

MINI: MINIMIZING NETWORK INTERFACE MEMORY REQUIREMENTS
WITH DYNAMIC HANDLE LOOKUP

by

Reza Azimi

A thesis submitted in conformity with the requirements
for the degree of Masters of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2002 by Reza Azimi

Abstract

miNI: Minimizing Network Interface Memory Requirements with Dynamic Handle
Lookup

Reza Azimi

Masters of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2002

Recent work in low-latency, high-bandwidth communication systems has resulted in building Network Interface Controllers (NIC) and communication abstractions that support direct access from the NIC to application virtual memory to avoid both data copies and operating system intervention. Such mechanisms require the ability to directly manipulate application buffers in host memory for protection and delivering data. Most modern NICs statically maintain address translation and protection information. However, this results both in high memory requirements for the NIC and limitations in the size of host memory.

In this thesis, we categorize the types of data structures for managing communication buffers used in modern NICs, and propose mechanisms to dynamically manage such data structures to alleviate the related limitations. We implement our approach in a modern user-level communication system. The contributions of this thesis are: (i) The integrated approach for dynamic handle lookup that deals with all major lookup data structures reduces NIC memory requirements significantly and eliminates restrictions on the amount of host memory. (ii) A mechanism for handling misses on the receive path by using the retransmission in the NIC. (iii) The detailed performance evaluation with micro-benchmarks and real applications.

Dedication

To my lovely wife Afsaneh,

and my beloved sisters, Marzieh, Farzaneh, Nasrin, Azadeh, Nahid, and Nooshin.

Acknowledgements

I would like to extend my gratitude to several people whom without their help and support, I would not have been able to accomplish this thesis.

First, my supervisor, Angelos Bilas, for providing his unlimited support in every aspect of the work, from great ideas and insights to patient and tireless discussions on the very detailed problems. I would like to give my special thanks to him for his excellent supervision.

I also would like to extend my special appreciation to Peter Jamieson for the very useful discussions and enjoyable co-operation we had together in our theses. His role was essential to achieve the goals of this work.

I am grateful to Raymond Fingas whose initial but effective effort on the problems addressed in this thesis was a jump start for me.

I am thankful to the members of ATHLOS project, Rosalia Christodouloupoulou, Michail Flouris, and Ahmed Abdelkhalek with whom I had useful discussions in the course of this thesis.

Finally I thankfully acknowledge the financial support of Natural Science and Engineering Research Council of Canada, Canada Foundation for Innovation, Ontario Innovation Trust, the Nortel Institute of Technology, Communications and Information Technology Ontario (CITO), and Nortel Networks.

Contents

1	Introduction	1
1.1	User-level Communication	4
1.1.1	General Architecture	5
1.1.2	Major Operations	6
1.1.3	Memory Pinning	8
1.2	VMMC System Overview	9
1.2.1	VMMC Operations	10
1.2.2	NCP Memory Map	11
1.2.3	Lookup Data Structures	12
1.3	This Thesis	14
1.3.1	Dynamic Handle Lookup	14
1.3.2	Summary of Results	14
1.3.3	Overview	16
2	Related Work	17
3	Design and Implementation	21
3.1	Overview	21
3.1.1	Network Interface Cache	21
3.1.2	Interrupt-based Miss Handling	22
3.1.3	The Role of Packet Retransmission	24

3.2	VMH Lookup	25
3.2.1	VMH Lookup Table	26
3.2.2	VMH Lookup Cache	27
3.3	CBH Lookup	30
3.3.1	CBH Lookup Table	30
3.3.2	CBH Lookup Cache	31
3.4	Imported CBH Table	32
3.5	PCH Table Memory Requirements	33
3.6	Implementation Issues	34
4	Results	36
4.1	VMH Lookup	38
4.1.1	Micro-benchmark tests	39
4.1.2	Real Applications	39
4.2	CBH Lookup	47
4.3	Summary	48
5	Conclusions and Future Work	49
5.1	Summary of the Results	50
5.2	Future Directions	50

List of Tables

1.1	The breakdown of NIC memory consumption in the base system and <i>miNI</i> .	15
4.1	Cluster node configuration.	37
4.2	Basic VMMC costs.	37
4.3	The problem size for SPLASH-2 benchmark applications.	40
4.4	The range of values for each of the VMH lookup cache parameters.	40
4.5	The execution time (in ms) of each application in the original VMMC system versus <i>miNI</i> for the best VMH and CBH cache configurations.	44
4.6	The miss delay time distribution for each application. Numbers represent percentages of total misses.	45
4.7	The miss processing time distribution for the applications. Numbers represent percentages of total number of misses.	46

List of Figures

1.1	The general architecture of user-level communication systems.	5
1.2	The block diagram of the Myrinet network interface controller.	9
1.3	Message exchange in VMMC. The arrow points to the direction of data flow, for a typical send operation.	13
3.1	Cache miss handling path.	23
3.2	The structure of the VMH lookup cache.	28
3.3	The structure of the CBH lookup cache.	32
4.1	The effect of VMH miss on ping-pong latency and bandwidth.	39
4.2	VMH lookup cache miss breakdown of the SPLASH-2 applications.	42
4.3	The speedup of the SPLASH-2 applications in different VMH lookup cache configurations.	43
4.4	The effect of CBH miss on ping-pong latency and bandwidth.	47

Chapter 1

Introduction

Recent work in improving the performance of interconnection networks in scalable servers and storage systems has resulted in new network interface controllers (NICs) that support user-level communication [5, 6, 12]. The main operations supported by these modern network interfaces are: (i) the ability for user programs to directly access the network in a protected manner for sending and receiving data and (ii) data transfer directly to- and from- program virtual memory without operating system intervention. These capabilities are now part of industry standards that are in use today or are being proposed for future interconnects [11, 16, 15]. To provide support for these mechanisms modern network interfaces require certain extensions.

First, they require efficient support for translating between virtual and physical memory addresses. Modern network interfaces usually perform DMA transfers only to and from physical host memory. However, user programs usually use virtual addresses to specify source or destination communication buffers. In order to obtain the corresponding physical addresses, the user application needs to perform a system call, which is an expensive operation in the common path. Therefore most modern user-level communication standards [11, 16] and network interfaces [12, 6] require efficient address translation on the NIC. To obtain physical addresses from virtual addresses, the NIC must maintain

some type of lookup data structure.

Second, to allow user programs to directly access the network without operating system intervention, network interfaces need to be able to verify user requests. For this purpose, the virtual memory regions that are used as communication buffers must be registered with the NIC. Registration implies that the network interface is aware of virtual memory regions that can be used for direct data transfer. Therefore user data transfer requests can be verified at the NIC level. The NIC must be able to maintain the information about a possibly very large number of registered communication buffers.

Third, such systems require a mechanism to authenticate remote programs. Because it is necessary to be able to limit access to registered buffers only to a subset of system users. In general, there are two alternatives to provide such a service: (i) to authenticate every incoming packet at the NIC level (ii) to set up a *connection* with the remote NIC, and authenticate the remote program only at the time of establishing the connection. The first case requires that the user program credentials are transferred with every request packet. In the second case the remote NIC is trusted. Therefore, it needs to maintain all protection information about all connections to the remote communication buffers. Since there may be many such connections, the size of the corresponding data structures might be large.

Finally, given that most servers today are required to support multi-programmed workloads, the NIC has to be able to support multiple user processes. For protection purpose, modern NICs can directly identify user processes by some handle without operating system intervention and they can lookup user requests to retrieve user-specific communication information.

The solution employed in most modern NICs is to perform all lookup operations by using static, on-NIC lookup tables. Although this solution has worked adequately until now, it imposes significant limitations for today's and future servers and systems that employ large memories, and support multiprogrammed workloads. Furthermore, modern

compute and database servers are easily equipped with multiple GBytes of memory. Since such systems process large working sets, they usually require the ability to transfer data to and from arbitrary locations of application virtual memory. For instance, recent work in using VI interconnects for database storage [28] or in scalable compute servers [17] has revealed that modern NICs cannot support the demands of today’s data processing applications and lead to customized solutions at higher system layers to alleviate these limitations. Furthermore, the static data structures result in high memory requirements for NICs. In some cases the price of such NICs is comparable to the price of the host workstation. Finally, static solutions imply that modern NICs cannot adapt to changing workload needs, requiring worst-case planning and resulting in expensive and difficult to replace interconnects.

In this thesis, we first categorize the major NIC data structures and functions that contribute to this problem. Then we propose moving all necessary data structures on the much larger host memory and using on-NIC memory as a stateless cache. We present the design and implementation of our schemes on a state-of-the-art programmable interconnect. In our implementation we use separate caches for each lookup function to accommodate their different requirements. Finally, we evaluate important aspects of the design configuration space by using both micro-benchmarks as well as real applications. Previous efforts to address these problems, most notably [8], [3], [22], and [19], have either provided solutions for only a subset of the issues or have evaluated their approaches with micro-benchmarks only, as discussed in detail in Section 2.

In the rest of this chapter we present the necessary background for user-level communication systems, we describe the problem addressed by this thesis, and we summarize our results.

1.1 User-level Communication

In this section we describe the general architecture of user-level communication systems and the main features they provide. First, we define some important terms that are used throughout the thesis.

- **Virtual Memory Handle:** Since the NIC handles requests from multiple processes simultaneously, it must be able to distinguish similar virtual addresses of different processes. We call the pair $\langle \text{Process Id, Virtual Address} \rangle$ *Virtual Memory Handle (VMH)*, in which the Process Id is some data item that identifies a process within a node. It can be the operating system assigned process Id, or any other unique identifier. VMH is used to lookup in NIC for address translation purposes.
- **Communication Buffer Handle:** Each communication buffer must have an identifier which is unique within the scope of a single process. Moreover some process identifier information must be added to the buffer Id to make it unique within a node. We call the pair $\langle \text{Process Id, Buffer Id} \rangle$ *Communication Buffer Handle (CBH)*. CBH is used by the NIC to obtain protection information about a buffer. The remote programs use the CBH along with remote node Id to specify a communication buffer in the cluster.
- **Process Communication Handle (PCH):** Given most today's servers are required to support multiprogrammed workloads, the NIC has to be able to support multiple user processes. For this purpose, modern NICs can directly identify user processes by means of a handle, without operating system intervention. We call this handle *Process Communication Handle (PCH)*.

1.1.1 General Architecture

Nodes in modern clusters are usually interconnected with low-latency, high-bandwidth SANs that support user-level access to network resources [13, 10, 6] and direct memory operations to local and remote virtual memory. By allowing users to directly access the network without operating system intervention, these systems dramatically reduce latencies compared to traditional TCP/IP-based local area networks.

Such NICs usually employ a communication assist, which can be a special-purpose network processor or a general-purpose processor. There is some amount of static or dynamic memory on the NIC. The size of the memory is in the order of few MBytes. It also has one or more DMA engines for transferring data between host memory and the NIC memory, and also between the NIC memory and the network link. A firmware-based control program runs on the communication assist and implements the communication protocol by managing NIC resources, mainly controlling the DMA engines and responding to system events.

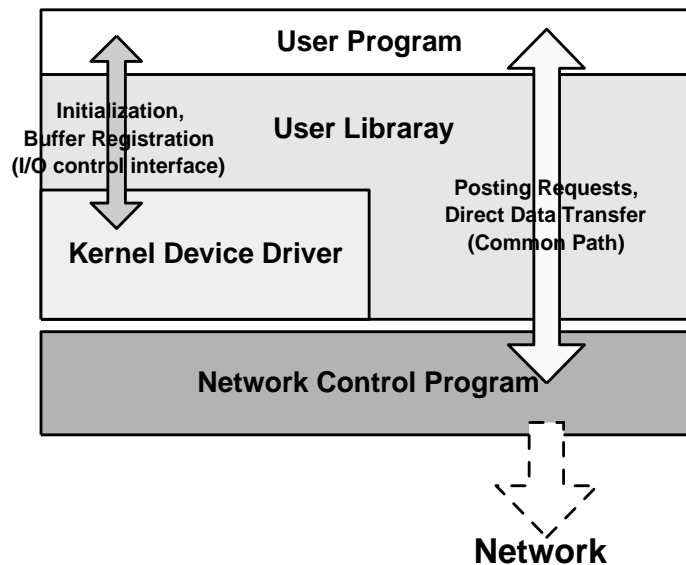


Figure 1.1: The general architecture of user-level communication systems.

Figure 1.1 shows the general architecture of most user-level communication systems. There are three major components in such an architecture:

- A kernel-level device driver that is responsible for initializing the network interface and performing trusted functions that cannot be performed directly by the user. Such functions are performed through a set of additional system calls (I/O control calls) provided by the driver. The most important of these functions is communication buffer registration. Moreover, the device driver handles NIC interrupts. Such interrupts can be issued for various purposes, such as internal communication between the NIC and the host, protocol processing tasks, debugging, and logging.
- A Network Control Program (NCP) that runs on the NIC and performs all protocol processing tasks. The NCP usually implements a set of state machines that handle all system events. The state machines are designed to allow for maximum concurrency and minimum context-switch cost. It usually implements a pipelined architecture for transferring data from the host memory to the NIC memory and then over the network, and vice versa.
- A lightweight user-level library with which user programs link in order to use the communication API for establishing connections and transferring data. The user-level library in co-operation with the NCP manages per-process send and receive queues on NIC memory. Each process has a set of send, receive, and completion queues in the NIC memory. Most of the time, the queues are mapped to the process address space so that the user program is able to put requests in the queues directly.

1.1.2 Major Operations

User-level communication systems implement two major types of operations: connection establishment and data transfer. These operations are explained in more detail in the following paragraphs.

Connection Establishment: This type of operations establishes a communication channel between the sender and the receiver that is used later on to transfer data back and forth. Given that user-level systems are required to support direct transfers to and from application virtual memory, these operations initialize the required data structures for memory protection, data transfer, and user authentication.

Data Transfer: Most user-level communication systems provide two categories of data transfer operations.

- **Send and Receive Operations:** In a send operation the sender program creates a send request descriptor that contains information about the source of the data, i.e. the send buffer VMH and the length of the data to be sent, as well as information about the destination that is node Id and process Id. The sender program submits the request directly to the NIC send queue and notifies the NIC that a new request for data transfer is available. On the receive side, the receiver program must post a receive descriptor to the NIC to mention the receive buffer virtual address. When the data packet arrives at the receive side, it is first transferred to a data buffer in the NIC memory. The NIC then DMA's the received data to the VMH of the receive descriptor directly to the user memory using the virtual-to-physical address translation mechanism.
- **Remote DMA (RDMA) Operations:** These operations decouple control transfer from data transfer. That means the receiver node is not interrupted when data is received. The data is directly transferred into the user process address space without a notification. This reduces CPU overhead for the receiver in the case of network I/O intensive applications. There are two RDMA operations: *RDMA read*, and *RDMA write*. With RDMA read, the data is fetched from a remote communication buffer, and put into a buffer region in the requesting node address space. With RDMA write, the data is copied from a region in the requesting process

address space to a remote communication buffer. In contrast to send and receive operations, the communication buffers control information, such as protection and address translation information, must be permanent throughout the program execution. In send/receive operations, such control information is only needed at the send or receive time, and will no longer be in use after the completion of the operation.

1.1.3 Memory Pinning

The virtual-to-physical address mapping of a virtual page must not change while a DMA operation on the page is in progress. Thus, every page needs to be *pinned* in physical memory during DMA. In most operating systems there are system calls for this purpose (e.g. `mlock` in Linux and `MmProbeAndLock` in WindowsNT). There are limitations on the number of pages that may be pinned. First, the number of pinned pages cannot exceed the total number of pages in the physical memory. Second, pinning too many pages might affect the overall system performance. Finally, systems in which the communication buffers need to be pinned statically, limit the size of the communication buffers an application can use. In such situations, either the application or the system with hints from the application must pin and unpin the pages dynamically.

1.2 VMNC System Overview

In this work, we use Virtual Memory Mapped Communication (VMNC) [10] as the base for implementing our approach. VMNC provides protected, user-level communication between the sender's and the receiver's virtual address spaces. Before communication can take place, the receiving process registers areas of its address space where it is willing to accept incoming data with a set of permissions. Communication is protected in that the exporter can restrict the processes and hosts that can access each buffer. VMNC guarantees FIFO message delivery between any two processes in the system and tolerates transient network errors by using packet retransmission. Finally, although VMNC implements advanced memory management functionality and handles real-life applications and setups, it is fairly portable. It currently runs both under Windows and the Linux operating systems.

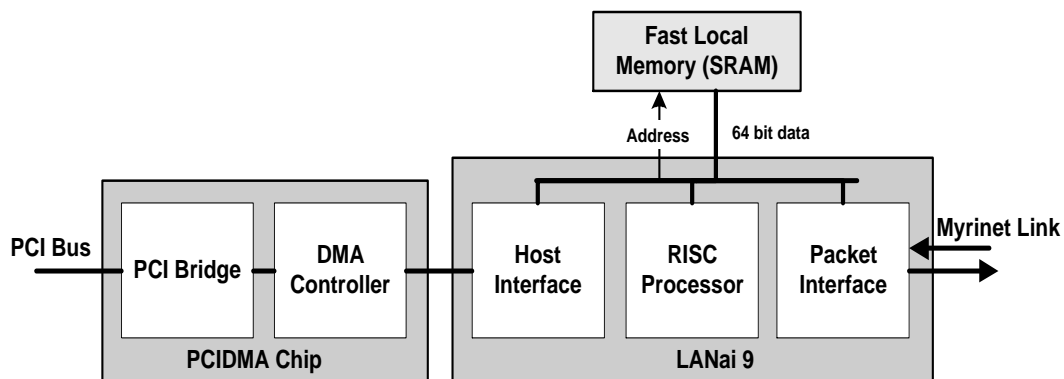


Figure 1.2: The block diagram of the Myrinet network interface controller.

VMNC is implemented on top of the Myrinet interconnect [6]. Figure 1.2 shows the block diagram of the Myrinet NIC architecture. The PCI-based NIC is composed of a 32-bit, general-purpose control processor (LANai9) with 2 MBytes of SRAM that executes the network control program (NCP). The SRAM is used for network buffers as well as for firmware code and data. Each NIC has three DMA engines: two for transferring data between the network and the SRAM and one for transferring data between the SRAM

and the host main memory over the PCI bus. The DMA engines can be controlled either by the host or the LANai9 processor. The host can also access the SRAM using programmed I/O.

1.2.1 VMMC Operations

- **Buffer Registration (Export):** In this operation, a process specifies a region of its address space as a communication buffer and performs a system call to register the buffer. This gives permission to remote nodes to read from and write to the specific region. The process has the option to be notified of data transfers to and from the buffer by registering a user-defined function as *notification-handler*. The registration operation requires pinning all pages of the buffer. The pages will only be unpinned when the buffer is deregistered.
- **Connection Establishment:** Before any access to a remote communication buffer, a connection to the buffer must be established. This is done by *importing* the remote communication buffer. The importer process must provide the remote buffer CBH in addition to the protection keys to the local NIC. Then the local NIC establishes a connection with the remote communication buffer and saves connection information in the NIC memory and gives an *Imported CBH* to the user process. The import operation enables the local NIC to filter unauthorized requests.
- **RDMA Operations:** VMMC implements both RDMA read and RDMA write. In these operations the user process specifies the CBH of the remote buffer, the offset within the buffer, the length of the data to be transferred, and the local virtual address that is the source/target of the operation depending on the direction of the data transfer. As mentioned above, the CBH of the remote buffer can only be specified indirectly by specifying the imported CBH. For both operations the data transfer happens directly to and from the remote buffer without interrupting the

remote node. However, a *notification* can be used to inform the remote node about the data transfer. In this case, the RDMA operation is issued with a notification flag. On receipt, the remote NIC raises a notification interrupt after the data transfer is complete. Although notifications can be used for both RDMA operations, they are usually provided only with RDMA writes since the remote node might be waiting for the written data.

1.2.2 NCP Memory Map

In general, the memory on modern NICs is usually divided into the following major conceptual regions:

- **NCP Code Region:** This region contains the firmware code that runs on the communication assist and handles all system events. This code can vary between 50-150 KBytes. Although it is important to reduce code size, this is usually not an issue for NICs that support user-level communication due to the complexity of the functions they offer and the fact that code size is mostly independent of system and application parameters.
- **Send and Receive Data Buffers:** Data that is being received or sent, is first transferred to buffers on the NIC. Such buffers are short-lived and usually a small number is adequate for good performance [23]. Furthermore, such buffers are always managed dynamically and their number can be adjusted based on the available NIC memory.
- **Lookup Data Structures:** This region contains all data structures needed for lookup operations. For instance, virtual address translation is required both on the send and the receive paths for translating between virtual and physical addresses for send and receive buffers. Furthermore, based on the protection model employed in each system, the NIC maybe required to maintain control information for memory

segments that can be used in data transfers. Unlike the NCP code and send and receive buffers, the size of the lookup data structures grow with the application demands. For instance, the data structure used for virtual address translation grows with the growth rate in the size of the virtual address regions used by user applications as communication buffers.

Thus, the lookup data structures used on modern NICS constitute the major bottleneck in memory usage. Next, we describe the different lookup data structures more in detail.

1.2.3 Lookup Data Structures

Most of the lookup data structures are almost entirely maintained in memory on the NIC and are statically managed for performance reasons. Figure 1.3 illustrates how messages are exchanged in VMMC.

VMH Lookup Table: On the send side, messages are transferred to the NIC in one of two ways: programmed I/O for small messages (≤ 32 bytes) and DMA for medium and large messages (> 32 bytes). Programmed I/O involves the host CPU transferring data directly to the NIC address space as opposed to data movement by the DMA engine. The NIC uses local virtual-to-physical address translations. Messages larger than 4 KBytes are segmented into chunks by the NCP. On the receive side, the packet is first deposited in a receiving queue on the NIC. The NIC then checks if the sender has permission to access the buffer specified in the packet. If yes, the NIC uses the local virtual-to-physical memory translations to directly transfer the incoming packet into host memory without having to interrupt the host processor.

CBH Lookup Table: The CBH lookup table in VMMC and most systems is static and is managed entirely on NIC memory. Each CBH lookup table entry is a 36-Byte communication buffer descriptor including the location of the buffer in the process address space, protection information and, other control information.

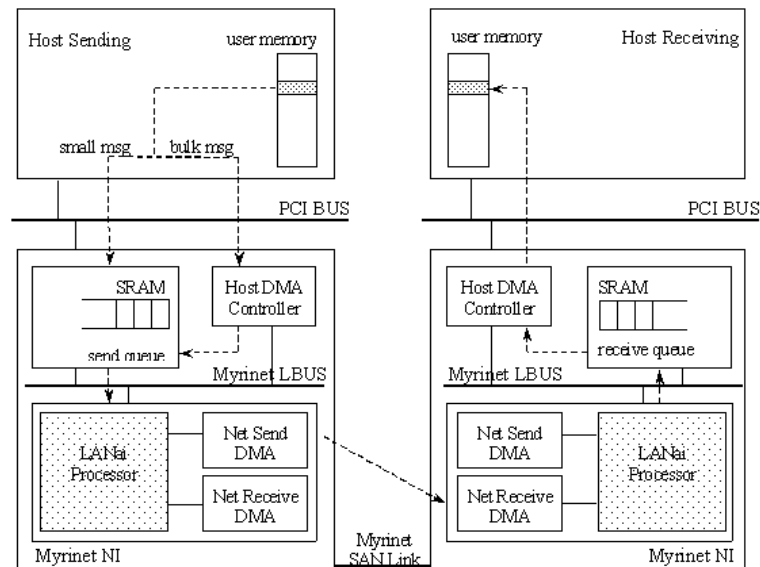


Figure 1.3: Message exchange in VMMC. The arrow points to the direction of data flow, for a typical send operation.

The number of buffers in CBH table is limited to a few hundreds or thousands, which is not sufficient for modern systems that can hold multiple GBytes of memory, especially when applications require the use of many separate communication buffers to avoid data copying.

Imported CBH tables: In VMMC in particular, each remote communication buffer needs to also be imported locally, during connection setup, before it is used. This requires the existence of tables to maintain information about the imported buffers. In VMMC there is one imported CBH table per user process. Each entry in these tables stores information about a remote registered buffer, including its location, length, and protection keys.

The memory of these tables are managed statically. This results in limitations on the total number of buffers that can be imported by each user process. It also limits the number of processes that can use the system simultaneously.

PCH table: As mentioned above, for multiprogrammed environments, the NIC needs to maintain context information for each program that uses the network. This

information is used when the process posts requests directly to the NIC as well as when packets arrive from the network and need to be delivered to process memory. For performance reasons the process communication context information is maintained on the NIC. The main piece of information in this process context is a dedicated post queue for user data transfer requests. Post queues are private for each process for protection reasons. The size of each queue is usually between 4 and 16 KBytes. The queue is mapped to the virtual address space of the user process and can be accessed with simple load and store operations through the user-level library of the communication system. Supporting many users in a multiprogrammed environment may increase memory requirements significantly.

1.3 This Thesis

1.3.1 Dynamic Handle Lookup

We see that lookup data structures are a major limitation both for building modern NICs and for handling large applications. We propose a generic scheme to maintain all important lookup information in the larger host memory, and use parts of the NIC memory as caches for current working set of various lookup information. These information are brought to the NIC memory on-demand. We apply this scheme for VMH, CBH, and PCH lookup information similarly. However, there are some differences in the low-level implementation mechanisms for different lookup information. Also we propose to eliminate the import CBH information from the NIC memory, and authenticate the remote programs on every request with an efficient key system.

1.3.2 Summary of Results

The major high-level conclusions of our work are the following:

	Base System	<i>miNI</i>
Code+Static Data	74 KBytes	87 KBytes
Send and Receive Buffers	200 KBytes	200 KBytes
Lookup Tables on NIC	524 KBytes	132 KBytes
VMH Info	265 KBytes	75 KBytes
CBH Info	163 KBytes	25 KBytes
Imported CBH	64 KBytes	0
PCH Info	32 KBytes	32 KBytes
Total	798 KBytes	419 KBytes

Table 1.1: The breakdown of NIC memory consumption in the base system and *miNI*.

- Our approach relaxes the limitations of most use-level communication systems. Such limitations are on the total size of the communication virtual address space, total number of communication buffers, and the number of processes that use the system concurrently.
- NIC memory requirements can be reduced substantially. By using dynamic handle lookup we are able to reduce the total NIC memory from about 800 KBytes to about 400 KBytes. In particular, the memory used for all lookup structures is reduced by about 80% from about 520 KBytes to about 130 KBytes. Table 1.1 summarizes the breakdown of the memory consumption for the code and various data structures on the NIC memory in the base system and after our modifications.
- For the VMH lookup in particular, we find that small caches (about 16K entries) are adequate to support large host memories. Large cache lines even up to 128 entries, result in efficient prefetching. Further reductions in memory requirements are possible with slightly higher performance penalties.
- Overall, system performance may be affected significantly by memory management

and in particular cache design and simple solutions may not be adequate for modern network interfaces.

- The impact of extra overhead of using more complex dynamic lookup structures relative to the static data structures used in VMMC, on the user application performance is not significant across most applications we examine.

1.3.3 Overview

The rest of the thesis is organized as follows. Chapter 2 discusses the related work. Chapter 3 describes the design and implementation of the data structures and mechanisms for dynamic handle lookup. Chapter 4 presents the results of our performance evaluation and analysis, both with synthetic micro-benchmarks and real applications. Finally, Chapter 5 draws our conclusions and provides directions for future work.

Chapter 2

Related Work

In this chapter we review the most relevant recent research in user-level communication. For each work we describe the most important characteristics related to the NIC memory management issues in user-level communication.

The authors in [8] propose a User-managed Translation Look-aside Buffer (UTLB) to dynamically manage translations for virtual memory used as send communication buffers. UTLB is implemented as a per-process two-level custom page table in the device driver memory. A shared TLB cache resides in NIC memory and caches the most popular entries of the driver-level tables. On a miss the NIC fetches the appropriate table entry by DMA, potentially replacing another cache entry. The UTLB approach handles all misses in the receive path statically. To limit the number of required DMA operations for each miss to one, the top-level, process page table (one page per process) is kept in the NIC memory. There is a user-level page bitmap of pinned pages. When a process sends a buffer, it first looks the bitmap and if the buffer pages are already pinned, it proceeds. Otherwise it asks the device driver to pin the pages. The NIC control program assumes the pages for miss requests have already been pinned and the corresponding entries in the custom page tables are set. Therefore, this scheme works only for the send path in VMMC. All translation in the receive path are handled statically.

In fact in the VMMC system all pages for registered communication buffers are pinned down statically at the registration time, and they remain pinned until the communication buffers are deregistered. All the address translation for the registered buffers reside in the NIC memory. That results in large memory requirements and limitations on the total size of the communication buffers. Unlike the UTLB approach, our system uses a uniform mechanism for address translation of handles for communication buffers, both in the send and receive path. Furthermore, we dynamically manage the CBH lookup information. Our extensions allow for arbitrary amounts of virtual memory to be accessed by remote nodes as communication buffers.

The authors in [19] propose the concept of virtual networks to address the issue of supporting large numbers of users over fast, user-level communication systems. The main goal of the work is to efficiently multiplex the system resources among multiple applications with different demands. The main abstraction is the network *endpoint*. An endpoint consists of the process send queue and receive queue and the address translation information of the static buffers involved in the communication. The endpoint is a coarse-grain unit of resource allocation. Each application can have several endpoints. Endpoints reside in host memory and are cached on NIC memory. Resident endpoints can be used by the corresponding application. Unlike many modern user-level communication systems, the virtual networks abstraction does not support RDMA operations to arbitrary memory locations, which eliminates the need for manipulating large virtual memory for the communication buffers, and also very large number of communication buffer handles.

U-Net/MM [3] is an extension of the U-Net [2] user-level communication architecture that also uses endpoints as the main communication facility. Each endpoint is associated with a buffer area that is pinned to contiguous physical memory and holds all buffers used with that endpoint. The U-Net/MM incorporates a TLB structure, in order to handle arbitrary user-space virtual addresses. Unlike the UTLB approach, but similar to our work, U-Net/MM uses an interrupt-based mechanism for handling TLB misses.

There are, however, significant differences between U-Net/MM and *miNI*. First, there is no RDMA operation in U-Net/MM. Pinning and unpinning actions can be guided by the application upon posting the send and receive descriptors, whereas with RDMA operations there is no such information. Second, in U-Net/MM whenever there is an eviction from the TLB, the operating system has to be notified to unpin the corresponding pages. In *miNI* the address translation information on the NIC memory is a cache of a larger data structure of pinned pages in the host memory. Therefore, there is no need to notify the host system in the case of an eviction. Third, *miNI* deals with protection issues in RDMA operations by caching communication buffer descriptors from host memory and allows for using arbitrary number of virtual memory regions to be used for communication. Finally, the authors in [3] use micro-benchmarks in their evaluation, whereas we use both micro-benchmarks and real applications to study the effect of various system parameters on miss rates as well as overall system performance.

The Virtual Interface (VI) architecture is an industry standard for user-level communication. The main abstraction in this standard is *Virtual Interface(VI)*. Each VI is a communication endpoint. The system provides bidirectional point-to-point communication between two VIs. Each VI has a pair of send and receive queues. Moreover, each VI can be associated with a *Completion Queue* in which the NIC posts the status of the requests posted in the send or receive queue of the VI.

The VI supports both send and receive operations as well as RDMA operations. Similar to our system, communication buffers must be registered prior to any data transfer. The registration includes both pinning the virtual region and setting up the protection information for it. In current VI implementations ([7],[20],[1], and [12]) all registered regions are statically pinned. This limits the total size of the registered buffer to a fraction of the total physical memory available, unless the user process takes care of dynamic registration and deregistration of the communication buffers. For instance the cLAN NICs [12] impose 1GByte limit on registered memory.

Infiniband [16] is the evolution of several industry projects in the system area networks architectures. It defines a switched communication fabric allowing many devices to concurrently communicate in a protected environment. The core operations and protection model of Infiniband is significantly influenced by VI. More specifically, it follows the same rules as VI in registering the communication buffers with small refinements in the protection model. To date, there is no complete implementation of Infiniband.

The authors in [22] present a survey of different mechanisms used for address translation in network interfaces. They define four requirements for address translation: (i) Flexibility of use for higher system layers. (ii) The ability to cover all of the user address space. (iii) The ability to take advantage of locality. (iv) Graceful degradation when system limits are exceeded. Then, by using simulation, they analyze and evaluate the advantages and disadvantages of various alternative methods of implementing address translation in network interfaces. They find that hardware lookup structures in the NIC are not required since the software schemes are fast enough. They suggest that the NIC should handle all address translation misses which introduces the limitations discussed in this work.

Finally, there have been efforts to eliminate the need for on-NIC address translation. Virtual DMA mechanisms and Programmed I/O mechanisms are two examples of such efforts [25, 21]. However, most modern communication systems and standard still use physical DMA as the main data transfer mechanism between the NIC and host memory.

Chapter 3

Design and Implementation

In this chapter we describe *miNI*, and the design and implementation of the dynamic structures for handling the most important lookup operations in the NIC.

3.1 Overview

3.1.1 Network Interface Cache

The main idea in managing dynamically the VMH and CBH tables is simply to move all necessary data structures to the much larger host memory and use the on-NIC memory as cache. On a cache miss, the NIC issues an interrupt to the host CPU. The device driver is responsible for handling miss interrupts and fetching entries that miss from host memory lookup structure and putting them into the cache data structure on the NIC.

Unlike the base system, we do not limit the number of processes that concurrently use the communication system. This is because we do not statically partition the on-NIC cache structures among the processes. However, the performance of the communication system might substantially degrade if many processes compete for the same NIC cache entries. The two major problems in such a design are (i) how to fetch the missed entries from the host memory and (ii) whether it is possible for the NIC to wait until the missed

entries are fetched or not. We discuss alternative solutions to these problems next.

3.1.2 Interrupt-based Miss Handling

There are two ways for the NCP to get data from host memory: (i) interrupting the host CPU, (ii) fetching data directly from host memory by DMA. The second case is faster and does not put any overhead on the host CPU, However it has a number of disadvantages.

First, it requires that the host CPU prepares all required information before the operation that needs the information takes place. This is an issue in the case of address translation for RDMA operations, it may not be impossible to pin all registered communication buffers in advance.

Second, the NIC must be aware of the physical location of the data items in the on-host data structures. The NIC must implement a data structure to maintain such physical address information. The large size of this data structure may become a problem, unless we put the on-host data structures on some physically contiguous areas of host memory. Moreover the design of the on-host data structures has to be simple to avoid complex remote lookup procedures from the NCP.

In contrast, the interrupt-based method allows the device driver to prepare the required entries on demand. Moreover, it has the advantage that the device driver may organize the lookup data structures arbitrarily.

In the interrupt-based method, there are two ways to handle a cache miss: (i) In the context of the interrupt handler (e.g. bottom-half handlers in Linux). (ii) Using a separate thread in the kernel. The first alternative is faster since there is no context switch overhead and unpredictable scheduling delay. However, it has two shortcomings.

First, certain calls in the kernel API, such as memory pinning and unpinning, cannot be performed in the context of the interrupt handler. This approach requires implementing custom calls to perform these or similar functions, which may not be possible

in proprietary operating systems, reducing the portability of the communication system. Second, the interrupt handler cannot be blocked, limiting the types of system functions that can be called in the handler context.

For these reasons, in this work we choose to handle VMH lookup cache misses in a separate kernel thread.

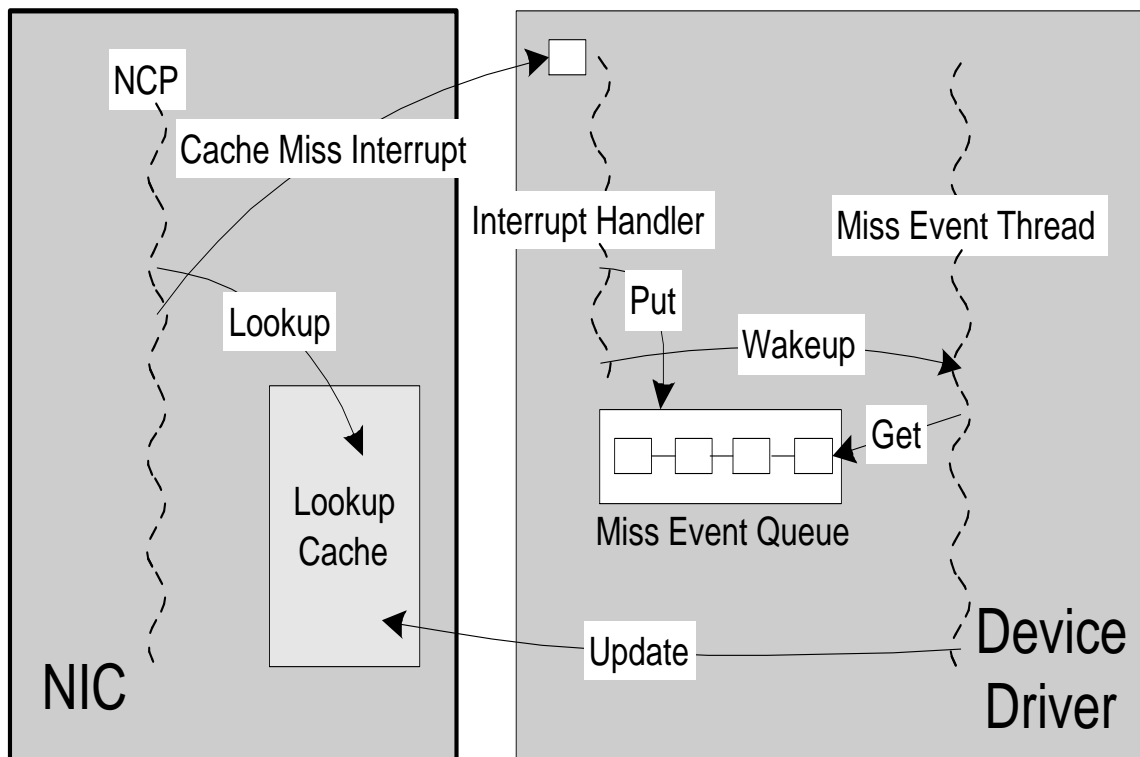


Figure 3.1: Cache miss handling path.

Figure 3.1 shows our design for miss handling. The NIC issues a miss interrupt when a lookup in one of the cache data structures misses. The interrupt handler in the kernel device driver just prepares a *miss event descriptor* with the information given by the NIC and places it in the miss event queue. It then wakes up the in-kernel *miss event thread*, which does all processing and updates the on-NIC cache.

3.1.3 The Role of Packet Retransmission

In our system we take advantage of packet retransmission to avoid buffering requests that miss on the receive side. VMMC includes a reliability mechanism [23] that retransmits packets until they are acknowledged by the receiving node. More specifically, VMMC deals with transient network failures and provides applications with reliable communication at the data link layer. The goal is to tolerate CRC errors, corrupted packets, and all errors related to the network fabric; links and switches can be replaced on the fly. VMMC implements a simple retransmission protocol at the data link layer between network interfaces. The scheme buffers packets on the sender side and retransmits when necessary. Each node maintains a retransmission queue for every node. Each packet carries a unique sequence number for the sender-receiver pair. Sequence numbers and retransmission information is maintained on a per-node and not on a per-connection basis. The receiver acknowledges packets. Each acknowledgment received by the sender, acknowledges (and frees) all previous packets up to that sequence number. There is no buffering at the receiver. If a packet is lost, all subsequent packets will be dropped. However, if a previously acknowledged packet is received again, it is acknowledged. In the current implementation, there are no negative acknowledgments for lost packets.

Therefore, when there is a cache miss for a received packet, the packet will simply be dropped with no acknowledgment. The retransmission mechanism in VMMC guarantees the packet will be retransmitted. Meanwhile, the miss handling procedure places the missed entries in the corresponding cache.

In general, by using the retransmission mechanism we eliminate the need for receive-side buffering and significantly simplify the design of the NIC.

3.2 VMH Lookup

VMH lookup is required both in the receive and the send paths. Receive path is the path in which the data is transferred from the network link to the NIC memory and then from the NIC memory to the host memory. Send path is the path in which the data is transferred from the host memory to the NIC memory and then from the NIC memory to the network link. In the next two paragraphs, we describe how we obtain VMH in both receive and send paths.

In the receive path, there are two cases. First, when the received data is the reply of an RDMA read request issued from the local host. In this case the VMH of the destination of the data on the host memory has already been posted at the time of issuing the RDMA operation. Second, the received data has to be written into a communication buffer, as a result of an RDMA write operation, issued from a remote node. In this case, the RDMA request contains: (i) the CBH of the communication buffer, (ii) the offset within the buffer. The CBH is used to lookup virtual address of the start of the communication buffer. This address is added to the offset to get the actual virtual address of the area to which the data is supposed to be written. Then the VMH of the area can be composed from this virtual address and the process Id field in the CBH.

Similarly there are two cases in the send path. First, data is sent as a result of an RDMA write request, issued at the local node. In this case, the VMH of the source data is in the request descriptor posted by the user process. Second, the data is sent as the result of an RDMA read request, issued from a remote node. In this case the RDMA read request contains: (i) the CBH of the communication buffer, and (ii) the offset of the data to be read within the buffer. The CBH is used to lookup virtual address of the start

of the buffer. The start address is added to the offset to get the actual virtual address of the area from which the data must be sent. Then the VMH of the area can be composed from this virtual address and the process Id field in the CBH.

There are two data structures that hold the address translation information used during VMH lookup: (i) a *VMH Lookup Table* that resides in host memory, and (ii) a *VMH Lookup Cache* which is located in NIC memory and acts as a cache for the VMH lookup table. In the next two sections we describe the design of each of these data structures.

3.2.1 VMH Lookup Table

For each process, there is one VMH lookup table in the device driver that keeps the virtual-to-physical address mappings for all pinned pages of the process. When there is a miss in the VMH lookup cache, the device driver looks for the missed entries in the VMH lookup table. If they are found there, the driver just updates the VMH lookup cache. Otherwise, it pins all the pages in the missed address range and obtains their physical addresses by calling existing operating system kernel functions. Then, it updates both the VMH lookup table and VMH lookup cache with the physical address information.

The VMH lookup table stores the address translation information both for the send path and for the receive path. Moreover, there is no distinction between the pages of the registered buffers and the virtual regions that are used as the source or target of RDMA operations in the issuing nodes.

The VMH lookup table structure is a two-level page table similar to the system page tables used for virtual memory management. The difference is that only the pinned addresses have valid entries in the VMH lookup table. Although possible, we do not use system page tables for portability and performance reasons. The device driver needs to be able to traverse the VMH table. However, the operating system does not always provide an interface for manipulating page tables at this level. By using our private VMH

table, *miNI* is more portable and can optimize the VMH lookup table for its own use.

Since there is a limit on the total number of pinned pages in the system, the VMH lookup table cannot grow arbitrarily. In fact by limiting the total pages in the VMH we can control the number of pinned pages per process. For this purpose there is a high and low water mark. When the total number of pinned pages in a VMH lookup table reaches to the high water mark, a replacement algorithm is activated to unpin a set of virtual pages and evict them from the VMH lookup table, until the total number of pages in the VMH lookup table becomes equal to the low water mark. The actual values for the high water mark and low water mark depend on the application behavior and can be tuned dynamically.

The eviction must be synchronized with the VMH lookup cache to avoid unpinning an in-use virtual page. In our implementation we conservatively assume all entries that are in the VMH lookup cache as potentially in-use and avoid evicting them from VMH lookup table. This is both efficient and easy to implement. However, it is suitable only for systems in which the size of the VMH lookup table is significantly larger than the VMH lookup cache, so that there are sufficient out-of-cache candidates for unpinning.

3.2.2 VMH Lookup Cache

Figure 3.2 shows the structure of the VMH Lookup cache. We use a set-associative cache structure with configurable associativity, cache line size, and cache size. We use the lower bits of the virtual address in the VMH as the set index and the rest of the virtual address bits and the process Id as the tag information. This prevents static partitioning of the cache among processes. Also, it reduces the chance of conflicts among the addresses close to each other for which we expect spatial locality of access. The VMH lookup cache is shared among all processes using the system. Furthermore, the send and receive paths of the NIC use the same cache for uniformity and simplicity.

Unlike most on-NIC address translation data structures, the VMH lookup cache

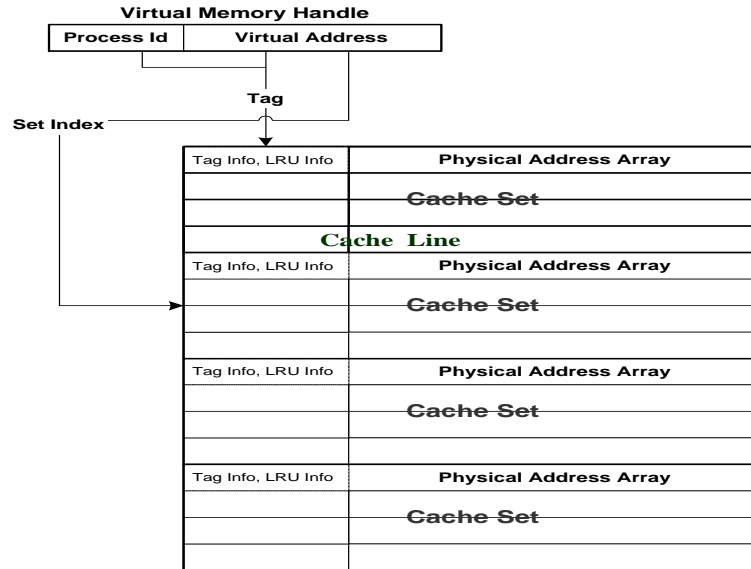


Figure 3.2: The structure of the VMH lookup cache.

lines can have multiple entries. The cache line size can be configured at compile time. Multiple-entry cache lines allow for a smaller tag-array. For each entry in a cache line, the full 32-bit word is allocated for physical page address, allowing up to 16 TBytes of physical memory to be used for communication buffers.

The unit of data transfer from the VMH lookup table to VMH lookup cache is a cache line. Therefore, when there is a miss for a virtual address in the cache, the miss virtual address that is reported to the device driver will be aligned to the cache line boundaries. This may result in effective prefetching for applications with sufficient spatial locality in their communication buffer access pattern. However, this increases cache miss penalty per miss, mostly due to larger pinning costs. But as our results show in the next chapter, for most applications, effective prefetching overshadows the increased miss handling penalties.

The VMH lookup cache can also be configured with different set-associativities. Although higher associativities may reduce the number of conflicts, they increase the cost of tag lookup, especially given the fact that the VMH lookup cache is implemented in software.

When there is a data transfer from or to a virtual page in a cache line, the whole cache line will be locked down in the VMH lookup cache, preventing any line entry from being replaced. For this purpose, we assign a *lock* bit for each cache line. We set the lock bit when there is an on-going DMA from a page of the line and reset the bit once the DMA is complete. Since different areas in a cache line can be used for DMA in the send and receive paths simultaneously, we duplicate the lock bit for the send and receive paths. Moreover, since the NCP implements a complex state machine, it is not easy to deal with all deadlock and live-lock scenarios that can happen due to concurrent lock and wait-for-lock operations on each VMH lookup cache entry between the send and receive paths. For this reason and to simplify the NCP design, we have implemented an atomic *lock-use-unlock* scheme between the send and receive paths that assumes an associativity level of at least two. This allows us to also relax contention over a single cache line between the send and receive path.

The VMH lookup cache uses an LRU replacement policy for the entries of each set. The cache replacement policy is fully implemented in the NIC. Since the NCP access to the VMH lookup cache is read-only, the replacement does not include any write-back operation and the entries to be replaced are simply thrown away. On a miss, the NCP applies the replacement policy to free some spots in the cache, and reserve them before issuing the interrupt. These cache spots remain reserved until the device driver puts the missed entries in them, and marks them as valid entries.

When an entry is replaced in the VMH lookup cache, it remains in the host VMH lookup table. Entries in the VMH lookup table are pinned as well. However, as mentioned above, once a line is evicted from the VMH lookup cache, it also becomes a candidate for eviction from the VMH lookup table, in case it reaches the high water mark. However VMH lookup table replacement happens periodically with long intervals. If a page is accessed frequently it will be brought back to the VMH lookup cache and thus, will not be evicted from the VMH lookup table.

3.3 CBH Lookup

Similar to the VMH handle lookup, the CBH table is located in host memory whereas the CBH cache resides in NIC memory. The CBH cache is separate from the VMH lookup cache. There are important differences between the two caches and how the lookup operations are performed, as we describe next.

3.3.1 CBH Lookup Table

There is one CBH table per process, located in the device driver memory. The buffer Id of each incoming packet is used as a key to lookup the appropriate communication buffer descriptor. Since in our system, there is no restriction on the value of the buffer Id, we implement the CBH table as a hash table. The communication buffer descriptors in the CBH table contain a superset of the fields available in the CBH cache with the extra descriptor fields used only by the device driver.

To register a region for use as a communication buffer, the user program uses a system call provided by the device driver through the device I/O control interface. The driver allocates a descriptor for the new buffer, initializes it, generates a random protection key, and inserts the newly created descriptor to the CBH table. In order to optimize the dynamic allocation of the buffer descriptors from the kernel memory, all buffer descriptors are allocated from a big descriptor pool that is shared among all processes. We use the lower bits of the buffer Id as the hash index. However, more sophisticated hash functions might be required for applications with thousands of communication buffers to distribute the buffers into the hash bins more evenly.

The descriptors in the CBH lookup table are permanent until the user process de-registers a specific buffer. In this case *miNI* invalidates both the descriptors in the CBH lookup table and the CBH lookup cache. However, the address translation entries of the buffer that are in the VMH lookup table and the VMH lookup cache will not be

invalidated, since they might be used by the user application as the source of an RDMA operation. If such entries are not used for a long time, they may be evicted, first from the VMH lookup cache, and then from the VMH lookup table.

3.3.2 CBH Lookup Cache

The CBH cache contains a subset of the communication buffer descriptors in the CBH table and resides in NIC memory. Each descriptor in the cache has a size of about 26 Bytes, including the virtual address and the length of the buffer, as well as the protection keys. From the CBH, the lower bits of the buffer Id are used as the cache index, whereas the rest of the bits in the buffer Id and the process Id are used as the tag. Figure 3.3 shows the structure of the CBH lookup cache.

The CBH cache is set-associative and the level of associativity can be configured at compile-time. Similar to the VMH lookup cache, the replacement algorithm within a cache set is LRU. The replacement is done by the NIC. Since the descriptors in the cache are read-only for the NIC, the replacement does not include any write-back operation. Unlike the VMH lookup cache, the cache line size is always one, since we do not expect any locality of access in terms of the communication buffer ids.

When the NCP receives a packet from a remote node, it extracts the destination buffer and process Id and checks the CBH cache for a matching entry. On a cache hit, the NCP will proceed with handling the incoming request. On a cache miss, the NCP reserves an entry in the CBH cache and drops the incoming request. Then, it raises an interrupt and notifies the device driver for a *CBH cache miss* event. The event handler in the driver searches for the requested descriptor in the CBH lookup table of the corresponding process. If the buffer request is valid, the driver allocates a CBH cache entry and inserts the buffer descriptor. If the buffer request is invalid, the driver marks the CBH cache entry with an *invalid* flag. When the network request is retransmitted through the NCP retransmission mechanism, the NCP will respond to the request with

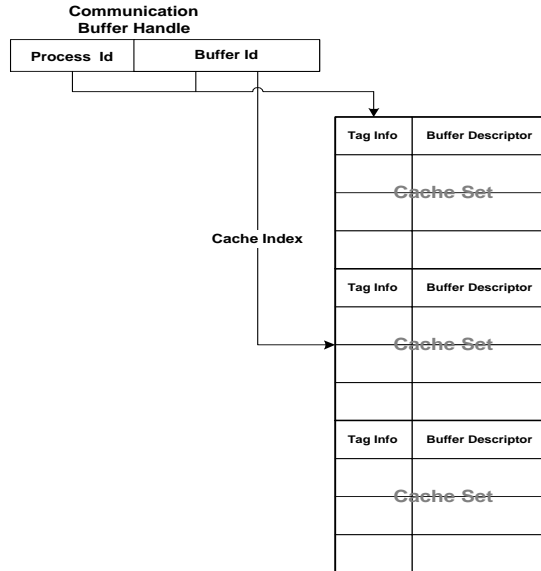


Figure 3.3: The structure of the CBH lookup cache.

an error code.

3.4 Imported CBH Table

Our analysis in this work has shown that the benefits from maintaining in each node information for remote communication buffers does not justify the memory required to support it. Each imported communication buffer descriptor requires about 16 Bytes. In our base system the imported CBH table size exceeds 64K per process. Due to the static management of this table, the number of imported CBH descriptors is limited to 4096. Moreover, managing imported buffers information significantly complicates the implementation of the RDMA operations. For these reasons, we have decided to eliminate the imported CBH table from the NIC memory and we move all information about remote buffers to the user library memory. All RDMA read and write requests posted by the user to the NIC contain information about the remote communication buffer, namely the node Id, the CBH of the buffer, and its protection keys.

The implication of this modification is that the NIC at the requesting node is not able

to verify if the user request is valid. However, this check is still performed on the receive side. Although this modification changes the original protection model, the practical implications are minor and the advantages outweigh the shortcomings.

A user can maliciously flood the interconnect with invalid requests that will be dropped on the receive side. However, this is not an issue for the system we examine. Moreover, in most cases the user has other means of achieving the same goal, even if the send-side NIC can check against such behavior. On the other side, removing the import buffer descriptors from NIC memory results in reducing the memory requirements, improving the performance of the common-case translation path, but most importantly simplifying the on-NIC translation code, which is a significant consideration for all systems at this level of complexity. In general, we believe that in most system designs, performing access checks either only on the receive or only on the send side (depending on the trust model) is adequate. In our implementation we perform all protection checks on the node that owns each communication buffer by means of the CBH table.

To verify incoming requests, we use a capability mechanism based on keys. In our scheme, a random key is generated for every communication buffer at registration time. The key is large enough number (64 or 128 bits) and hard to guess. If a communication buffer needs to be used by a remote node, the key is sent to the node during connection establishment phase. On every operation after establishing the connection, the remote node has to provide this key, or otherwise the node that owns the buffer will consider the operation as invalid and drop the request. Keys are invalidated once the owner decides to deregister the communication buffer.

3.5 PCH Table Memory Requirements

As mentioned in section 1.2.3, the private post queue for each user process may increase memory requirements significantly in multiprogrammed environments. In our work we

propose maintaining a small number of post queues on the NIC and sharing them among multiple users by mapping them to their virtual address spaces on demand. This is possible, since the post queues are used synchronously to program execution, only when a process explicitly posts a data transfer request. At these points, the post operation results in a page fault if the post queue is not mapped and the system takes control to perform the necessary mapping actions. For the numbers of users that need to be supported on such systems, the rest of the user context information can be maintained on the NIC.

We believe that the common case performance impact of this mechanism in the environments we are considering is very low. In scalable servers that provide services to multiprogrammed workloads. The common case involves a small number of user processes. Since most NICs today can easily support at least a few –ten or so– post queues, we allow each user process to have its own post queue, as is most realistic in current setups, and due to space limitations we do not evaluate the performance impact of this specific extension.

3.6 Implementation Issues

In this section we describe some important issues about the implementation platform we use.

The NCP runs on Myrinet LANai9 processor [6] and is completely operating system independent. It is written in C++ and compiled with gcc cross compiler to generate LANai9 code. The generated code is then downloaded to the NIC memory and then the LANai9 processor starts executing from a predefined address.

The NCP implements a set of state machines for the send and receive path. For concurrency reasons, the state machines are heavily asynchronous, consisting of short-lived states. The NIC has little support for debugging. The most useful debugging

mechanism is a slow print facility through host interrupts and a trace facility in the driver to track the NCP execution points. There is no way to run the code with breakpoints or in single-step mode, or to attach to the running program and see the current state of the NIC memory.

The user library is mostly operating system independent. It is implemented in C and C++. It provides wrap functions for the driver system calls and calls to directly manipulate the queues on the NIC memory. Moreover, it creates the notification handler thread and the miss event handler thread at the start of the user program.

The device driver code is written in C and C++, and is structured to be highly portable. It is divided in two parts: (i) a thin, kernel-dependent layer that initializes the driver and wraps the calls to the operating system internal functions (mostly hardware manipulation functions) and (ii) an operating system independent part that implements the actual functionality of the device driver. Although in this project we use Linux (kernel version 2.2.16) as the operating system, the device driver works for both Linux and WindowsNT.

Chapter 4

Results

The goal of our performance analysis is twofold: First, we would like to tune the lookup cache design parameters in our extensions so that we both reduce the memory requirements as well as impose little additional overhead to real applications. Second, we are interested in gaining insight on how the system behaves as the parameters vary within a wide range of values.

The experimental system we use for evaluation is a cluster of four 2-way PentiumIII nodes. The exact configuration of each node is shown in Table 4.1. For the real applications, we use the largest problem set size permitted by the 2GBytes address space of the Linux operating system. Using large cluster configurations result in less stress for the NIC memory system, since the application working set is divided among more nodes, and therefore the communication working set of each node is smaller. Taking into account these factors we run each application with up to 8 processors.

The nodes in the cluster are interconnected with a Myrinet network [6]. All system nodes are connected with a 16-port full crossbar Myrinet switch. To evaluate the impact of our approach on the performance of the system, we use both synthetic micro-benchmarks as well as real applications. We use micro-benchmarks to provide basic measurements in uncontended conditions and to independently stress specific components of

the system.

Processors	2 x Intel Pentium III, 800 MHz
Cache	32K (L1), 512K (L2)
Memory	512MB SDRAM
Operating System	RedHat Linux Kernel 2.2.16-3smp
PCI buses	32 bits, 33 MHz, 133MBytes/s
NIC	Myricom M3M-PCI64B
NIC CPU	LANai9, 133 MHz
Network Link	Bi-directional, 160MBytes/s each direction

Table 4.1: Cluster node configuration.

VMMC Operation	Overhead
1-word send (one-way latency)	$8\mu s$
1-word fetch (round-trip latency)	$22\mu s$
4 KByte send (one-way latency)	$52\mu s$
4 KByte fetch (round-trip latency)	$81\mu s$
Maximum ping-pong bandwidth	118 MBytes/s
Maximum fetch bandwidth	118 MBytes/s
Notification	$18\mu s$

Table 4.2: Basic VMMC costs.

The applications we use are a subset of the SPLASH-2 benchmarks [27] on top of a shared virtual memory (SVM) system that provides the illusion of a single system image. The specific applications we use are FFT, WaterNsquared, WaterSpatial, LUContiguous, Ocean, Volrend, Radix, and Raytrace. This set of applications cover a wide range of application features. The SVM protocol we use is GeNIMA [4, 18], which is a home-

based, page-level SVM protocol. GeNIMA has been optimized for use with modern system area networks that support remote RDMA operations.

Table 4.2 shows the cost of basic VMMC operations on our cluster. VMMC provides a one way, end-to-end latency of around $8\mu s$, which is among the best performing systems using a Myrinet interconnect.

Since the miss penalty depends on the retransmission timer interval used in the NCP, we adjust it to a small value, based on the work in [23]. The retransmission interval is the time that the requesting node waits for acknowledgments for transmitted packets before retransmitting each packet. Our experiments show that a retransmission interval of 200-300 μs results in optimal system behavior. Furthermore, we verify that each request that misses involves on average about one retransmissions. Although, each miss takes less than this amount of time to service, setting the retransmission timer to a lower value interferes with normal system behavior [23].

We present results both for the VMH as well as the CBH lookup mechanisms. However, we place more weight on the VMH lookup mechanism, since this is more critical to system performance. We present statistics on both miss rates as well as application speedups. We divide cache misses based on two factors: (i) whether they occur in the send or the receive path and (ii) whether they are cold misses or capacity and conflict (non-cold) [14]. The first factor provides information about the send and receive communication working sets in each application whereas the second helps us understand better the impact of each cache parameter.

4.1 VMH Lookup

First we explore the configuration parameter space for the VMH lookup cache.

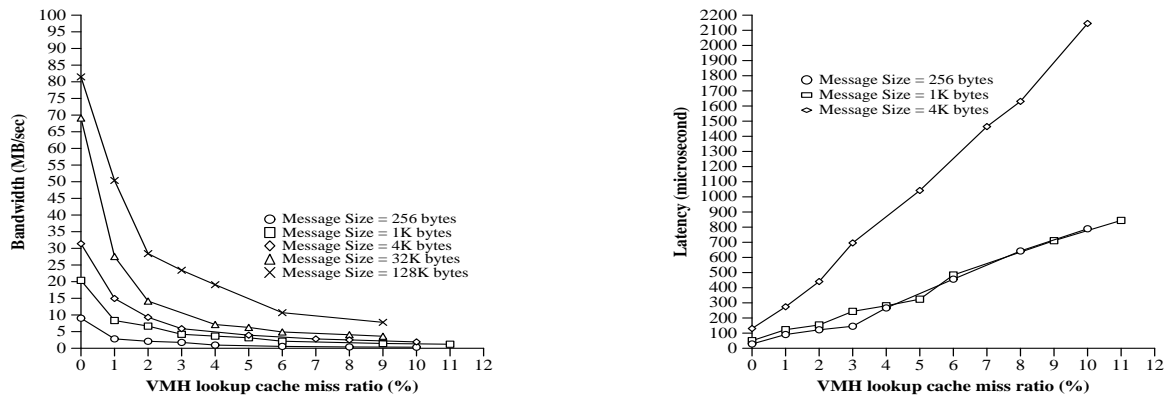


Figure 4.1: The effect of VMH miss on ping-pong latency and bandwidth.

4.1.1 Micro-benchmark tests

Figure 4.1 use a micro-benchmark to show how latency and bandwidth vary as we change the VMH lookup cache miss ratio. The micro-benchmark we use is essentially a ping-pong test, where requests miss on one of the two nodes only. The micro-benchmark is able to control the miss rate by using specific source and destination buffers for communication. We see that the bandwidth degrades rapidly and the latency increases linearly as the miss ratio increases. In both cases the negative effect of a miss is higher for larger message sizes. This is due to the fact that the cost of dropping and retransmitting on a miss increases with message size, due to the lack of negative acknowledgment and receive side buffering.

4.1.2 Real Applications

Next, we examine the SPLASH2 applications. For each application we choose a fairly large problem size, as shown in Table 4.3 to stress the VMH lookup cache. There are two reasons why we cannot choose larger problem sizes: (i) Due to the design of the SVM system we use, the total size of data that is globally shared cannot be more than half the process address space. In Linux, each process can use up to 2GBytes of the virtual address

space on 32-bit processors. (ii) Some applications execution time increase rapidly with problem size and since we are interested in examining the behavior of each application in many different configurations, it is not practical to choose such problem sizes.

Application	Input Size
WaterSpatial	4096 Mols
WaterNsquared	4096 Mols
Raytrace	1024x1024 car
Radix	8M Keys
Ocean	514x514
LUContiguous	2048x2048
Volrend	CST head
FFT	4M Complex Numbers

Table 4.3: The problem size for SPLASH-2 benchmark applications.

We vary each of the three VMH lookup cache design parameters, cache size (C), line size (L), and associativity (A), with a wide range, as shown in Table 4.4.

Parameter	Values
Cache Size (C)	8K, 16K, 32K entries
Cache Line (L)	8, 16, 64, 128 entries
Associativity (A)	2, 4, 8 lines

Table 4.4: The range of values for each of the VMH lookup cache parameters.

We first examine the impact of each cache parameter on the cache miss ratio. Figure 4.2 shows the VMH lookup cache miss ratio breakdown for each set of parameters. The first observation is that the number of misses varies greatly, between about 0-40%,

among different configurations. Overall, the miss ratio of WaterNsquared, WaterSpatial, Ocean, Raytrace, and Volrend is within the range 0-2% range, whereas for FFT, LUContiguous, and Radix the miss ratio is much larger and in the 2-40% range. Second, we observe that all parameters have a significant effect on system miss rates, as explained next.

Increasing the cache line size results in very effective prefetching and reduces the number of misses for most applications even when the cache line size is increased up to 128 entries. However, when increasing the cache line size from 16 to 64 and then to 128 entries, Ocean exhibits a large number of conflicts with non-cold misses more than doubling when the cache size is less than 32K entries. Overall our results suggest that a line size of 64 works well for most applications we examine.

Cache size has a small effect on the number of misses at small line sizes, but the importance of the cache size increases for larger line sizes. Doubling the cache size from 16 to 32K entries more than halves the total number of misses for FFT and LUContiguous in the L128 configurations (Figure 4.2). In general, our results show that a 16K-entries cache is adequate for most cases we examine.

Finally, cache associativity is important mostly at smaller cache and line sizes. Increasing associativity from 2 to 4 has a positive effect on most applications. Increasing the associativity from 4 to 8 seems to have a smaller effect. Furthermore, since the VMH lookup cache is implemented in software, higher associativities result in higher lookup times (e.g. for Radix C8.L128, Ocean C32.L128, and WaterSpatial C8.L64, and C8.L128 Figure 4.2). For these reasons, an associativity of 4 seems to be the best compromise between reducing the number of misses and performance impact.

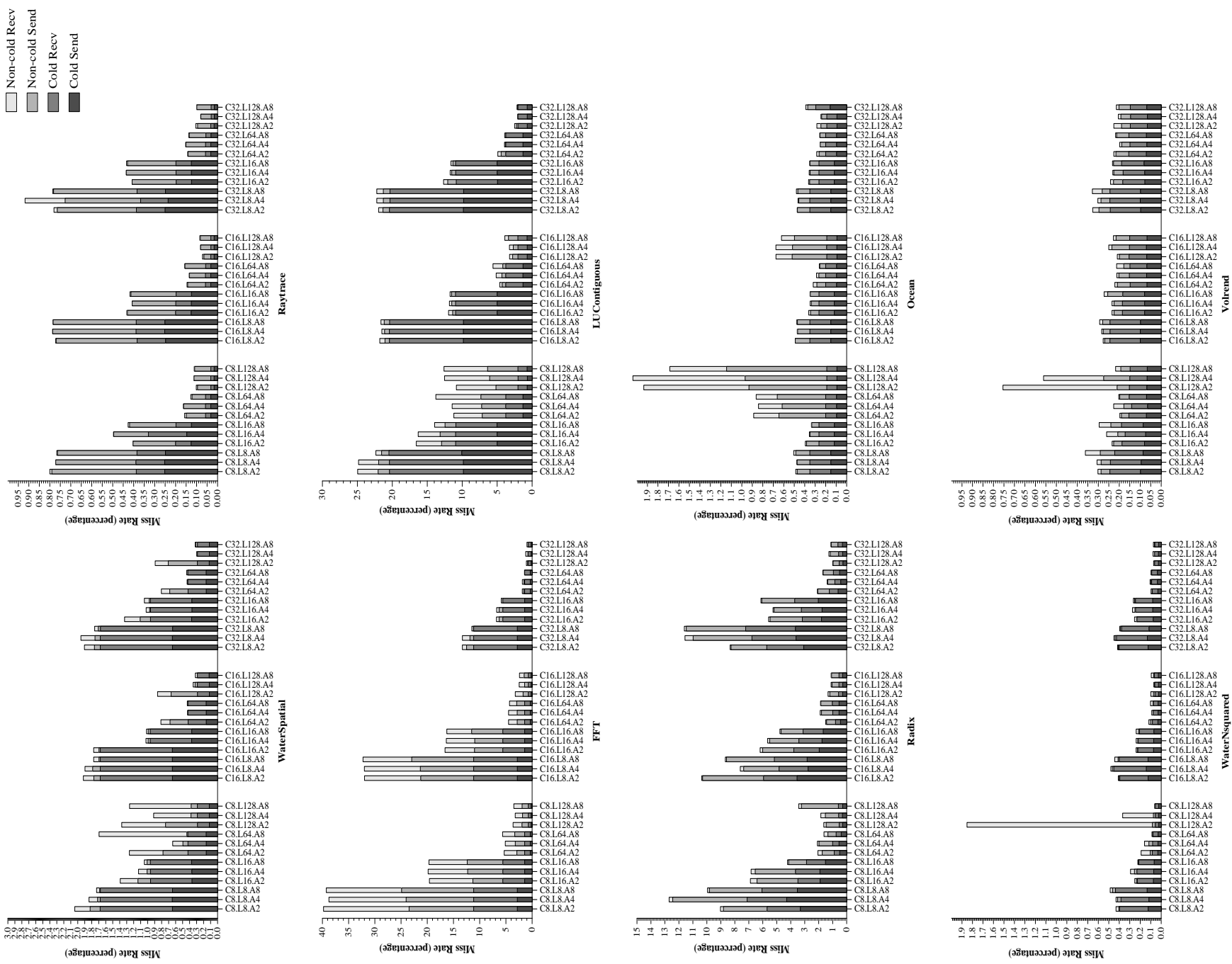


Figure 4.2: VMH lookup cache miss breakdown of the SPLASH-2 applications.

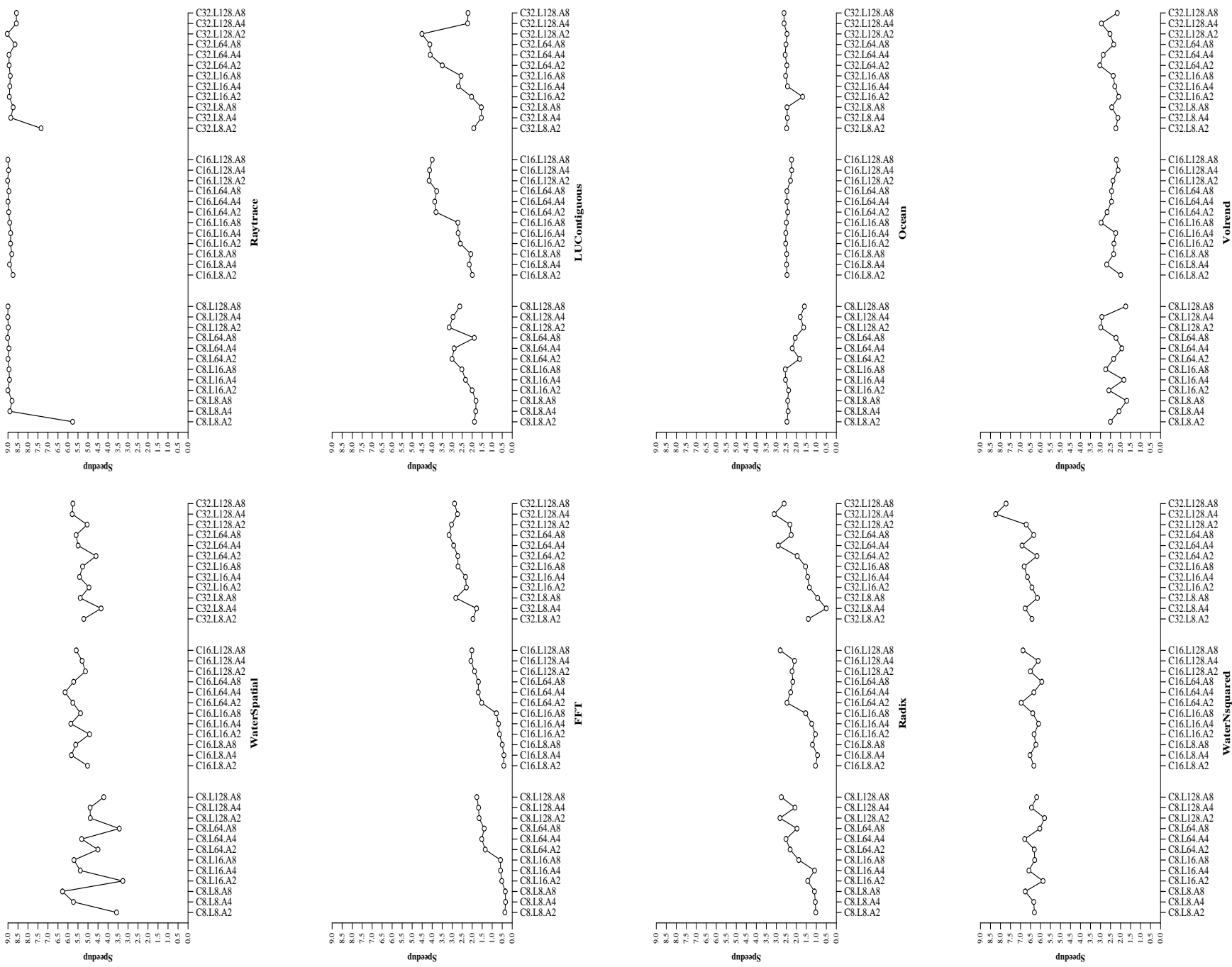


Figure 4.3: The speedup of the SPLASH-2 applications in different VMH lookup cache configurations.

We also look at the impact of cache configuration on the application speedups. Figure 4.3 shows the application speedups for each system configuration. We see that there is a close correlation between the VMH lookup cache miss rates and application speedups, even for the configurations with small miss ratio. Generally we can divide applications in two categories. In the first category belong WaterNsquared, WaterSpatial, Volrend, Raytrace, and Ocean that, except for few configurations, are not very sensitive to the cache configuration due to the small miss rates they exhibit. Overall, for these applications, a small VMH lookup cache of 8K entries results in similar performance to the static system configuration, where the full VMH table is maintained on NIC memory. The second category includes FFT, Radix, and LUContiguous that show significant variations in speedups with cache configurations. For these applications the cache line size is the most important parameter, resulting in more than 100% in speedup variation.

Application	VMMC	<i>miNI</i>	Slowdown %
WaterSpatial	3182	3032	-4.7
WaterNsquared	20547	20747	0.9
FFT	2010	2085	3.7
Ocean	2842	2930	3.4
Radix	1535	1934	26.0
Raytrace	18960	18039	-4.8
LUContiguous	13209	7755	-41.1
Volrend	-	3306	-

Table 4.5: The execution time (in ms) of each application in the original VMMC system versus *miNI* for the best VMH and CBH cache configurations.

Table 4.5 shows the execution time of the applications in the original VMMC system versus their best performance in *miNI*. We observe that except for Radix, the overhead

of *miNI* is negligible. For some applications the new system performs slightly better. We have not investigated this effect completely yet, but we believe that the changes in the common path due to the imported CBH table elimination result in somewhat lower system overheads for cases that do not exhibit many VMH and CBH misses¹.

Based on our results, we conclude that a configuration of (C16, L64, A4) results in the best tradeoff between performance and NIC memory requirements. This configuration uses about 75 KBytes of NIC memory, which is a substantial reduction from the original configuration, which uses more than 256 KBytes for the full VMH lookup table. Moreover, the current configuration does not have a limit on the amount of host memory it can support, although performance tradeoffs may change as application working set sizes increase.

Application	0– 10 μ s	10– 20 μ s	20– 50 μ s	50– 100 μ s	100– 200 μ s	200– 500 μ s	>500 μ s
WaterNsquared	21.43	64.29	12.50	0.89	0.00	0.00	0.89
WaterSpatial	41.97	55.74	0.98	0.66	0.33	0.00	0.33
FFT	45.43	52.07	2.03	0.05	0.02	0.08	0.33
Radix	23.40	59.75	14.89	0.53	0.00	0.18	1.24
LUContiguous	27.09	58.29	11.48	0.49	0.69	0.98	0.98
Ocean	20.82	50.41	24.93	2.74	0.00	0.27	0.82
Volrend	20.75	54.72	18.87	1.89	1.89	1.89	0.00
Raytrace	30.69	60.29	5.42	2.53	0.36	0.36	0.36

Table 4.6: The miss delay time distribution for each application. Numbers represent percentages of total misses.

Another important metric in the system is the miss penalty, which we define as the

¹This explanation does not include the significant improvement in the execution time of LUContiguous. We are currently investigating this effect.

Application	0– 10 μ s	10– 20 μ s	20– 50 μ s	50– 100 μ s	100– 200 μ s	200– 500 μ s	>500 μ s
WaterNsqared	0.00	15.22	2.90	15.22	6.52	54.35	5.80
WaterSpatial	0.00	1.27	15.87	39.05	0.32	39.37	4.13
FFT	0.09	66.03	5.63	19.79	0.24	8.16	0.06
Radix	1.62	41.49	6.30	16.74	6.57	26.91	0.36
LUContiguous	0.54	22.59	6.93	40.77	1.53	25.83	1.80
Ocean	0.78	25.83	7.38	20.39	5.44	39.03	1.17
Volrend	2.35	28.24	1.18	7.06	5.88	45.88	9.41
Raytrace	3.83	50.08	15.16	6.13	0.15	23.12	1.53

Table 4.7: The miss processing time distribution for the applications. Numbers represent percentages of total number of misses.

time between the occurrence of the miss in the NIC and the time when the device driver services the miss request. We divide this penalty into two parts. The first part, *miss delay time*, is the time between the occurrence of the miss interrupt up to the point when the miss handler in the driver is scheduled by the operating system. The second part, *miss processing time*, is the time to service the miss in the device driver and to update the VMH cache. While both of them depend on various system parameters, the miss delay time is more unpredictable since it depends on the operating system scheduling delay.

Our results show that the distribution of the miss delay time and the miss processing time are independent from the VMH lookup cache configuration. However, they both vary across different applications. Tables 4.6 and 4.7 show the miss delay and processing time distributions respectively for each application. Overall, the miss processing time incurs high variations, within the 0-500 μ s range. This is justified by the fact that servicing a miss includes expensive operating system memory management functions and the fact

that the miss handler has to compete with the application threads for CPU cycles. The delay miss time incurs lower variations and is usually within the 0-50 μ s range. Misses outside this range are due to the fact that we use a kernel thread to service the miss requests, which needs to be scheduled by the operating system. Since kernel threads are scheduled as regular threads, there is no guarantee that this thread will run within a specific interval, resulting in large variations in the miss delay times.

4.2 CBH Lookup

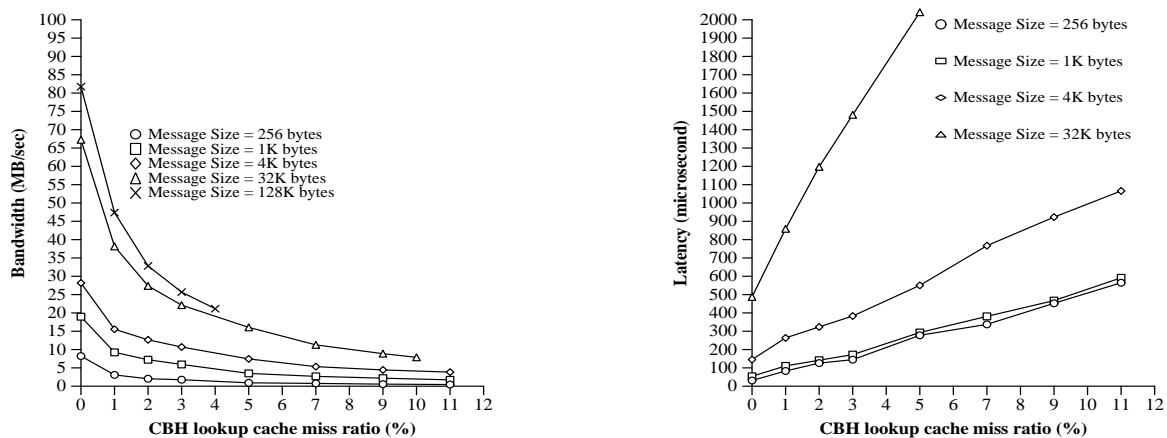


Figure 4.4: The effect of CBH miss on ping-pong latency and bandwidth.

Since our higher system layers have been optimized to use a small number of communication regions, exactly due to the fact that modern NICs impose limitations, it is difficult to use real applications to examine the impact of the CBH cache parameters on performance. Instead, we use a micro-benchmark similar to the one used for our VMH lookup cache evaluation to examine the impact of the miss ratio on basic ping-pong latency and bandwidth. In order to isolate the impact of CBH lookup cache misses, we configure VMH lookup cache to incur a negligible number of misses. Figures 4.4 show that the effect of CBH lookup cache miss both on bandwidth and latency is very similar

to the effect of VMH lookup cache misses. For larger message size the penalty of dropping packets and retransmission is higher, which results in higher performance penalty per cache miss. However, a small CBH lookup cache is adequate to eliminate most cache misses, alleviating the effort of the high miss penalty. More importantly, by implementing dynamic CBH management we are able to reduce the amount of memory required on the NIC to support the problem sizes we examine from 163 KBytes to 25 KBytes without any noticeable impact on performance.

4.3 Summary

Overall, our system extensions for dynamically managing handle lookup on the NIC result in significant reductions in memory requirements from 524 KBytes to about 132 KBytes with a small impact on system performance for the configurations we examine. Furthermore, our extensions eliminate any NIC-imposed restrictions on how much host memory can be used for communication buffers, an important problem in modern storage, database, and application servers.

Chapter 5

Conclusions and Future Work

To improve communication performance, modern interconnects used in scalable servers as well as storage systems, provide user applications the ability to directly transfer data between application virtual address spaces in different nodes. To support such data transfer operations, network interfaces need to translate between virtual and physical addresses as well as between other types of user and system handles. In most current user-level communication systems, the mechanisms used for address translation are static and use data structures maintained on NIC memory. With increasing host memory sizes, this approach requires the use of large on-NIC memories to maintain various mappings. It also imposes limitations on the amount of host memory that can be used for communication buffers.

In this work we deal with these shortcomings by extending a state-of-the-art communication layer to perform all necessary translations dynamically and use the NIC memory only as a cache for the larger host memory. We implemented this approach in the programmable Myrinet network interface and measure its impact on system performance with both micro-benchmarks and real applications.

5.1 Summary of the Results

Overall, our system extensions for dynamically managing handler lookup on the NIC results in significant reductions in memory requirements from 524 KBytes to 132 KBytes with a small impact on system performance for the configurations we examine. Furthermore, our extensions eliminate any NIC imposed restrictions on how much host memory can be used for communication buffers, an important problem in modern storage, database, and application servers.

We find that the communication working sets of realistic applications are in many cases small so that small on-NIC lookup caches have almost no impact on performance. For the others, we show that the cache configuration parameters have major impact both on the number of cache misses and the overall system performance. We find that there are considerable spatial locality in the communication access pattern of most applications. Therefore, prefetching cache entries by using large cache lines is effective and overshadows the increased miss penalty costs.

Finally the mechanisms we use in our approach are designed to avoid unnecessary complexity in system design. This makes our approach appropriate for incorporating in network interfaces that use general purpose processors with firmware, as well as more aggressive hardware implementations.

5.2 Future Directions

Although we design, implement, and evaluate system extensions to address memory issues on modern NICs, it is still important to better explore the configuration space and understand the implications of various configurations on system performance. More specifically, future directions of this work may include the following:

- Investigation of alternative methods for cache miss handling.

- To propose solutions for alternative protection models in user-level communication systems.
- More general evaluation of our approach including including (i) multi-programmed workloads (ii) larger cluster configurations (iii) wider classes of applications, in particular client-server applications (iv) larger application problem sizes.

To summarize, our results show that techniques similar to our approach can be very useful in building the next-generation NICs for modern application, database, and storage servers.

Bibliography

- [1] M. Banikazemi, B. Abali, and D. Panda. Comparison and evaluation of design choices for implementing the virtual interface architecture (via). In *Workshop on Communication and Architectural Support for Network-based Parallel Computing CAPCN-HPCA*, 2000.
- [2] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP15)*, December 1995.
- [3] A. Basu, M. Welsh, and T. von Eicken. Incorporating memory management into user-level network interfaces. <http://www2.cs.cornell.edu/U-Net/papers/unetmm.pdf>, 1996.
- [4] A. Bilas, C. Liao, and J. P. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proc. of the 26th International Symposium on Computer Architecture (ISCA26)*, May 1999.
- [5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A virtual memory mapped network interface for the shrimp multicomputer. In *Proc. of the 21st International Symposium on Computer Architecture (ISCA21)*, pages 142–153, Apr. 1994.
- [6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.
- [7] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the virtual interface architecture. In *Supercomputing (SC)*, pages 7–13, 1998.

- [8] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li. UTLB: A mechanism for address translation on network interfaces. In *Proc. of The 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS8)*, pages 193–203, San Jose, CA, Oct. 1998.
- [9] DAFS Collaborative. *DAFS: Direct Access File System Protocol Version: 1.00*, Sept. 2001.
- [10] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: efficient support for reliable, connection-oriented communication. In *Proc. of The 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V)*, Aug. 1997. A short version of this appears in *IEEE Micro*, Jan/Feb, 1998.
- [11] D. Dunning and G. Regnier. The Virtual Interface Architecture. In *Proc. of The 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V)*, Aug. 1997.
- [12] Gigaset. Gigaset cLAN family of products. <http://www.emulex.com/products.html>, 2001.
- [13] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proc. of the IEEE Spring COMPCON '96*, Feb. 1996.
- [14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.
- [15] I. E. T. F. (IETF). iSCSI, version 08. In *IP Storage (IPS), Internet Draft, Document: draft-ietf-ips-iscsi-08.txt*, Sept. 2001.
- [16] InfiniBand Trade Association. Infiniband architecture specification, version 1.0. <http://www.infinibandta.org>, Oct. 2000.
- [17] P. Jamieson and A. Bilas. Cables: Thread control and memory management extensions for shared virtual memory clusters. In *Proc. of The 8th IEEE Symposium on High-Performance Computer Architecture (HPCA8)*, Feb. 2002.
- [18] D. Jiang, B. Cokelley, X. Yu, A. Bilas, and J. P. Singh. Application scaling under shared virtual memory on a cluster of smps. In *Proc. of the 13th ACM International Conference on Supercomputing (ICS99)*, pages 165–174, June 1999.

- [19] A. M. Mainwaring and D. E. Culler. Design challenges of virtual networks: Fast, general-purpose communication. In *Proc. of The 1999 ACM Symposium on Principles and Practice of Parallel Programming (PPOPP99)*, pages 119–130, May 1999.
- [20] National Energy Research Scientific Computing Center. M-via: A high performance modular via for linux. <http://www.nersc.gov/research/FTG/via>, 1998.
- [21] S. Pakin, M. Buchanan, M. Lauria, and A. Chien. The Fast Messages (FM) 2.0 streaming interface. Usenix'97, 1996.
- [22] I. Schoinas and M. D. Hill. Address translation mechanisms in network interfaces. In *Proc. of The 4th IEEE Symposium on High-Performance Computer Architecture (HPCA4)*, 1998.
- [23] J. Tang and A. Bilas. Tolerating network failures in system area networks. In *Proc. of the 2002 International Conference on Parallel Processing (ICPP02)*, Aug. 2002.
- [24] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato. PM: a operating system coordinated high performance communication library. High-Performance Computing and Networking '97, April 1997.
- [25] S. G. Thorsten von Eicken, David Culler and K. Schausser. Active messages: a mechanism for integrating communication and computation. In *Proc. of the 19th International Symposium on Computer Architecture (ISCA19)*, pages 256–266, May 1992.
- [26] J. Wilkes. Hamlyn – an interface for sender-based communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, Nov. 1993.
- [27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd International Symposium on Computer Architecture (ISCA22)*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.
- [28] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. Philbin, and K. Li. Experiences with vi communication for database storage. In *Proc. of the 29th International Symposium on Computer Architecture (ISCA29)*, May 2002.