount of memory in the NI board or by using a second-level translation cache in main memory. In designs with second-level translation caches, unlike the designs of Section 4.1, the OS still treats the NIs as devices. Therefore, pages with translations in the NI translation structures must be wired using standard kernel interfaces.

Increasing the number of entries in the translation structures makes the problem of poor memory utilization more severe. Because there are many entries, flushing the entire cache may become too expensive to do indiscriminately. Therefore, we need to replicate the kernel's paging algorithms for the wired pages in the NI and its device driver. In particular, we must maintain dirty and reference bits and periodically sweep unused pages by flushing their translations from the NI translation structures.

While larger translation structures increase the reach of the NI address translation mechanism, the fundamental problem of graceful degradation has not been addressed. If the size of the application working set is larger than the size of the translation structures thrashing still occurs. Therefore, we do not consider it further.

Reduce the miss penalty. Despite the fact that kernel interfaces for wiring pages are not very fast, we argue that the operation is not logically complicated. The kernel must translate a virtual address to a physical address and keep track that a translation is being used by the NI. We can therefore, devise a specialized, fast kernel interface as follows. In the common case, we can use the CPU translation hardware by temporarily switching contexts to the process that the virtual address belongs to and then probe the memory management unit (MMU) to get the translation from the hardware page tables. Then we can update the NI translation cache and set a bit in a kernel structure to ensure that the page is wired. The paging algorithm must be modified to take these bits into account. These operations can be performed in the kernel trap handler fast (~few thousand cycles). Using similar techniques, we have been able to reduce the roundtrip time for synchronous traps on 66 MHz HyperSparcs, from 101 µsecs with the standard Solaris 2.4 signal interface to 5 µsecs with optimized kernel interfaces [45]. Other researchers have reported similar results [40,50].

However, specialized fast interfaces are not viable in the long run. First, we violate kernel structuring principles. Device specific support must be implemented at the lowest kernel levels where no public interfaces exist to support this functionality. The exact details of the handler depend on the processor and system architecture as well as the OS version. A different handler is most likely required for every combination of device, OS version, processor implementation, and system architecture (if the OS natively supports such an interface, its development becomes more manageable). Second, it will become more difficult to maintain good performance with this technique in the long run. As CPUs get more complicated (e.g., speculative execution), larger amounts of CPU state need to be flushed on a trap, thereby increasing the trap overhead. Nevertheless, the method is a good way to get acceptable performance in prototype implementations.

Tolerate the miss penalty. For graceful degradation, it is sufficient to be able to fall back in the performance of the single-copy approach when the translation cache exhibits thrashing. This can be achieved by defining the single-copy as a fast default fallback path for the message to follow while handling translation misses is postponed. This strategy is most effective when the miss detection and the decision to use the fallback path occur at the same place, that is with host lookup for outgoing messages and with NI lookup for incoming messages.

With this technique, the NI may or may not move the data to the final destination. If it does not, some code running within the messaging library emulates the correct behavior by copying the data to the destination. This design moves the miss service off the critical path. Minimal messaging becomes an optimization that the system uses when possible. We can further enhance the mechanism's flexibility by allowing the user code to initiate miss service through an explicit request at appropriate times.

By itself, the existence of the single-copy fallback path is not enough to track the performance of the single-copy mechanism. We postpone servicing miss requests, but we still have to do them. However, the existence of the single-copy fallback path allows us to ignore those requests. Therefore, we can achieve graceful degradation if we control the rate of miss processing. This can be done using internal knowledge (e.g., avoid it when we know we will exceed the capacity of the translation structures), directly exposing it to the higher layers, or providing worst case bounds. We have implemented a worst case bound as follows: miss processing should not degrade the performance of the worst case by more than a predefined amount over the single-copy method. A simple hysteresis mechanism that counts the message bytes transferred and only services misses once every *n* bytes, approximates this constraint. Effectively, we sample the address stream once for every *n* data bytes transferred. In other words, we trade the ability to take advantage of locality present in highly variable streams for worst case bounds.

5 Simulation Results

Two methods of studying address translation in NIs are simulation and hardware measurements. Hardware measurements are very accurate for the system under study, but are difficult to extrapolate to other systems and can be limited by constraints of the particular environment. Since we want to study a basic mechanism across design points, simulations, being more flexible, are better suited to this task. However, in Section 6, we present results from an implementation on real hardware to demonstrate the feasibility of our approach.

Our goal is to demonstrate that for the designs discussed in Section 4, the desired performance properties hold. The designs are able to take advantage of locality and to degrade gracefully when the capacity of the translation structures is exceeded. However, we do not attempt to determine whether NI address translation mechanisms can capture the locality present in typical workloads nor do we attempt to determine by how much the performance of those workloads can be improved with each design. Such study, interesting by itself, is highly dependent on the application domain and its programming conventions and beyond the scope of this paper.

The simulation results were obtained using a detailed execution-driven discrete-event simulator WWT-II [41,34]. The simulator can execute actual Sparc binaries by rewriting them to insert code that keeps track of the execution cycles. Each node contains a 300 MHz dual-issue SPARC processor, modeled after the ROSS HyperSPARC [43]. Each processor contains an 1 MB direct-mapped processor cache with 32-byte blocks. The processor sits on a 100 MHz, 64-bit wide MBUS-like [23] memory bus. The memory bus offers sustained bandwidth of 320 MB/sec for 32 byte transfers (e.g., cache fills and DMA transfers), 200 MB/sec for uncached stores to the NI and 120 MB/sec for uncached loads to the NI.

Each node includes an NI similar to the CM-5 NI [52] but augmented with processor-coherent DMA and an address translation mechanism. The CPU constructs messages by writing with uncached stores to the output queue of the NI. It receives the messages with uncached loads from the input queue. For DMA operations, the packet format has been extended to include transfer descriptors. The NI examines the headers of incoming or outgoing messages. When the appropriate field is set on the header, it extracts the local or remote virtual address and the data length from the transfer descriptors and performs the DMA transfers.

The network characteristics were chosen so that any performance degradation is strictly due to data transfers within the node. Therefore, the network latency is fixed to only 100 CPU cycles and the network bandwidth is only limited by an eight-message sliding window protocol. Moreover, similar to the CM-5, the NI resides on the memory bus. As we verified with simulations, moving it to the I/O bus makes DMA transfers even faster than CPU transfers with uncached memory accesses. Therefore, transfers in minimal messaging become even faster than transfers in single-copy messaging. Minimal messaging was implemented within the Tempest messaging model [39], a variant of Berkeley Active Messages [55].

Best Case Throughput & Latency. We want to determine the effectiveness of minimal messaging in the best case. Therefore, we shall measure the maximum possible benefit across message sizes. We are interested in two metrics, throughput and latency. To measure latency (round-trip time), we pingpong a message between two nodes. To measure throughput, we blast messages from one node to another. Each message sends the same data buffer from the sender to the receiver. Consequently, we never miss on the translation structures after the first message.

Figure 4 presents the results from this experiment for five designs. The lookup overhead is zero cycles in all cases (including accesses to the intermediate buffer). From the results, we can observe that:

- For small data blocks, minimal messaging offers little, if any advantage. The gain for avoiding the extra copy is small and it is balanced by the extra overhead of writing the transfer descriptors and performing processor-coherent DMA operations (if the message data are not aligned to 32 bytes processor-coherent DMA requires read-modify-write bus transactions). It should be noted that this result is limited to user-messaging models in which like active messages, a handler is invoked for every incoming message. In simpler user-messaging models as for example those which support remote memory accesses, minimal messaging can enhance performance simply because the CPU is not in the critical path on the receiver and therefore, communication and computation can be overlapped.
- The limiting factor is the memory bus occupancy and not the bandwidth of DMA operations vs. uncached accesses (i.e., we have not tilted the experimental setup to favor block memory bus transfers). Indeed, using an intermediate buffer results in worse performance compared to directly accessing the NI with uncached memory operations since the data move through the memory bus three times.
- Minimal messaging offers enhanced performance because of the overhead in moving data through the CPU. When minimal messaging is used on both the sender and the receiver, it buys more than what we gain by adding the benefits of doing it only on one side. The reason is that processor-coherent DMA transfers invalidate the data in the processor cache. When the CPU accesses them again for the next message, it incurs cache misses.

Lookup & Miss Behavior. We want to evaluate the performance of the designs discussed in Section 4, as the stress on the translation structures changes. For this purpose, we measure the throughput and latency as before, but instead of one buffer, we transfer *n* buffers. Each buffer resides in a different page. For small *n's,* (i.e., number of buffers), the performance is going to be determined by the lookup method. Therefore, we will be measuring the best case scenario. For large *n's*, the translation structures will be stretched and start

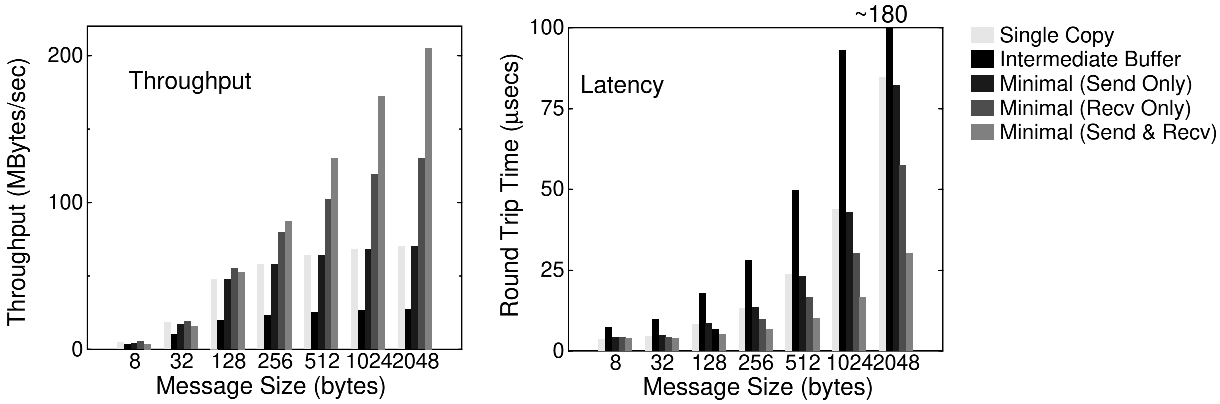

FIGURE 4. Simulated best-case throughput and latency

***Single Copy** uses uncached accesses to the NI. **Intermediate Buffer** uses an intermediate buffer which the NI can access with DMA operations and the CPU with cachable accesses. For the next three designs, we use **minimal messaging** on the sender only, on the receiver only or on both the sender and the receiver.*

thrashing as they encompass a larger working set. Therefore, we will be measuring the worst case scenario.

Figure 5 presents the throughput and latency results for two message sizes: 512 and 2048 bytes. Our results demonstrate that while all the designs can take advantage of locality, only three designs gracefully degrade. Serving misses on the NI, offers enhanced performance even with a miss in every message operation. Using optimized kernel interfaces also offers good performance but starting from larger messages. Finally, using the single-copy fallback path, with a threshold to control miss processing, degrades to within 10% of the single-copy approach.

We determine the impact of the lookup when we always hit in the translation structures. As expected, the graphs show that the lookup method does not affect the messaging performance significantly. Using hardware structures for the lookup is slightly better than using software ones but the difference is small. Nevertheless, software structures can easily contain thousands of entries compared to a few tenths included in a TLB. As a result, there is no incentive for hardware structures (e.g., a network coprocessor with a powerful MMU). CPU lookup also performs well in this experiment. Yet, its performance slightly degrades as the number of translation entries increases. For realistic translation structure sizes, the CPU would experience more cache misses in accesses to the translation structures than in these experiments.

We determine the impact of miss handling when the translation structures thrash. Our results indicate:

- For both message sizes, handling the misses on the NI is fast enough so that minimal messaging remains more efficient than single-copy.
- Using fast kernel interfaces also reduces the miss overhead. For 2048-byte messages, minimal messaging remains more efficient than single-copy. However, the overall performance when we exceed the limits of the translation structures with 512-byte messages is worse than the single-copy approach.
- The single-copy fallback technique when the miss service rate is controlled (one miss is serviced for every 64KB of data transferred), tracks the performance of the single-copy approach. Hence, the design degrades gracefully. Without a controlled rate, the performance plummets and the single-copy fallback fails to offer any benefits by itself. In fact, it makes things worse since we have to service the miss and use the slower single-copy path.
- The performance of the single-copy approach is also sensitive to the number of buffers. It drops as the size of the buffer exceeds the capacity of the CPU cache which again demonstrates the cost of moving data through the CPU.

6 Myrinet Results

To demonstrate the feasibility of our approach, we present results from an implementation on real hardware (Myrinet). This implementation started from the same NI control program distributed by Berkeley. Subsequently, it was modified to support the messaging subsystem of Blizzard [46,44,45], our fine-grain distributed shared memory system. To this date, it has been kept source level compatible with the Berkeley Active Message library. Less than 200 lines of C code in the NI control program and device driver were required to support address translation.

The testbed consists of Sun Sparcstation 20's connected through a Myrinet network. Each workstation contains two 66 MHz dual issue HyperSPARCs with 256 KB direct mapped processor cache. The processors sit on 50 MHz 64-bit wide memory bus (MBUS [23]). The NI is located on the I/O bus (25 MHz 32-bit SBUS [30]) which supports processor coherent I/O transfers between the memory and I/O devices. The NI is controlled by a 5 MIPS 16-bit microcontroller. The bus bridge supports an I/O memory management unit (IOMMU) which provides translation from SBUS addresses to physical addresses. The sustained bandwidth for 32-byte transfers on the memory bus is 160 MB/sec. The transfer bandwidth to the NI is limited by the I/O bridge to less than 30 MB/sec. The NI-to-NI network bandwidth is 40 MB/sec while the NI-to-NI network latency is 1.6 µsecs for every 128 bytes.

For user-level messaging, the device driver maps the NI's local memory, which is accessed with uncached memory operations, in the user address space. The NI control program implements separate send and receive queues on the NI memory. Each queue has 256 entries pointing to the message data. Under the standard I/O architecture, the NI can use DMA only for kernel addresses. Thus, to permit DMA to the user address space, the driver maps a kernel I/O buffer in the user address space, which is used for the message data. For mini-

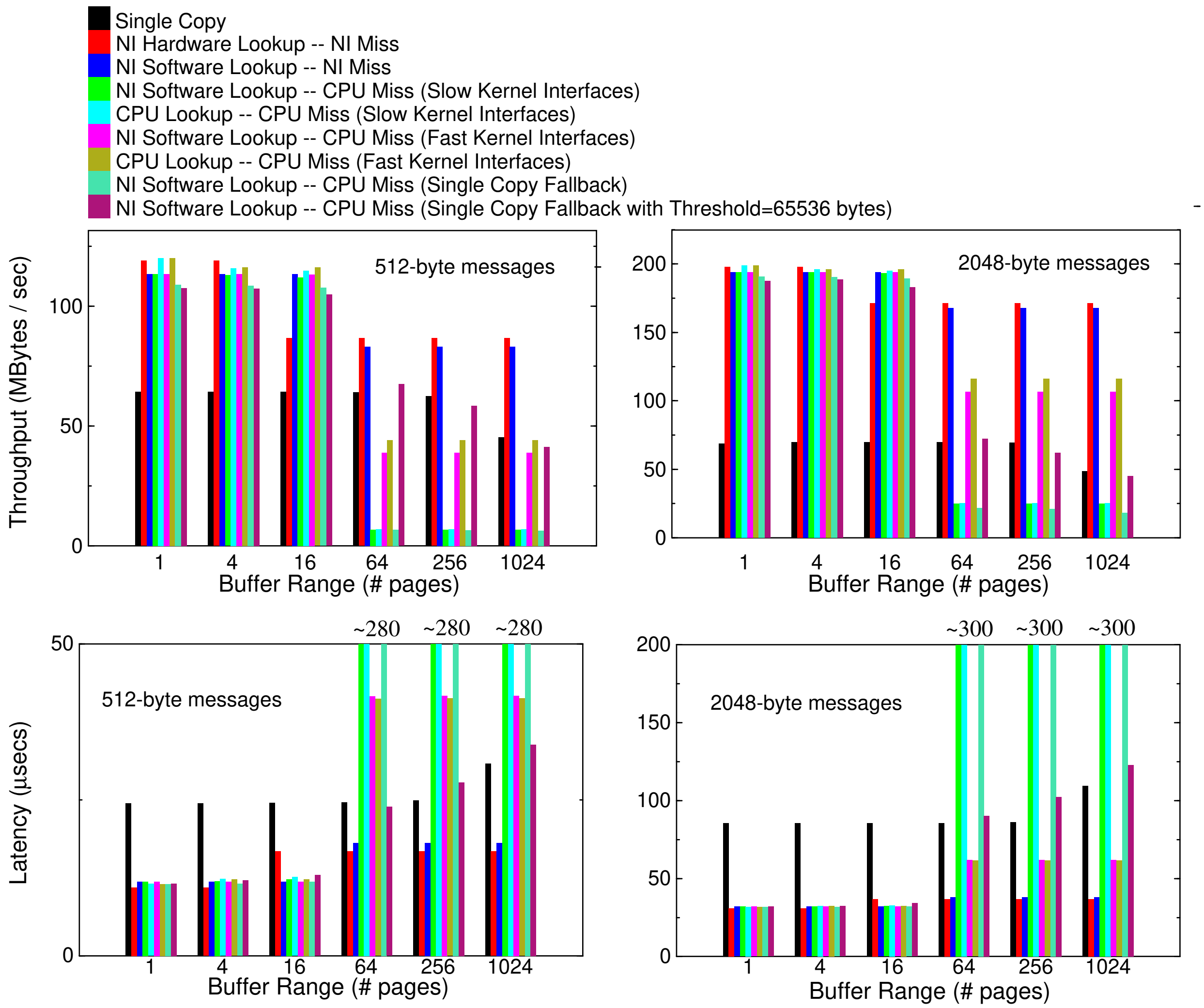

FIGURE 5. Simulated throughput and latency vs. buffer range.

We present results for 512-byte and 2048-byte messages. ***Single Copy*** *uses uncached accesses to the NI. All the rest have address translation structures with 32 entries (this is too small for software structures but it makes it easier to present the results). The replacement policy is random except in the host lookup method where we use an LRU algorithm. We evaluate* ***three lookup methods****:* ***(a) NI Hardware Lookup*** *uses a fully associative hardware structure on the NI with zero lookup overhead;* ***(b) NI Software Lookup*** *uses a two-way associative software structure on the NI with lookup overhead of 45 cycles per set;* ***(c) CPU Lookup*** *performs the lookup on the host using a user-maintained three level page table structure (180-500 cycles overhead). We evaluate* ***five miss handling methods****:* ***(a) NI Miss*** *handles the miss on the NI with an overhead of 450 cycles (three times the overhead of a CPU TLB miss);* ***(b) CPU Miss (Slow Kernel Interfaces)*** *handles the miss on the host CPU, triggered from the NI by an interrupt or the application through a system calls and it is serviced by the device driver using standard kernel interfaces (200 cycles interrupt overhead + 20000 cycles for the operation);* ***(c) CPU Miss (Fast Kernel Interfaces)*** *handles the miss on the host CPU, triggered as in (b) but serviced at the lowest kernel level within the kernel trap handler (200 cycles interrupt overhead & 2000 cycles for the operation);* ***(d) CPU Miss (Single Copy Fallback)*** *uses uncached accesses when the lookup fails and deals with misses after the message operation;* ***(e) CPU Miss (Single Copy Fallback with Theshold=65366)*** *is like (d) but only handles one miss for every 64Kb of data transferred.*

mal messaging, we allow the queue entries to point to user virtual addresses (or table indices for host lookup). We then use the BYPASS mode of the SBUS-to-MBUS bridge to directly access the application address space. With this mode, intelligent peripherals can use physical addresses *bypassing* the IOMMU translations.

On the sender, we do the lookup on the CPU. On the receiver, we do the lookup on the NI. In both cases, we use the single-copy fallback technique. Due to the design of Myrinet, this is a requirement on the receiver. If an NI does not empty the network fast enough, the sender NI causes the receiver NI to reset itself. The translation structures contain 256 entries each. The lookup overhead is 2 µsecs and the miss overhead is 100 µsecs using standard kernel interfaces to wire pages and unwire pages.

Figure 6 presents the best-case results across message sizes and Figure 7 presents the stress results. Our results show that: (i) in the best case, minimal messaging reduces latency by 25% and 40% for 512-byte and 2048-byte messages, respectively. (ii) in the worst case, it increases latency by 9% and 2%

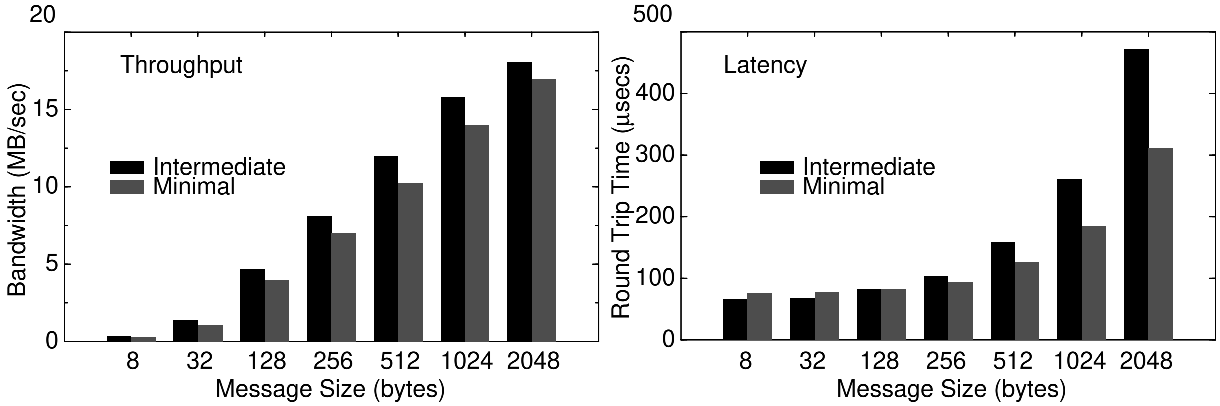

FIGURE 6. Myrinet best-case throughput and latency

Intermediate *uses the shared user/kernel buffer for the message data.* ***Minimal*** *accesses directly the application data structures.*

for 512-byte and 2048-byte messages, respectively, when we control the rate of miss processing. Overall, the latency results show similar trends as in the simulator.

Unlike the simulator, throughput does not increase. In these machines, there is limited bandwidth across the SBus/MBus bridge. In the intermediate buffer approach, the transfer across the bridge (which is the bottleneck) is overlapped with copying the data to the intermediate buffer for the next messages. With minimal messaging, the CPU is free to poll for incoming messages across the bridge. Polling consumes cycles on the I/O bridge and therefore, it slightly reduces throughput by 15% and 2% for 512-byte and 2048-byte messages respectively.

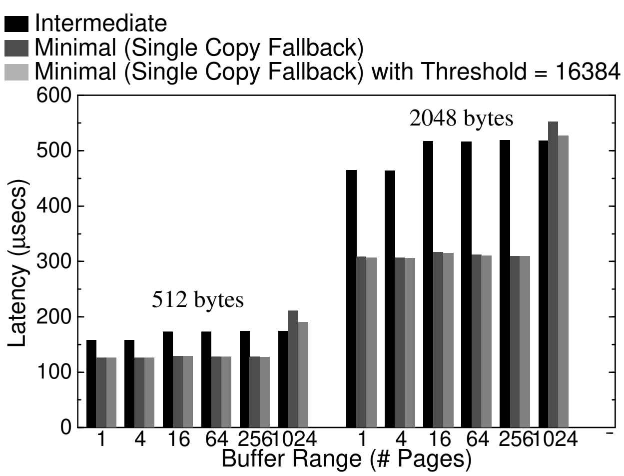

FIGURE 7. Myrinet latency vs. buffer range

Intermediate *uses the intermediate shared user/kernel buffer for the message data.* ***Minimal*** *accesses directly the application data structures.* ***Minimal with Threshold = 16384*** *services only one miss for every 16384 bytes transferred.*

7 Related Work

Abstractions that provide **sender-based communication** such as **HP Hamlyn** and **Berkeley Active Messages,** are powerful enough to support minimal messaging. The sender specifies the source and destination addresses as offsets in message segments for every message. In theory, you can define message segments that cover the entire application address space. However, in current Hamlyn [8,56] and Active Messages [29,26,10] implementations the address translation structures are not there or are limited to their reach. For example, the latest prototype Hamlyn implementation [8] is built on hardware identical to ours (Myrinet), yet message buffers must be pinned in main memory. This limits the coverage of the mechanism to the amount of application data that can be wired in physical memory. Similarly, the Active Messages implementation on Myrinet uses a single-copy approach through an intermediate shared user/kernel buffer where the NI pulls or pushes message data.

Arizona ADCs [14] have been designed to optimize stream traffic. In Section 3, we discussed why this design cannot fully support minimal messaging. The base **Cornell UNet** [54] architecture supports an abstraction similar to ADCs, and therefore has the same limitations as ADCs. In the original UNet paper [54], a direct access UNet architecture is discussed that includes communication segments able to encompass all the user address space but the architecture is restricted to future NI designs. Recent work [2] attempted to incorporate address translation mechanisms for existing NIs. Nevertheless, the ADC abstraction has not changed and therefore, the designs are unable to move data to their final destination without extra copying.

Mitsubishi DART [36] is a commercially available NI that comes close to properly support minimal messaging. DART core has been designed to support ADCs including sophisticated address translation support. Moreover, it defines an interface to a separate coprocessor that process messages. Presumably, it will be used to support the hybrid deposit model [35], an abstraction similar to Active Messages, in which the data destination is a function of the receiver's and t sender's state. Unfortunately, the address translation is geared towards ADCs. As a result, the translation structures are not flexible enough to efficiently support minimal messaging. The host CPU is always interrupted to handle misses while the message is blocked until the miss can be resolved. Furthermore, there are not any provisions for a fallback action. Thus, the design requires fast kernel interfaces to gracefully degrade once the translation limits are exceeded.

Designs with a **network coprocessor**, like **Meiko CS** [1] and **Intel Paragon** [24] can support minimal messaging using the microprocessor's address translation hardware and a separate DMA engine. Nonetheless, address translation mechanisms implemented for CPUs (TLBs) are not always appropriate for NIs. There are two potential problems. First, the reach of a CPU TLB is very small, typically a few dozen of pages. Message operations can span over a wide range of addresses, which is much larger than what TLBs can cover. Moreover, the data transfers compete with other memory accesses (kernel instructions, kernel data, I/O addresses), effectively making the TLB miss the common case for any message operation. Second, data transfers from/to the user address space require the CPU to switch the hardware context to the appropriate process. This operation can have significant overhead depending on the coprocessor's architecture (e.g., number of CPU hardware contexts, TLB and/or the cache flushing). Alternatively, the coprocessor can access page tables in software making the coprocessor TLB useless for minimal messaging.

In these designs that support **remote memory accesses**, memory pages in the sender's address space are associated with memory pages in the receiver's address space. Memory accesses on the sender are captured by the NI and forwarded to the associated page on the receiver. Page associations are either direct (the sender knows the remote physical address) or indirect (through global network addresses). Examples of this approach include **Princeton SHRIMP** [4], **Forth Telegraphos II** [28], **DEC Memory Channel** [18] and **Tandem TNet** [22]. SHRIMP and Telegraphos II use direct page associations. The Memory Channel and TNet use indirect page associations. Common characteristic of these designs is their inability to handle misses in the translation structures. Therefore, the translations must in place before messaging operations, which requires communication pages to be locked in memory. Moreover, changing the reach of the translation mechanisms requires expensive system calls. SHRIMP's prototype NI can hold up to 32K of associations between pages. Thus, minimal messaging is supported for up to 128Mb of application data from every sender. Similarly, the Memory Channel supports up to ~50K pages. In TNet, remote memory operations are supported in a 32-bit window to a node's physical memory.

A class of designs supports minimal messaging by using the CPU in kernel mode to instruct the NI to move the data to the appropriate place. Such approaches include page remapping in the kernel (implemented in Solaris 2.6 TCP [25]), Washington's in-kernel emulation of the remote memory access model [51] and other VM manipulations [7]. In these systems, minimum messaging is achieved if the NI can directly access the main memory. However, the kernel is involved in every transfer and thus, user-level messaging is not supported.

Princeton User-level DMA (UDMA) [4] avoids OS intervention and supports minimal messaging when it is used both to send and receive messages. UDMA is used in SHRIMP to initiate DMA transfers. In this case, it supports minimal messaging on the sender but on the receiver, it suffers from the same problems that we have discussed for SHRIMP. Nevertheless, it can be used on both the sender and the receiver (without SHRIMP's support for remote memory accesses). Consequently, it supports minimal messaging in a way that shares common features with the design that allows user-controlled mappings (Section 4.3). In the common case, both avoid kernel intervention for data transfers and in both the CPU is in the critical path for message operations. Unlike our design, UDMA requires hardware support in NIs to capture transfer requests.

Cray T3E [47] combines remote memory accesses with an approach similar to UDMA. It supports minimal messaging through special NI registers. The CPU initiates transfers directly from remote memory to the NI registers on the local node. It subsequently initiates the transfer of the data from the NI registers to local memory. The CPU must be involved once for every 64 bytes transferred (the maximum message size supported). T3E includes extensive hardware support for address translation in the form of complete page tables that describe global communication segments. However, the page tables must always have valid translations, and therefore the communication pages are wired in memory.

8 Conclusions

We have argued that eliminating all redundant copies in user-level messaging (minimal messaging) requires address translation support in the NI. The address translation mechanism should have a flexible interface, be able to cover all the user address space, take advantage of locality, and degrade gracefully.

We have classified the address translations mechanisms according to where the lookup and the miss handling are performed. This classification defined a design space, which we systematically analyzed. Furthermore, it exposed a design point for which we proposed a novel interface for user-controlled mappings. We examined the required OS support for each design point and proposed techniques to guarantee that the designs gracefully degrade in the absence of appropriate OS interfaces.

Simulation results validated that we can take advantage of locality and degrade gracefully, even without appropriate OS interfaces. Moreover, experimental results from real hardware demonstrated the feasibility and potential of this approach.

The simulation results indicated that for the performance of the address translation structures, more attention should be given to how misses are handled than how the lookup is performed. To summarize:

- Hardware lookup structures in the NI are not required since simple software schemes are fast enough.
- We would prefer the NI to handle misses, for performance and clean integration with the OS.
- If this is not possible due to lack of OS support, minimal messaging should be considered an optimization to the single-copy path. It will help when possible but it should not thrash when the hardware limits are exceeded.
- Fast kernel interfaces are a short-term solution; even if we disregard the poor OS integration, as CPUs get more complicated, their performance cost is going to rise.

The practical meaning of this study for system designers is that the key principle in the design of address translation mechanisms should be flexibility to support all possible levels of OS integration during the lifetime of the design.

Acknowledgments

This work was done in the supportive environment provided by members of the Wisconsin Wind Tunnel Project (http://www.cs.wisc.edu/~wwt). We would like to thank Pei Cao, Babak Falsafi, Andy Glew, Rebecca Hoffman, Andreas Moshovos, Shubu Mukherjee, Leah Parks, and Madhu Talluri for their helpful comments in earlier versions of this paper. Steve Scott clarified Cray T3E's address translation mechanisms. Many thanks to Rich Martin, David Culler and the NOW group at University of California for releasing the AM

communication package for Myrinet, which provided the infrastructure for Blizzard's messaging subsystem.

References

[1] Eric Barton, James Cownie, and Moray McLaren. Message Passing on the Meiko CS-2. *Parallel Computing*, 20:497–507, 1994.

[2] Anindya Basu, Matt Welsh, and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Hot Interconnects '97*, 1997.

[3] Gordon Bell. 1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac? *Communications of the ACM*, 39(3):11–15, March 1996.

[4] Matthias A. Blumrich, Cesary Dubnicki, Edward W. Felten, and Kai Li. Protected User-level DMA for the SHRIMP Network Interface. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[5] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

[6] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[7] Jose Carlos Brustoloni and Peter Steenkiste. Effects of buffering semantics on I/O performance. In *Second USENIX Symposium on Operating Systems Design and Implementation*, October 1996. Seattle, WA.

[8] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Second USENIX Symposium on Operating Systems Design and Implementation*, October 1996. Seattle, WA.

[9] Eric Cooper, Onat Menziolcioglu, Robert Sansom, and Francois Bitz. Host Interface Design for ATM LANs. In *Proceedings of the 16th Conference on Local Computer Networks*, pages 14–17, October 1991.

[10] David Culler, Lok Tin Liu, Richard Martin, and Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, pages 35–43, February 1996.

[11] Chris Dalton, Greg Watson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. *IEEE Network*, pages 36–43, July 1993.

[12] Peter Druschel, Mark B. Abbot, Michael A. Pagels, and Larry L. Peterson. Network Subsystem Design. *IEEE Network*, 7(4):8–19, July 1993.

[13] Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP)*, pages 189–202, December 1993. Asheville, NC.

[14] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of the ACM SIGCOMM '94 Conference*, pages 2–13, September 1994. London, UK.

[15] Dave Dunning and Greg Regnier. The Virtual Interface Architecture. In *Hot Interconnects '97*, 1997.

[16] Message Passing Interface Forum. MPI: A Message-Passinh Interface Standard. Technical Report Draft, University of Tennessee, Knoxville, May 1994.

[17] George Gilder. The Bandwidth Tidal Wave. *Forbes ASAP*, December 1994. Available from http://www.forbes.com/asap/gilder/telecosm10a.htm.

[18] R. Gillett, M. Collins, and D. Pimm. Overview of Memory Channel Network for PCI. In *Proceedings of the 41th IEEE Computer Society International Conference (COMPCON'96)*, 1996.

[19] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.1*, 1995.

[20] John Heinlein, Kourosh Gharachorloo, Scott A. Dresser, and Anoop Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, San Jose, California, 1994.

[21] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[22] Robert W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, February 1995.

[23] Sun Microsystems Inc. *SPARC MBus Interface Specification*, April 1991.

[24] Intel Corporation. Paragon Technical Summary. Intel Supercomputer Systems Division, 1993.

[25] Hsiao keng Jerry Chu. Zero-Copy TCP in Solaris. In *Proceedings of the '96 USENIX Conference*, January 1996. San Diego, CA.

[26] Lok T. Liu and David E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Proceedings of Intel Supercomputer Users Group Conference*, 1995.

[27] Alan Mainwaring and David Culler. *Active Messages: Organization and Applications Programming Interface*, 1995.

[28] Evangelos P. Markatos and Manolis G.H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, 1996.

[29] Richard P. Martin. HPAM: An Active Message Layer for a Network of HP workstations. In *Hot Interconnects '94*, 1994.

[30] Sun MicroSystems. *Sun-4M System Architecture, Rev 2.0*, 1991.

[31] Sun MicroSystems. *Kodiak SX Memory Controller Specification*, 1994.

[32] Jeff Mogul. Network locality at the scale of processes. In *Proceedings of the ACM SIGCOMM '91 Conference*, September 1991. Zurich.

[33] Shubhendu S. Mukherjee and Mark D. Hill. A Survey of User-Level Network Interfaces for System Area Networks. Technical Report 1340, Computer Sciences Department, University of Wisconsin–Madison, February 1997.

[34] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID*, June 1997.

[35] Randy Osborne. A Hybrid Deposit Model for Low Overhead Communication in High Speed LANs. In *Fourth International Workshop on Protocols for High Speed Networks*, August 1994.

[36] Randy Osborne, Qin Zheng, John Howard, Ross Casley, and Doug Hahn. DART - A Low Overhead ATM Network Interface Chip. In *Hot Interconnects*, 1996.

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

[38] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Perspective*. Prentice Hall, 1997.

[39] Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[40] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[41] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[42] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High Performance Local Area Communication With Fast Sockets. In *Proceedings of the '96 USENIX Conference*, January 1996. San Diego, CA.

[43] ROSS Technology, Inc. *SPARC RISC User's Guide: hyperSPARC Edition*, September 1993.

[44] Ioannis Schoinas. *Fine-Grain Distributed Shared Memory on Clusters of Work-Stations*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, 1997.

[45] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lucas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[46] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[47] Steve L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 26–36, 1996.

[48] I2O SIG. *Intelligent I/O Architecture Specification Rev. 1.5*, March 1997.

[49] Peter Steenkiste. A Systematic Approach to Host Interface Design for High-Speed Networks. *IEEE Computer*, March 1994.

[50] Chandramohan A. Thekkath and Henry M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 110–119, San Jose, California, 1994.

[51] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control Transfer in Distributed Operation Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, California, 1994.

[52] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary, 1991.

[53] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice Hall, 1996.

[54] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, December 1995.

[55] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[56] John Wilkes. Hamlyn — an interface for sender-based communications. Technical Report HP-OSR-92-13, HP, November 1992.