Design and Implementation of High Performance Communication Subsystems for Clusters

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Mohammad Banikazemi, M.S.

* * * * *

The Ohio State University

2000

Dissertation Committee:

Prof. Dhabaleswar K. Panda, Adviser

Prof. Ponnuswamy Sadayappan

Prof. Mario Lauria

Approved by

Adviser

Department of Computer and Information Science

© Copyright by
Mohammad Banikazemi
2000

ABSTRACT

With the significant increase in computing power of processors and tremendous improvement in the performance of networking hardware, clusters have become a popular platform for high performance computing. In order to make the performance of clusters comparable to that of traditional high performance computing systems, it is crucial to make the communication subsystems of these systems as efficient as possible. In recent years, communication subsystems with user-level protocols have been proposed by the research community and industry to address this issue. All of these communication systems use much simpler communication protocols in comparison with legacy protocols such as the TCP/IP. The role of the operating system has been much reduced in these systems and in most cases user applications are given direct access to the network interface. The Virtual Interface Architecture (VIA) specification has been developed to standardize these user-level protocols and to make their ideas available in commercial systems. The primary objective of this research is to design and implement efficient and high performance communication subsystems for clusters with user-level protocols such that the high performance of the networking technologies is passed to applications. To achieve this goal, this thesis is focused on five components of communication subsystems: network interface support, communication mechanism, distribute shared memory (DSM) support, distributed memory support, and performance evaluation. Several design choices for various components of VIA are proposed and evaluated on different platforms. A prototype implementation of VIA is developed for IBM SP-connected clusters. This implementation remains to be the most efficient software implementation of VIA to date. The performance of this implementation is extensively evaluated and performance bottlenecks have been identified. Furthermore, several hardware enhancements for improving the performance are studied. Design and implementation of the communication infrastructure required for supporting distributed shared memory and distributed memory programming models on top of user-level communication protocols are also studied. The proposed communication mechanisms and their extensive evaluation demonstrate significant potential to be applied to the design of communication subsystems for current and future clusters.

Dedicated to my mother Parvin and my father Hossein

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. D. K. Panda for his invaluable guidance through the course of my graduate study. I am grateful to him for the tremendous time, effort, and wisdom he invested in guiding me and steering my research. More importantly I would like to express my gratitude toward his friendship and understanding during the tough periods of my stay at Ohio State University.

I would like to thank Prof. P. Sadayappan for his support and help as a member of our research group. I thank him for the discussions we had about my research and for his insightful comments. I also thank Prof. M. Lauria for being in my dissertation committee and for his valuable suggestions.

I am grateful to Dr. Bulent Abali of IBM T. J. Watson Research Center and Dr. Rama Govindaraju of IBM for their guidance and friendship both during my internships and afterwards. I am also thankful to Dr. Govindaraju for nominating me for an IBM graduate fellowship award.

I gratefully acknowledge the financial support provided by the Department of Computer and Information Science, National Science Foundation (NSF), an IBM Graduate Fellowship, and a Presidential Fellowship.

I thank Jiuxing Liu, Sencer Kutlug, and Arun Ramakrishnan for their help in running experiments related to Chapters 4 and 5 of this dissertation. It would have been impossible for me to complete this work without their valuable help.

Special thanks go to Darius Buntinas with whom I have spent many hours discussing technical and not so technical topics. I cannot forget his willingness to extend his help under any conditions and at any time. I value his friendship tremendously.

I would also like to thank previous members of the PAC group. In particular, I would like to thank Rajeev Sivaram, Ram Kesavan, Donglai Dai, and Matt Jacunski for hours of discussions on various topics and for their friendship.

I would like to thank the great friends I have made during the course of my stay at Ohio State: Ashkan, Susana, Manfredi, and Neda. Their friendship alone would have made my stay at Ohio State worth while.

Finally, I thank my family, my parents and sister, for their immeasurable love, encouragement, and support. I feel very happy to share with them, the joy of writing this thesis.

VITA

1967	Born - Kashan, Iran
1989	B.S., Electrical Engineering, Isfahan University of Technology, Isfahan, Iran
1989 - 1994	Software Engineer and System Administrator, Tehran, Iran
Winter 1995 - Summer 1996	Graduate Teaching Associate, Computer and Info. Science and Electrical Engineering Departments, The Ohio State University.
1996	M.S., Electrical Engineering, Ohio State University
Fall 1996 - Spring 1997	Graduate Teaching Associate, Dept. of Computer and Info. Science, The Ohio State University.
Fall 1997 - Spring 1998	Graduate Research Associate, The Ohio State University.
Summer 98	Summer Intern, Power Parallel Group, IBM Corporation.
Fall 1998 - Spring 1999	Graduate Research Associate, The Ohio State University.
Summer 1999	Summer Intern, Research Division, IBM Corporation.
Fall 1999 - Spring 2000	IBM Graduate Fellow, Ohio State University.
Summer 2000 - Fall 2000	OSU Presidential Fellow, Ohio State University.

PUBLICATIONS

Research Publications

M. Banikazemi, B. Abali, L. Herger, and D. K. Panda. "Design Alternatives for VIA and an Implementation on IBM Netfinity NT Cluster." *Special issue of Journal of Parallel and Distributed Computing on Cluster and Network-Based Computing,* to appear.

B. Abali, C. B. Stunkel, J. Herring, M. Banikazemi, D. K. Panda, C. Aykanat, and Y. Aydogan. "Adaptive Routing on the New Switch Chip for IBM SP Systems." *Special issue of Journal of Parallel and Distributed Computing on Routing in Computer and Communication Networks,* to appear.

M. Banikazemi, D. K. Panda, and P. Sadayappan. "Implementing TreadMarks on Virtual Interface Architecture (VIA): Design Issues and Alternatives." *Ninth Workshop on Scalable Shared Memory Multiprocessors, held in conjunction with International Symposium on Computer Architecture (ISCA'2000),* June 2000.

M. Banikazemi, V. Moorthy, L. Hereger, D. K. Panda, and B. Abali. "Efficient Virtual Interface Architecture Support for IBM SP Switch-Connected NT Clusters." *International Parallel and Distributed Processing Symposium (IPDPS'2000),* pp. 33-42, May 2000.

M. Banikazemi, C. Stunkel, D. K. Panda, and B. Abali. "Adaptive Routing in RS/6000 SP-like Bidirectional Multistage Interconnection Networks." *International Parallel and Distributed Processing Symposium (IPDPS'2000),* pp. 43-52, May 2000.

M. Banikazemi, B. Abali, and D. K. Panda. "Comparison and Evaluation of Design Choices for Implementing the Virtual Interface Architecture (VIA)." *Workshop on Communication and Architectural Support for Network-based Parallel Computing (CANPC-HPCA' 2000).*

M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda. "Implementing Efficient MPI on LAPI for the IBM-SP: Experiences and Performance Evaluation." *International Parallel Processing Symposium (IPPS '99),* pp. 183–190, April 1999.

M. Banikazemi, J. Liu, D. K. Panda, and P. Sadayappan. "Implementing TreadMarks over VIA on Myrinet and Gigabit Ethernet: Challenges, Design Experience, and Performance Evaluation." *Technical Report OSU-CISRC-07/00-TR15.*

M. Banikazemi, J. Liu, S. Kutlug, A. Ramakrishnan, P. Sadayappan, H. Shah, and D. K. Panda. "VIBe: A Microbenchmark Suite for Evaluating Virtual Interface Architecture (VIA) Implementations." *Technical report OSU-CISRC-10/00-TR20.*

M. Banikazemi and D. K. Panda. "Can Scatter Communication Take Advantage of Multidestination Message Passing?." *Int'l Symposium on High Performance Computing (HiPC '00),* to be presented.

M. Banikazemi, J. Sampathkumar, S. Prabhu, D. K. Panda, and P. Sadayappan. "Communication Modeling of Heterogeneous Networks of Workstations for Performance Characterization of Collective Operations." *IPPS Heterogeneous Computing Workshop (HCW '99),*pp. 125–133, April 1999.

M. Banikazemi, Vijay Moorthy, and D. K. Panda. "Efficient Collective Communication on Heterogeneous Networks of Workstations." *International Conference for Parallel Processing (ICPP'98),* pp. 460-467, Aug. 1998.

D. K. Panda, D. Basak, D. Dai, R. Kesavan, R. Sivaram, M. Banikazemi and V. Moorthy. "Simulation of Modern Parallel Systems: A CSIM-Based Approach." *1997 Winter Simulation Conference (WSC '97),* pp. 1013-1020, Dec. 1997.

D. Basak, D. K. Panda, and M. Banikazemi. "Benefits of Processor Clustering in Designing Large Parallel Systems: When and How?" *International Parallel Processing Symposium (IPPS '96),* pp. 286-290, Apr. 1996.

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

Computer Architecture	Prof. Dhabaleswar K. Panda
Networking	Prof. Raj Jain
Operating Systems	Prof. Thomas W. Page Jr.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Network-Based Computing

Network-Based Computing (NBC) is becoming increasingly popular for providing cost-effective and affordable parallel computing for day-to-day computational needs [10, 31, 52]. Such environments consist of *clusters* of workstations connected by *System, Local,* or *Wide Area Networks* (SANs, LANs, or WANs). Figure 1.1 illustrates such an environment. The major advantage of the NBC environments over traditional high performance computing platforms (such as massively parallel processors (MPPs)) is their lower cost. Although the cost of a group of workstations connected to each other through available networking technologies has been always much less than the cost of an MPP with the same number of nodes, the major obstacles in the usage of clusters as high performance computing platforms has been the limitation of the available networking technologies (in terms of achievable latency and bandwidth). The introduction and mass production of high bandwidth networking technologies such as Myrinet [22], Fast Ethernet, Gigabit Ethernet, FDDI, and ATM [11] has made clusters and Networks of Workstations (NOWs) a viable alternative platform for high performance computing. A wide variety of systems have recently emerged to create a new class of computing platforms for high performance computing. Systems from the Beowulf clusters of Linux PCs interconnected through Ethernet or Myrinet networking technologies to clusters of RS6000 workstations interconnected by SP switches [50, 8] fall into this new class of computing platforms. However, since the new hardware and software networking technologies have not been primarily developed for high performance computing, the communication overhead seen by high performance applications can be too high to make them directly usable for this branch of computing. In order to make the performance of clusters comparable to that of traditional high performance computing systems, it is crucial to make the communication subsystems of these systems as efficient as possible.

A portable programming environment [29] is also key to the success of high performance computing systems. Over the last few years, researchers have developed standard interfaces such as PVM [52, 57] and *Message Passing Interface* (MPI [23, 39, 40]) to provide portability for the parallel programs written in distributed memory programming model. These interfaces and standards do not force an application developer to understand the intricate details of the hardware, software, and network characteristics. However, the performance of applications depends heavily on the latency and bandwidth required for interprocessor communication and synchronization across the nodes as seen by applications developed by using these standards.

In recent years, the MPI standard has become the most popular and widely used standard for developing message passing high performance applications. Many older applications have been

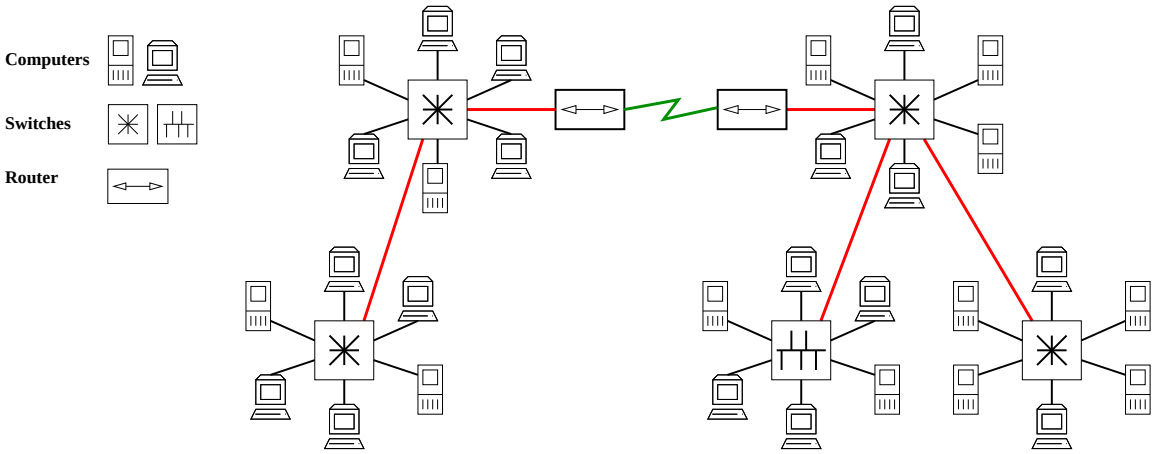

Figure 1.1: A Network-Based Computing environment consisting of different types of computers and networking technologies.

rewritten using the communication primitives as defined by the MPI standard. New applications are mostly being written by using this standard as well. The MPI standard defines a set of functions for point-to-point communications. In the earlier versions of MPI (Versions 1.0 and 1.1), all point-to-point operations were two-sided operations. In these operations, for every send operation there should exist a corresponding receive operation. In the latest version of the MPI standard (Version 2.0), one-sided operations have been provided too. One sided operations (also known as Remote Memory Access operations) extend the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending side and for the receiving side [40]. In addition to point-to-point operations, MPI defines a rich set of collective operations (such as broadcast, barrier synchronization, gather, and reduction). Collective communication is defined as communication that involves a group of processes. Many parallel applications make extensive use of such collective communications.

The distributed shared memory (DSM) programming model [29] is another programming model used in developing parallel applications. In order to support applications developed in this model, the communication subsystem should provide the required facilities to maintain a coherent view of the shared memory by all application processes running across different computing nodes of a cluster. The performance of this type of applications will heavily depend on the performance of the underlying communication subsystems.

In order to achieve an acceptable performance (comparable to the performance of MPPs) in NBC environments, having efficient communication and synchronization services is crucial. Furthermore, the high performance of the communication subsystem should be passed to the applications, if NBC systems are to become a viable choice for high performance computing.

The remaining part of this chapter is organized as follows. Different components of communication subsystems in NBC environments are discussed in Section 1.2. The description of the problem this research aims to solve is presented in Section 1.3. The overview of this thesis is presented in Section 1.4.

1.2 Components of Communication Subsystems in NBC Environments

Designing an efficient communication subsystem requires a comprehensive study of different components of the communication subsystem and the interaction between these components. We divide the communication subsystem into several layers based on the functional and architectural characteristics of the components. Figure 1.2 shows the different components of the communication subsystem in an NBC environment used for high performance computing.

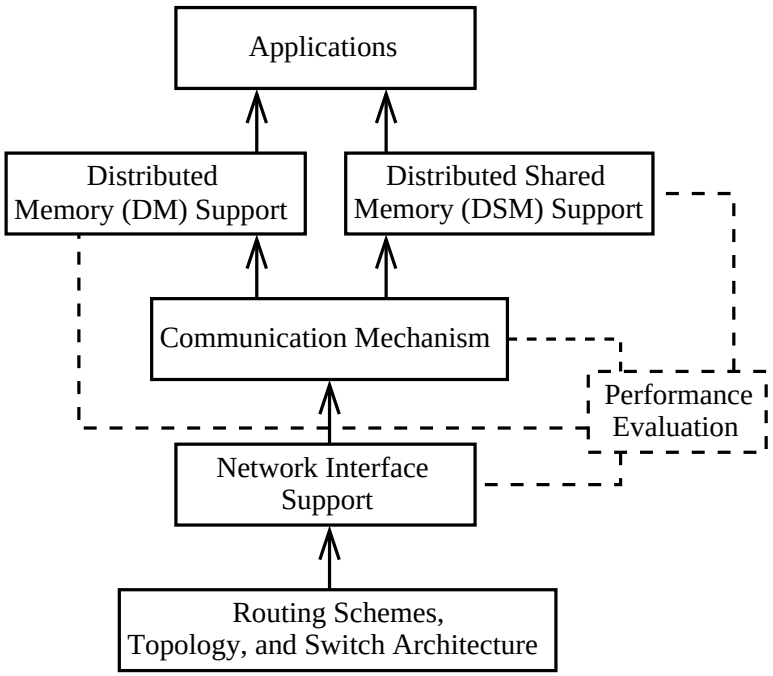

Figure 1.2: The components of the communication subsystem in a network-based computing environment.

The bottom most layer consists of the interconnection topology, switching technology, and routing schemes. As shown in Fig. 1.1, the topology of the interconnection network in an NBC system is typically irregular. In general, such a system can be a WAN consisting of several LANs connected to each other through possibly different networking technologies. Switching elements are responsible for forwarding packets from one link to another one. They perform the forwarding of packets by using techniques such as store-and-forward and cut-through. Given the source and destination of a message, the routing scheme determines the path (links) the message takes. The routing scheme must also provide a mechanism for deadlock avoidance or recovery.

The next layer in the communication subsystem is called network interface support. The network interface at the host typically has a processor, memory and a few DMA engines. DMA engines are used to transfer the packets between the host memory and the interface memory and between the interface memory and the network. Modern network interfaces are programmable and can have a significant role in reducing the communication load of host processors.

The next layer in Fig. 1.2 is called the communication mechanism. This layer is responsible for transmitting the data from a source node to a destination node. This layer is also responsible for adding the required header information to the message. The communication mechanism layer deals also with the fragmentation (packetization) and re-assembly of messages at sending and receiving sides. This layer may enforce the ordering among messages and may guarantee the reliable delivery of data.

The communication mechanism layer is used to implement two components of the next layer: 1) distributed memory support for the programs written in this model by providing communication and synchronization services, and 2) distributed shared memory support for DSM programs by providing the required infrastructure for coherence protocols.

The performance evaluation component as shown in Fig. 1.2 is used for evaluation and tuning of different components of the communication subsystem. It is crucial that the performance of different components and alternative choices for their implementation are evaluated such that the implementation can be tuned for the best performance. Furthermore, the performance evaluation can be used for identifying software and hardware approaches that can be used to eliminate the performance bottlenecks.

1.3 Problem Description

The primary objective of this research is to design and implement efficient and high performance communication subsystems for Network-Based Computing environments with SAN/LAN technologies which are also known as Clusters such that the high performance of the networking technologies is passed to applications. To achieve this goal, this thesis is focused on five major components of communication subsystems (depicted in Fig. 1.2): network interface support, communication mechanism, DSM support, distributed memory support, and performance evaluation components. In the rest of this section, we explain the challenges involved in achieving such a goal and present the specific problems this research addresses.

Raw bandwidth of networks have increased significantly in the past few years and networking hardware supporting bandwidths in the order of gigabits per second have become widely available. However, the traditional networking architectures and protocols do not reach the performance of the hardware at the application level. The layered nature of the legacy networking softwares and the usage of expensive system calls and extra memory–to–memory copies required in these systems are some of the factors responsible for degradation of the communication subsystem performance as seen by the applications. In recent years, *user-level* communication subsystems [29] such as AM [55], VMMC [21], FM [42], U-Net [54, 56], LAPI [45], and BIP [43] have been proposed by the research community and industry to address these issue. All of these communication systems use much simpler communication protocols in comparison with legacy protocols such as the TCP/IP. The role of the operating system has been much reduced in these systems and in most cases user applications are given direct access to the network interface.

The Virtual Interface Architecture (VIA) specification has been developed to standardize these user-level network interfaces and to make their ideas available in commercial systems [7]. However, the flexibility of the VIA specification has left several choices viable for the implementation of various components of VIA. These design choices should be identified and evaluated. The impact of network interface support on the performance of these components and the overall performance of the communication subsystem needs to be investigated. Whether a framework for evaluating different implementations of VIA can be developed is an important question which should be addressed.

The main goal of providing high performance (low latency and high bandwidth) communication subsystems is to make such a performance available at the user application level. If only a small fraction of the low latency and high bandwidth provided by the communication subsystem (such as VIA and LAPI) becomes available at the application level, the main purpose of using efficient communication subsystems has been practically defeated. As mentioned earlier, distributed memory and DSM are the two major programming paradigms used for high performance computing. Therefore, it is crucial to provide the performance of the underlying communication subsystems to the applications written in either of these programming models.

Specifically, this thesis looks at the following problems:

1. How different components of VIA can be designed and developed with respect to modern network interface cards?
2. How these components can be used to build VIA in the most efficient manner?
3. Can a set of micro-benchmarks be used for evaluating different implementations of VIA?
4. How DSM applications can take advantage of high performance user-level communication protocols such as VIA?
5. How the high performance and efficiency of user-level communication protocols such as LAPI can be passed to distributed memory applications?

Let us look at these problems and the associated challenges in detail.

- **Design, Implementation, and Evaluation of VIA Components:**

  As mentioned earlier in this chapter, VIA is made of several components. In order to implement VIA in the most efficient manner, various approaches for developing these components should be studied, implemented, and evaluated. Virtual-to-physical address translation, doorbells, and completion queues are among the major components whose implementations are extremely important. How the network interface support can be utilized to implement these components in the most efficient manner is also an important issue which should be addressed.

  Approaches for implementing the virtual-to-physical address translation component should address two major questions: 1) which agent is responsible for performing the translation (i.e. the host processor or the Network Interface Controller (NIC)) and 2) where the address translation tables are stored. VIA doorbells can be implemented in software or hardware. Both software and hardware approaches for implementing doorbells should be studied and evaluated. Similarly, software and hardware implementations of completion queues should be proposed and evaluated.

- **Design and Implementation of VIA:**

  After different design choices for VIA components are evaluated, it is important to show how these components can be put together in order to achieve the lowest latency and highest bandwidth possible. The impact of software and hardware restrictions and limitations on any given platform needs to be taken into account. The division of the work between different units such as the host processor and the NIC may have a significant impact on the overall performance of the communication subsystem. Considering the imbalance in processing power of these units, it is crucial that units with smaller computing power are not overloaded by performing too many operations. Furthermore, the performance of the implementation needs

to be evaluated in a comprehensive manner such that the factors limiting the performance can be identified.

- **Framework for Evaluating VIA Implementations:**

  Considering the importance of VIA and its potential for becoming the standard communication protocol for clusters, providing a systematic method for evaluating different implementations of VIA is of utmost importance. However, no such method is currently available. A suite of micro-benchmarks can be used for this purpose. Such a suite of micro-benchmarks can be used not only for the evaluation of different implementations of VIA, but also for pinpointing performance bottlenecks and possible approaches for alleviating them. Furthermore, such a suite of micro-benchmarks can be used by designers of higher communication layers to achieve the best performance by taking into account the performance of different VIA features.

- **Efficient Shared Memory Support for DSM Applications:**

  DSM applications rely on a global shared address space across different machines. Softwares such as TreadMarks [36, 9] provide such a view and hide from the applications the communication operations required for providing a coherent view of the shared address space. These systems usually use legacy protocols such as UDP and TCP for the required communications. It is important to take advantage of the high performance of user-level communication protocols to perform the required communication operations in the most efficient manner. The request/reply model used in Software DSM (SDSM) systems such as TreadMarks requires the communication subsystem to deal with unexpected requests. Therefore, using an efficient mechanism for notifying the system of the arrival of requests is crucial. Connection management, buffer management, and support for unexpected messages are among other issues which should be studied.

- **Efficient Message Passing Facility for Distributed Memory Applications:**

  In order to pass the high performance of the communication subsystem to the distributed memory applications, the communication primitives as defined in standards such as MPI should be implemented in terms of the low level primitives provided by the underlying communication subsystem. It is crucial to keep the overhead of these implementations as low as possible such that the high performance associated with the communication networks and low-level messaging libraries become available to the applications. The most important issue in implementing standards such as MPI is avoiding any unnecessary data copies. Another important issue is that in MPI, received messages can be matched with posted receives in an out-of-order fashion. Providing such a feature without using extra data copies is another challenge.

1.4 Thesis Overview

Having examined the research issues in the design of efficient communication subsystems for clusters, we now present an overview of our solutions. We focus on improving the performance of applications in cluster environments along five major directions as discussed in Section 1.3.

Chapter 2 focuses on different design issues involved in implementing a high bandwidth, low latency communication subsystem. We evaluate and compare the performance of different implementations of essential VIA components. We discuss the advantages and disadvantages of each design approach and describe the required network interface support for implementing each of

them. In particular, different possible approaches for implementing components such as software doorbells, virtual-to-physical address translation, and completion queues are discussed. We use NAS Parallel Benchmarks [6] to study the effect of caching the address translation tables on the NIC and to study design issues involved in implementing completion queues. We consider two platforms: IBM Netfinity SP cluster running the NT 4.0 operating system and a Myrinet connected cluster of PCs running the Linux operating system. We identify the best choices for implementing each of these components on both of these platforms.

In Chapter 3, we look at a prototype implementation of VIA for SP-connected NT clusters. We use the results presented in Chapter 2 and show how different components of a communication subsystem can be put together in order to achieve the lowest latency and highest bandwidth possible. In particular, we explain how the virtual-to-physical address translation can be implemented efficiently with a minimum Network Interface Card (NIC) memory requirement. We show how caching the VIA descriptors on the NIC can reduce the communication latency. We also present an efficient scheme for implementing the VIA doorbells without any hardware support. A comprehensive performance evaluation study of the implementation is provided. The performance of the implemented VIA surpasses that of other existing software implementations of the VIA and is comparable to that of a hardware VIA implementation. The peak measured bandwidth for our system is observed to be 101.4 MBytes/s and the one-way latency for short messages is 18.2 microseconds. We evaluate the performance of our prototype in a comprehensive manner and present the factors limiting the performance. Furthermore, we show how additional hardware support can be used to improve the performance. It is shown that with hardware support for doorbells and a reasonably large amount of NIC memory, it is possible to provide single digit one-way latency with current technology.

VIA has different components (such as doorbells, completion queues, and virtual-to-physical address translation) and attributes (such as maximum transfer unit and reliability modes). Different implementations of VIA lead to different design strategies for efficiently implementing higher level communication layers/libraries (such as Message Passing Interface (MPI [39]). It also has implication on the performance of applications. Currently, there is no framework for evaluating different design choices and for obtaining insight about the design choices made in a particular implementation of VIA and their impact on the performance. In Chapter 4, we address these issues by proposing a new micro-benchmark suite called Virtual Interface Architecture Benchmark (VIBe). This suite consists of several micro-benchmarks which are divided into three major categories: non-data transfer related micro-benchmarks, data transfer related micro-benchmarks, and client/server micro-benchmarks. By using the new benchmark suite, the performance of VIA implementations can be evaluated under different communication scenarios and with respect to the implementation of different components and attributes of VIA. We demonstrate the use of VIBe to evaluate two implementations of VIA (M-VIA and Berkeley VIA). Through these evaluations we show how the VIBe suite can provide insights to the implementation details of VIA and help higher layer software developers.

As mentioned earlier in this Chapter, VIA, as well as other high performance communication subsystems, has a low-level API which provides only the basic communication primitives, it is difficult for user applications to directly use these primitives unless the applications are rewritten. In Chapter 5, we take on a challenge of developing a communication substrate over VIA such that applications using the popular TreadMarks DSM package can take advantage of the enhanced communication performance of VIA. We take a four-step approach in developing the targeted substrate. First, we identify the mismatches between the communication requirements by TreadMarks and

the services provided by VIA. After identifying these mismatches, we propose a set of schemes to eliminate such mismatches. These schemes include connection setup, buffer management, advance posting of descriptors for unexpected messages, and alternative designs to handle asynchronous messages. We also propose and evaluate different design alternatives for enhancing some VIA functions (such as the VIA Notify mechanism) so that the new substrate can be designed with low overhead. Finally, we derive the best set of alternatives and implement them on two enhanced implementations of VIA (MVIA [3] and Berkeley VIA [24]) on two different networking technologies, Gigabit Ethernet and Myrinet, respectively. We evaluate the performance of our implementation by using several micro-benchmarks and applications. We show that the communication and wait times, and therefore the total execution times of different applications can be significantly reduced by using VIA. A reduction in the overall execution time up to 2.05 on an eight node system is demonstrated in comparison with the original UDP implementation. The new implementation also demonstrates better parallel speedup as the system size increases.

Since a large number of high performance applications are written (and being written) in the distributed memory programming model by using the communication primitives provided by the MPI standard, it is crucial to implement MPI on top of user-level communication protocols. The IBM RS/6000 SP system is one of the most cost-effective commercially available high performance machines. IBM RS/6000 SP systems support the Message Passing Interface standard (MPI) and LAPI [45]. LAPI is a user-level, reliable and efficient one sided communication API library, implemented on IBM RS/6000 SP systems. In Chapter 6, we explain how the high performance of the user-level communication library LAPI has been exploited in order to implement the MPI standard more efficiently than the existing MPI. We describe how to avoid unnecessary data copies at both the sending and receiving sides for such an implementation. The resolution of problems arising from the mismatches between the requirements of the MPI standard and the features of LAPI is discussed. As a result of this exercise, certain enhancements to LAPI are identified to enable an efficient implementation of MPI on LAPI. The performance of the new implementation of MPI is compared with that of the underlying LAPI itself. The latency (in polling and interrupt modes) and bandwidth of our new implementation is compared with that of the native MPI implementation on RS/6000 SP systems. The results indicate that the MPI implementation on LAPI performs comparably or better than the original MPI implementation in most cases. Improvements of up to 17.3% in polling mode latency, 35.8% in interrupt mode latency, and 20.9% in bandwidth are obtained for certain message sizes. It is shown that the implementation of MPI on top of LAPI also outperforms the native MPI implementation for the NAS Parallel Benchmarks.

In Chapter 7, the research contributions of this thesis are summarized. In addition, directions for future research are discussed. Some interesting open problems in related areas are also described.

CHAPTER 2

COMPARISON AND EVALUATION OF DESIGN CHOICES FOR IMPLEMENTING THE VIRTUAL INTERFACE ARCHITECTURE (VIA)

The Virtual Interface Architecture (VIA) [7] is the most important communication protocol developed for clusters. It has been developed to standardize user-level communication protocols. In this chapter, we discuss the essential components of VIA and present different approaches for implementing these components. We discuss the advantages and disadvantages of each approach and present the required support for their implementations. In particular, we discuss different possible approaches for implementing components such as software doorbells, virtual-to-physical address translation, and completion queues. We use the NAS Parallel Benchmarks to study the effect of caching the address translation tables on the NIC and to study different completion queue implementations. We use a subset of VIA implemented on an IBM SP-connected Netfinity cluster [17] running the MS Windows NT operating system and a Myrinet-connected cluster of PCs running the Linux operating system to evaluate different components of VIA.

The rest of this chapter is organized as follows: In Section 2.1, we briefly overview the Virtual Interface Architecture, discuss the VIA send and receive operations in detail, and identify different components involved in these operations. Different design alternatives for implementing these components of VIA are discussed in Section 2.2. The performance evaluation results are presented in Section 2.3. Related work is discussed in Section 2.4. In Section 2.5, we present our conclusions.

2.1 Virtual Interface Architecture (VIA)

In this section we first present an overview of VIA. Then, we discuss different events that occur during the send and receive operations and present the basic components involved in performing these operations. We focus on systems with programmable NICs.

2.1.1 Overview

The Virtual Interface Architecture (VIA) is designed to provide high bandwidth, low latency communication support over a System Area Network (SAN). A SAN interconnects the nodes of a distributed computer system[7]. The VIA specification is designed to eliminate the system processing overhead associated with the legacy network protocols by providing user applications a protected and directly accessible network interface called the Virtual Interface (VI).

Each VI is a communication endpoint. Two VI endpoints on different nodes can be logically connected to form a bidirectional point-to-point communication channel. A process can have multiple VIs. A send queue and a receive queue (also called as work queues) are associated with each VI. Applications post send and receive requests to these queues in the form of VIA descriptors. Each

descriptor contains one Control Segment (CS) and zero or more Data Segments (DS) and possibly an Address Segment (AS). Each DS contains a user buffer virtual address. The AS contains a user buffer virtual address at the destination node. Immediate Data mode also exists where the immediate data is contained in the CS. Applications may check the completion status of their VIA descriptors via the *Status* field in CS. A doorbell is associated with each work queue. Whenever an application posts a descriptor, it notifies the VIA provider by ringing the doorbell. Each VI work queue can be associated with a Completion Queue (CQ) too. A CQ merges the completion status of multiple work queues. Therefore, an application need not poll multiple work queues to determine if a request has been completed.

The VIA specification requires that the applications *register* the virtual memory regions which are going to be used by VIA descriptors and user communication buffers. The intent of the memory registration is to give an opportunity to the VIA provider to pin (lock) down user virtual memory in physical memory so that the network interface can directly access user buffers. This eliminates the need for copying data between user buffers and intermediate kernel buffers typically used in traditional network transports.

The VIA specifies two types of data transfer facilities: the traditional send-receive messaging model and the Remote Direct Memory Access (RDMA) model. In the send/receive model, there is a one to one correspondence between send descriptors on the sending side and receive descriptors on the receiving side. In the RDMA model, the initiator of the data transfer specifies the source and destination virtual addresses on the local and remote nodes, respectively. The RDMA write operation is a required feature of the VIA specification while the RDMA read operation is optional. In this chapter, we focus on the send/receive messaging facilities of VIA.

2.1.2 Message Passing in VIA

For sending and receiving messages, the following major steps are taken:
Constructing the descriptor: The application creates a descriptor in a registered memory region. This descriptor includes the virtual address of the send or receive buffer and its length. The message buffer is allocated from a registered memory region. The descriptor also contains a status field which the VIA provider updates upon completion of the operation. **Posting the descriptor:** The application posts the descriptor using the `VipPostSend` or `VipPostRecv` function call. Through the doorbell mechanism, the NIC is informed about the existence of the posted descriptor. **Obtaining the descriptor by the NIC:** The NIC retrieves from the descriptor the information required for sending or receiving a message. The information includes the address and the length of the user buffer and the address of the status field of the descriptor. **Performing the operation:** The NIC performs the send operation by injecting the data into the network after it is transferred from the user buffer to the NIC. For the receive operation, the message is received from the network into the NIC memory and then into the user buffer. **Marking the descriptor as complete:** After performing the send or receive operation, the NIC marks the status field of the VIA descriptor as complete. If a CQ is associated with the VI, the NIC also makes an entry in the CQ so that the application can detect the completion through CQ as well. **Application detecting the completion of the operation:** The application can check the status of the operation using `VipSendDone` and `VipRecvDone` in a non-blocking fashion, `VipSendWait` and `VipRecvWait` in a blocking fashion, and `VipCQDone` and `VipCQWait` if a CQ is associated with the corresponding work queue.

2.1.3 Basic Components of VIA

Considering different operations involved in sending and receiving messages, three major components can be identified as the basic components of the message passing operations. These components are: 1) informing the NIC of an outstanding send or receive request, 2) the NIC obtaining information about the outstanding operation and corresponding user data buffers and performing the operation, and 3) the NIC informing the user program of the completion of send and receive operations. In order to implement the send and receive operations efficiently, it is crucial to implement these components as efficiently as possible. In the next section, we present different design alternatives for implementing these components and present the pros and cons of each of them. It should be noted that we only consider the methods which do not require any unnecessary data copies.

2.2 Design Alternatives

In this section, we discuss the implementation of doorbells which are related to the first component of message passing operations and are used for informing the NIC of the existence of outstanding send or receive descriptors. We also study different implementations of virtual-to-physical address translation and the possibility of caching descriptors. These two issues relate to the second basic component or the mechanism through which the NIC obtains information about the outstanding operations and corresponding user data buffers. The third component, the mechanism through which the user program is informed of the completion of send and receive operations, is also discussed with respect to the implementation of completion queues.

2.2.1 Doorbells

VIA specifies that each VI be associated with a pair of doorbells. The purpose of a doorbell is to notify the NIC of the existence of newly posted descriptors. Doorbells can be implemented in hardware or software. However, most of the current generation NICs do not provide any hardware support for doorbells, they need to be implemented in software. Therefore, in this chapter, we focus on the design choices for implementing doorbells in software.

Approach 1 (D1): One approach for implementing doorbells in software is allocating space for each doorbell in the NIC memory and mapping it to the address space of the process. The user application rings the doorbell by simply setting the corresponding bit in the NIC memory or by writing the address of the descriptor (or the descriptor itself) in the NIC memory. To protect a doorbell from being tampered by other processes, doorbells of different processes need to be on separate memory pages in the NIC since protection granularity of a kernel is one page (e.g. 4KB). The advantage of using this mechanism is that there is no need to go through the kernel for ringing the doorbells and this operation can be implemented in user space. The disadvantage of this approach is the cost of polling the VIs for send descriptors. As the number of active VIs increases, the NIC spends more time polling the send doorbells to check if there is any send descriptor to be processed. This limits the scalability of the communication subsystem. The other shortcoming of this approach when a single word or bit is used for each VI is that when a descriptor is posted, the subsequent post cannot proceed until the NIC becomes aware of the first posted descriptor. To overcome this shortcoming, a circular buffer can be used as a queue for each VI such that multiple descriptors can be posted by the user application even when the NIC firmware is busy performing

other operations (such as sending and receiving messages) and hasn't become aware of some of the posted descriptors yet.

Approach 2 (D2): In order to avoid the cost of polling of VIs for send descriptors, a second approach in which the kernel intervention is required can be used. In this approach, a centralized queue of send descriptors (or handles to descriptors) are maintained by the NIC. Since all VIs share the same centralized queue, a mechanism is required to guarantee that this queue is accessed in an operating system safe fashion. Thus kernel intervention is required. In this approach, the need for polling all the active VIs is eliminated and the NIC needs to only look at the centralized queue for send descriptors. The disadvantage of this approach is the added delay of going through the kernel. The advantage of this approach is the elimination of the NIC polling active send requests.

The problem of polling send descriptors does not occur for receive descriptors. When a message is received at the NIC, the VI id of the received message is used to obtain the receive descriptor posted for that particular VI. If for some reasons the posted receive descriptors need to be preprocessed before the messages arrive (for example to perform the virtual-to-physical address translation which will be discussed later) then finding receive descriptors requires polling the active VIs and causes a similar problem.

2.2.2 Caching Descriptors

As discussed in Section 2.1.2, when the NIC recognizes that a descriptor is posted, it needs to obtain the information about the message (such as the user buffer address and the size of the message) from the descriptor. The descriptors are constructed by the VIA applications and therefore are stored in the host memory. The question is whether the host initiates the transfer of the descriptor or the NIC. Since DMA is the only way by which most NICs can access the host memory but the host can use PIO for transferring data to the NIC, there is a tradeoff between these two approaches with respect to the size of the descriptor being transferred from the host memory to the NIC memory. For the receive descriptors, the advantage of moving the descriptors to the NIC memory when the descriptors get posted is that the time for this transfer is not part of the the message latency. It should be noted that the host processor is required to be involved in PIO operations while the DMA operations are performed without the involvement of the host processor. We'll have a performance evaluation of these methods in Section 2.3.2.

2.2.3 Address Translation

Most NICs (including the widely used PCI based NICs) use physical addresses for performing DMA operations, whereas VIA descriptor elements, e.g. user buffer addresses, are virtual addresses. Therefore virtual-to-physical address translation is required. This address translation is required not only for transferring data, but also for accessing descriptors (if they are not cached in the NIC memory) and updating the status of operations by NIC. VIA specifies a memory registration mechanism to ensure that the page frames which are accessed by the NIC are present in the physical memory. Registered virtual memory pages are pinned down in physical memory. Before data is transferred to or from these memory regions, the virtual addresses should be translated to physical addresses. It should be noted that using approaches such as using a preallocated pinned contiguous buffer (at the boot time) from which user buffers are allocated or using DMA regions through which data transfers to and from NIC are performed is not reasonable. Allocating user buffers from a preallocated buffer requires modifications to the applications to use a custom routine for user buffer

allocations. Using DMA regions for data transfers is not a viable choice because of the required extra data copies to and from these regions at the sending and receiving nodes.

Two critical issues in implementing the address translation for VIA are the location of address translation tables (commonly known as Translation Lookaside Buffers or TLBs) and the method of accessing them (i.e. whether the host or the NIC performs the translation). The VIA TLBs can be located in the host or NIC memory and can be accessed by the host or the NIC. Therefore, there are four possible approaches for performing the address translation: 1) the TLB is in the host memory and host performs the address translation, 2) the TLB is stored in the NIC memory and the NIC does the address translation, 3) the TLB is located in the host memory and the NIC performs the translation, and 4) the TLB is in the NIC memory and the host performs the address translation. Among these approaches, the fourth approach does not provide any advantage over the other approaches and has no practical use. In the rest of this section, we discuss the other three approaches in more detail.

Approach 1 (AT1): In this approach, the TLB is located in the host memory and the address translation is performed by the host. Since the user processes can not be trusted to provide the physical addresses, the translation (the TLB lookup) is performed in kernel space. The disadvantage of this approach is the need for user to kernel context switching. Since the VIA requires all data buffers to be in registered memory regions, the TLB lookup cost can be minimized by the creation of an address translation table for each registered memory region. This table should include the addresses of all the physical page frames which correspond to the memory region. By creating such a table at the memory registration time, the address translation can be efficiently done by indexing this table. The advantage of this approach is that the NIC memory requirement is small since the TLB is located in the host memory.

Approach 2 (AT2): In this approach, the TLB is located in the NIC memory and the NIC is responsible for performing the virtual-to-physical address translation. The limitation of this approach is the size of memory required for the TLB. For example, in order to support 256 MB of registered memory, a TLB of 256 KB is required. The available memory of the NIC is usually much smaller than that of the host, and the memory required for storing the TLB puts a heavy burden on the NIC resources and makes the implementation not scalable.

Approach 3 (AT3): In this approach, the TLB is located in the host memory but the translation is done by the NIC. The advantage of using this approach is that there is no need for using a big portion of the NIC memory for storing the TLB. The disadvantage of this approach is that the NIC requires to access the host memory for obtaining the translation. This access is usually done by a DMA operation and may have a high DMA startup delay. In order to minimize this problem, a portion of the NIC memory can be used to cache the translations such that future references to a particular page frame can be resolved without accessing the host memory. The size and characteristics of this cache along with the behavior of the application programs affect the overall performance of the address translation operation.

We discuss the cost of implementing these approaches for the virtual-to-physical address translation in Section 2.3.3.

2.2.4 Completion Queues

As mentioned in Section 2.1.1, each work queue can be associated with a Completion Queue (CQ). In these cases, the notification of completed requests should be directed to a CQ on a per-VI work queue basis. The description of the `VipCQDone` states that it is possible to have multiple threads of a process wait on a CQ and its associated work queues [7]. Therefore, the VIA provider

updates both the work queue and its associated CQ upon the completion of a request. Marking a descriptor as complete (in the work queue) is done by DMAing the status field of the descriptor (with the bit corresponding to the completion of the operation set) from the NIC to the host. For supporting the CQs, there are two possible approaches.

Approach 1 (CQ1): In this approach, the NIC in addition to updating the status field of the descriptor, inserts the descriptor handle into the associated CQ. The disadvantage of this approach is that an extra DMA operation is required for the insertion of the descriptor handle to the CQ. The advantage of this approach is that the application spends constant time checking for a completed operation regardless of the number of work queues associated with a CQ.

Approach 2 (CQ2): In this approach, no entries are added into the CQ. In fact there is no CQ in the host memory. The completed operations are simply found by polling the work queues associated with the CQ. That is, the `VipCQDone` function is implemented such that either `VipSendDone` or `VipRecvDone` is called for each work queue associated with the CQ. The advantage of this approach is that NIC need not perform a DMA operation for inserting the handle of the completed descriptor into the CQ. The disadvantage of using polling in this manner is that the method does not scale well with the increase in the number of work queues associated with a CQ. However, since in many applications each node communicates only with a small set of other processes, and therefore a limited number of work queues are associated with each CQ, this approach may be viable for implementing CQs.

We compare the cost of the implementation of these two approaches in Section 2.3.4. We also investigate how the scalability issue of the second approach can be dealt with.

2.3 Performance Evaluation

In order to evaluate different design alternatives discussed in Section 2.2, we implemented a subset of VIA on two different systems. The first system consisted of 300 MHz Pentium II PCs with 128 MB of SDRAM and a 33 MHz/32-bit PCI bus and ran the Linux 2.0 operating system. The Myrinet switches and 33 MHz LANai 4.3 NICs were used as the interconnect [22]. The second system was an IBM Netfinity SP switch-connected Cluster [17]. This cluster consisted of 450 MHz Pentium III PCs. Each node had 128 MB of SDRAM and a 33 MHz/32-bit PCI bus and ran the NT 4.0 operating system. These PCs were interconnected by an IBM SP switch and 100 MHz TB3PCI NICs [17]. These two testbeds represent a wide range of available network-based computing platforms.

In the rest of this section, we first present the cost of the basic operations in these two systems. Then, we evaluate and compare different alternatives for implementing different components of VIA.

2.3.1 Basic Operations

Since Programmed I/O (PIO) and DMA are the major methods for transferring data between the host and the NIC, we measured the cost of these operations. We also measured the cost of user to kernel space switch for both systems. For our NT testbed we used the Fast IO Dispatch method [53] and for our Linux testbed, we used a fast trap. These measurements are presented in Table 2.1.

Operation	Myrinet-Linux	SP-NT
Host PIO Write	0.16 μs/word	0.33 μs/word
Host PIO Read	0.49 μs/word	0.87 μs/word
User-space to Kernel-space	1.06 μs (Fast Trap)	2.27 μs (Fast IO Dispatch)
DMA Startup (host to NIC)	1.72 μs	1.78 μs
DMA Startup (NIC to host)	1.47 μs	1.61 μs

Table 2.1: Cost of basic operations in the Myrinet-Linux and SP-NT testbeds.

2.3.2 Caching Descriptors

As discussed in Section 2.2.2, the choice of caching the send descriptors when they are posted depends on the cost PIO and DMA operations. From the cost of these operations in our testbeds (Table 2.1), it can be seen that transferring up to five words through PIO is less time consuming than using the DMA in the Netfinity SP system. In the Myrinet-Linux testbed, transferring up to ten words can be done in a faster manner by using PIO. It should be noted that in neither of our testbeds, PCI write combining was used. If a system supports PCI write combining, a larger number of words can be transferred by PIO before the point where using DMA becomes more efficient. Another factor which affects the decision about caching send descriptors is the CPU utilization. While the host processor is not involved if DMA is used, using PIO requires the host to perform the transfer and increases the host CPU cycles used for send operations.

The situation is slightly different for receive descriptors. If the receive descriptors are to be accessed by DMA operations, a simple implementation performs the DMA when the corresponding message is received at the NIC of the receiving node. This will result in an increase in the latency by the cost of transferring the descriptor to the NIC. However, if the descriptor is cached at the time it gets posted, in most cases the cost of this transfer is not part of the send and transmission times of the message. Even if the NIC is responsible for the transfer, it is possible to mask the transfer time for receive descriptors by transferring the descriptors before the corresponding messages arrive at the NIC. However, implementing this feature requires an increase in the complexity of the NIC firmware. Furthermore, the NIC may need to poll all the receive queues of active VIs to see if there is any posted receive descriptor to be processed. Since the NIC processors are usually much slower than the host processors (4.5 times in our Netfinity cluster and 10 times in our Myrinet network), the increase in the complexity of firmware and the need for polling can degrade the performance of the firmware and increase the latency of messages. Furthermore, if the rate of incoming messages is high and/or the rate of messages being sent out from a particular node is high, the NIC may not get a chance to get the receive descriptor before the message arrives. In these situations, before NICs can retrieve the information about the descriptor, it has to store the message in a temporary location. If the message is kept in the NIC memory, messages might be dropped or the reception of messages might need to be stalled because of the usually small amount of available NIC memory. If the temporary storage is in the host memory (with an address known to the NIC), there will be an unnecessary data copy. Either way, the performance of the communication subsystem will degrade.

It is to be noted that the whole descriptor need not to be cached. Only those portions of the descriptor which are required by the NIC should be cached. In particular, the address and size of the data buffer, the control field of the descriptor (which includes the information such as the type of the operation) and the address of the status field of the descriptor should be cached on the NIC.

2.3.3 Address Translation

Three approaches for performing the address translation were discussed in Section 2.2.3. In the first approach (AT1), where the TLB is in the host memory and the host performs the translation, the cost of the address translation is essentially the one time user space to kernel space switch for each send or receive operation and the cost of the table lookup for each page frame of the send or receive buffer. In order to reduce the TLB lookup cost, one table for each registered memory can be created upon the registration of the memory region. This table includes the physical addresses of (the beginning of) all the page frames that the memory region spans over. By creating such a table, the virtual-to-physical address translation can be done by indexing the address translation table without any need for searching the table or multiple indirections. The Average cost of the address translation when the AT1 approach is used, is shown in the first row of Table 2.2. The overall cost of the translation is this additional cost plus the time required for accessing the TLB for each page frame of the send or receive buffer.

AT **Method**	**Location/ Translator**	**NIC Memory Requirement**	**Myrinet-Linux Additional Cost**	**SP-NT Additional Cost**
AT1	host/host	None	1.06	2.27
AT2	NIC/NIC	Proportional	0	0
AT3	host/NIC	Constant	$1.72 \times Miss\ Rate$	$1.78 \times Miss\ Rate$

Table 2.2: Cost of different methods of implementing the virtual-to-physical address translation. (See Figures 1 through 4 for the value of Miss Rate for different benchmarks.)

In the second approach (AT2), where the TLB is located in the NIC memory, a similar mechanism can be used. In this method, there is no need to go through the kernel for the address translation. The second line in Table 2.2 shows the additional cost for performing the translation by using this approach. It can be seen that this additional cost is zero. The overall cost of the translation for each send or receive operation is equal to the number of page frames of the send or receive buffers times the time required to access an element of the TLB. The cost of registering memory regions is increased in this method because the TLB should be created and transferred to the NIC. Creating the TLB on the NIC requires multiple PIO write operations (based on the size of the registered memory). However, since the memory registration happens infrequently, this increase in the cost of memory registration can be tolerated. The more limiting factor for implementing this approach is the large memory space required for keeping the TLBs on the NIC. While there are NICs with large amount of memory, most NICs provide a limited amount of memory. On the other hand, with the increase in the size of available host physical memory and registered memory

regions, the required memory on the NIC increases. These requirements make the third approach a more realistic and scalable approach for implementing the address translation.

In the third approach (AT3), the NIC perform the translation while the TLB is stored in the host memory. Since the TLB is stored in the host memory, the memory requirement on the NIC is minimal. However, if for every address translation the NIC is required to access the host memory (through DMA) this approach performs much worse than the second approach. In order to reduce the cost of the address translation while the size of required NIC memory is kept low, caching the address translations is used. If the translation of a particular physical address is found in a software cache (kept in the NIC memory), the translation can be performed quickly by accessing the corresponding cache entry. If the translation is not found in the cache, an access to the TLB in host memory (through DMA) is required (Table 2.2).

In order to evaluate the effectiveness of caching and estimating the required cache size, and in the absence of the existence a wide variety of applications and benchmarks for VIA, we used the NAS Parallel Benchmarks (NPB) [6, 12] version 2.3 to gather the list of addresses being referred in these benchmarks. We profiled the NAS benchmarks to record the addresses of the send and receive buffers being used in these benchmarks. We ran the benchmark with 4, 16, and 64 processes and used two different problem sizes: class A and class B. We used different TLB cache sizes and degrees of associativity. It should be noted that the TLB cache is implemented in software and is stored in the NIC memory. (We haven't presented the data for the Embarrassingly Parallel (EP) and Fast Fourier Transform (FT) benchmarks because the communication operations used in these benchmarks are such that the performance of the address translation does not affect the execution time of the program significantly.)

Figure 2.1 shows the cache miss rates for the class A NAS benchmarks on a system with 128-entry direct-mapped caches. The results for running these programs on four and 16 processes are shown and cache misses are broken down into send and receive misses (compulsory and non-compulsory). It can be seen that with such a small cache and when four processes are used, in four of the benchmarks more than 80% of memory accesses result in a cache miss. When the programs are run on 16 processes the number of cache misses reduces significantly. If the cache size is increased to 1024 (Fig. 2.2), the cache miss rates for all benchmarks other than IS become negligible. Increase in the number of processes result in an decrease in message sizes and this compensate the effect of the increase in the number of messages being transmitted. It is interesting to see that miss rates are identical for a 1024-entry direct mapped cache or a 1024-entry cache with the degree of associativity of eight. The access time of a software direct-mapped cache is less than that for a software associative cache. Therefore, given the same performance, using a direct-mapped cache is preferred over an associative cache when implemented in software.

Figure 2.3 shows the cache miss rates for the class B NAS benchmarks on a system with 128-entry direct-mapped caches. Note that the results shown in this figure have been obtained from running these programs using 16 and 64 processes. The cache miss rates for systems with 1024-entry caches are shown in Figure 2.4. A similar pattern to those for class A benchmarks (smaller problem size) can be seen. It is interesting to compare the cache miss rates for these benchmarks with different problem sizes. When the benchmarks use 16 processes, increasing the problem size (from class A to class B) result in an increase in the cache miss rates. Using caches with 1024 entries are shown to be enough to make the cache miss rates for all class A benchmarks negligible. However, when the problem size is increased, the BT and IS benchmarks produce a significant number of misses.

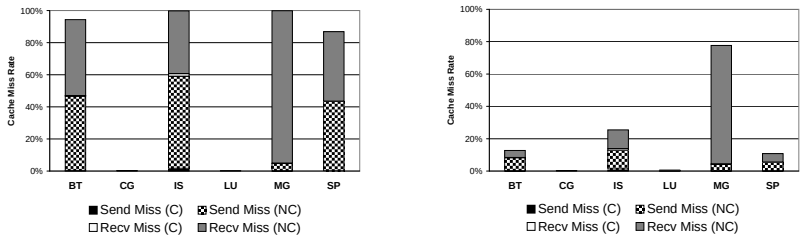

Figure 2.1: The cache miss rate for the NAS benchmarks (class A) using four processes (left) and 16 processes (right) with 128-entry direct-mapped caches. C and NC denote compulsory and non-compulsory misses, respectively.

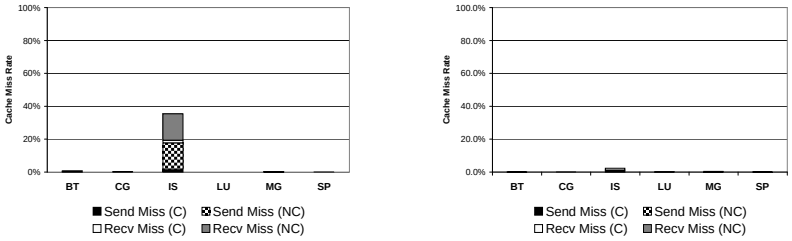

Figure 2.2: The cache miss rate for the NAS benchmarks (class A) using four processes (left) and 16 processes (right) with 1024-entry direct-mapped caches and 128-entry 8-way associative caches. (The miss rates are identical for both of these cache types.) C and NC denote compulsory and non-compulsory misses, respectively.

It can be seen that providing a larger cache size reduces the number of misses significantly. The required cache size for making cache misses negligible is shown to be very small. We have also studied the effect of using victim caches. The results show that the gain obtained from using victim caches is minimal. (We do not present the results for victim caches here because of the space limitation.) It should be noted that the NAS benchmarks are only representative of scientific applications and other applications and benchmarks need to be used to evaluate the caching for VIA too.

It should be noted that for receive operations, the cost of address translation might be hidden if the translation is done before the message arrives. The AT1 method can be easily used to take advantage of this characteristic. But the AT2 and AT3 methods can be implemented more easily if the translation is done when the message arrives. When the AT2 and AT3 methods are used, performing the translation before the message arrives increases the complexity of the firmware and can decrease the overall performance of the communication subsystem. Another issue which should be considered is that while for performing the address translation by using the AT3 approach the host processor is not involved, the AT1 approach requires the host to perform the translation.

Another important issue worth mentioning is the translation of the address of the status field of descriptors. Since after the completion of an operation, the status field of the corresponding descriptor should be updated, the NIC needs to know the physical address of the status field.

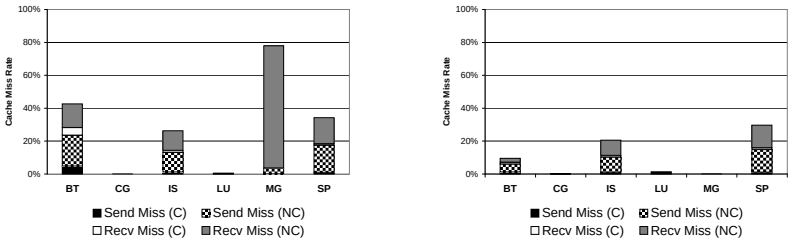

Figure 2.3: The cache miss rate for the NAS benchmarks (class B) using 16 processes (left) and 64 processes (right) with 128-entry direct-mapped caches. C and NC denote compulsory and non-compulsory misses, respectively.

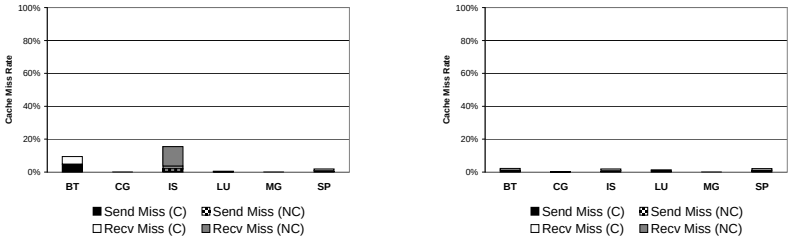

Figure 2.4: The cache miss rate for the NAS benchmarks (class B) using 16 processes (left) and 64 processes (right) with 1024-entry direct-mapped caches and 128-entry 8-way associative caches. (The miss rates are identical for both of these cache types.) C and NC denote compulsory and non-compulsory misses, respectively.

(Obviously, this update could be done by issuing an interrupt to the host, but this approach will be too costly to be used in situations where the application is polling for the completion of an operation.) If the address translation is to be done by the NIC, there will be a need to access the TLB one more time to perform the translation of the status field address for each operation.

2.3.4 Completion Queues

We presented two approaches for implementing the completion queues in Section 2.2.4. The cost for the first approach (CQ1) is practically the cost of NIC performing a DMA operation to add an entry to the CQ. In the second approach (CQ2) the work queues associated with a CQ are polled. CQ2 approach won't be scalable if the number of work queues associated with a CQ is large. On the other hand in many real-life applications each process usually communicates only with a small set of processes. In order to evaluate the performance of CQ2, we used the NAS benchmarks. Among the NAS benchmarks, the LU and MG benchmarks use the `MPI_Waitany` function to receive any message from a collection of processes. Usage of this primitive is similar to waiting to receive a message by examining the completion queue associated with a set of VI receive queues. In order to find out the number of work queues associated with a CQ, we recorded the number of processes with which a process communicates and waits for the completion of the transfers by using the

	Proce-sses	Avg. # of Recv Queues / CQ	Average CQ2 Cost	Myrinet-Linux CQ1 Cost	SP-NT CQ1 Cost
LU	4	2	0.16	1.47	1.61
LU	16	3	0.24	1.47	1.61
LU	64	3.5	0.28	1.47	1.61
MG	4	3	0.24	1.47	1.61
MG	16	4.6	0.37	1.47	1.61
MG	64	6.5	0.52	1.47	1.61

Table 2.3: Comparison between different approaches for implementing CQs.

`MPI_Waitany` function. Table 2.3 shows the average number of processes a process communicates with using `MPI_Waitany` function in a 64-process system running the LU and MG benchmarks. The data shows that processes communicate with only a small set of processes. For example, in MG benchmark running on 64 nodes, each process communicates to 6.5 other processes on the average. Polling the VI work queues of these 6.5 processes is less time consuming (0.52 microseconds) than the NIC adding a completion entry to CQ (1.61 microseconds). It can be seen that the cost of the CQ2 approach is less than that of the CQ1 approach for these applications. It should be noted that the host CPU utilization is higher for the CQ2 approach.

2.4 Related Work

There have been several implementations of VIA. The Berkeley VIA (Version 1) [25] is one of the first software implementations of the VIA. (This implementation is a partial implementation of VIA mainly done to obtain a better insight on different aspects of the implementation of the VIA.) In this implementation, a memory page on the NIC memory has been used for the implementation of a pair of doorbells. The doorbells for send queues are polled for finding outstanding send descriptors. This polling is expensive and increases linearly with the number of active VIs. The Berkeley VIA does not perform any caching of descriptors. In other words, for sending messages NIC has to access the host memory twice: once for obtaining the descriptor and once for obtaining the data itself. In this implementation, only a subset of descriptors are moved between the host and the NIC to reduce the high cost of transferring the descriptors. The Berkeley VIA (Version 2) [24] is based on the the Berkeley VIA (Version 1) implementation and adds memory registration and increased VI/user support. In this implementation each memory page on the NIC can support up to 256 pairs of doorbells that belong to a single process. For the address translation a buffer with limited size on the NIC is used for the TLB. If the size of registered memory is bigger than what can be supported with this table, the translation of some portions of the registered memory won't be present in the NIC TLB. In these cases the host memory is accessed to obtain the translation. The location of the host buffers holding the complete translations for registered memory regions are known to the NIC. The FirmVIA [17] is an experimental implementation of the Virtual Interface Architecture for the IBM SP Switch-Connected NT cluster which is one the newest clustering platforms available. In this implementation, the address translation is performed by the host. Descriptors are also cached

for improving the performance. The performance of GigaNet cLAN [1] and the Tandem ServerNet VIA implementations are studied in [48].

The effect of using caching for address translation for user-level network interfaces (and in particular U-Net) has been presented in [56]. The address translation issues have also been studied in [44] and the address translation methods are classified according to where lookup and the miss handling are performed. The major difference between the address translation in systems discussed in these papers and that in systems supporting VIA is the memory registration mechanism required by VIA. In VIA, all the memory locations used as send and receive buffers are in registered memory regions and VIA implementations are not concerned with the possibility of accessing a location which belongs to swapped page frames.

2.5 Summary

In this chapter, we studied different components of VIA for sending and receiving messages. We presented various approaches for implementing different components of VIA and evaluated these approaches on two different platforms. We showed that caching the descriptors in the NIC memory can improve the performance of the communication subsystem by overlapping some portions of the receive overhead with those of send and transmission overhead. Using the NAS benchmarks, we showed that a small caching area for the address translation entries eliminates the need for accessing TLBs stored in the host memory for most of the send and receive operations. We also discussed the issues related to the implementation of completion queues. We showed that a software implementation (polling) performs well for the NAS benchmarks because of the limited number of processes with which a given process communicate. We also presented a few approaches for implementing VIA doorbells in software.

CHAPTER 3

EFFICIENT VIRTUAL INTERFACE ARCHITECTURE SUPPORT FOR IBM SP SWITCH-CONNECTED NT CLUSTERS

In the previous chapter, we presented several approaches for implementing various components of VIA. We also evaluated these alternative approaches on two different platforms. It is important to study the overall performance of the implementation when VIA components are put together. In this chapter, we present the results of an exercise in implementing a subset of VIA on the IBM SP systems hardware using the NT 4.0 operating system. We call our implementation *FirmVIA,* which stands for VIA implemented in firmware.

The IBM SP is one of the most successful parallel systems commercially available today. The IBM SP system consist of RS/6000 nodes running AIX. These nodes are interconnected by the IBM SP switch interconnect. The RS/6000 SP network interface cards (NIC) support the TCP/IP protocol suite and a proprietary user–space protocol. The MPI and LAPI [45, 14] communication libraries are layered over the proprietary user–space protocol. During 1999, IBM announced the *Netfinity SP System* which is an SP Switch–connected NT cluster. The Netfinity SP nodes are based on Intel x86 architecture and run the NT 4.0 operating system. Netfinity SP supports the TCP/IP protocol suite over the SP Switch. The implementation of a low–level, high–performance communication subsystem such as VIA for the Netfinity SP system seems to be the next logical step.

We studied approaches in implementing VIA efficiently on the Netfinity SP system. However, we believe that our results are applicable to many other hardware and software platforms. There were many challenges in the implementation including: 1)performing fast and efficient virtual-to-physical address translation, 2) eliminating the double indirection of VIA, and 3) fast implementation of VIA doorbells on the NIC in the absence of any hardware support and without using polling. In this chapter we address all of these issues. The main contributions presented in this chapter are:
1) We explore the partition of the VIA functions among the user space, the kernel space, and the NIC firmware. A NIC processor is generally not as powerful as host processor in current systems. In SMP systems, multiple host processors need to communicate with a single NIC processor. Thus, in our design, only the operations that impact latency and bandwidth are performed by the NIC. We describe mechanisms for offloading NIC housekeeping tasks to the host processor.
2) We introduce the notion of a *Physical Descriptor* (PD) which is a condensed VIA descriptor with all the virtual addresses translated to physical addresses. PDs allow for efficient virtual-to-physical address translation without putting burden on the NIC processor or the NIC memory. PDs are cached in the NIC memory. There is no separate Translation Lookaside Buffer (TLB) on the NIC. This approach makes most efficient use of the NIC memory. Physical Descriptors are written to the NIC by the host instead of DMA to eliminate the NIC overhead. Therefore, in our design, the so called *double indirection* of VIA is implemented efficiently.

3) In the send/receive model, caching PDs in the NIC memory eliminates the need for stalling the reception of messages (for doing address translation lookup from host memory), or the need for copying the received data into intermediate buffers. Therefore, we implement a zero-copy protocol both on sending and receiving ends, transferring data directly between the user buffer and the NIC. 4) In the absence of hardware support for doorbells, we use a centralized (but protected) doorbell/send queue for caching PDs on the NIC. Firmware overhead of polling multiple VI doorbells of multiple user processes is eliminated. VIA is intended to be a user space protocol. However, we confirm the observations that going through the kernel is not very costly. In fact, the overhead of going through kernel is more than compensated by eliminating the NIC overhead of polling multiple doorbells and DMA for address translation, as well as supporting multiple user processes easily.

We have measured a peak point-to-point bandwidth of 101.4 MBytes/s for our implementation. This performance number surpasses all published VIA results that we are aware of [3, 24, 25, 48]. The half-bandwidth is reached for messages of 864 bytes. The one-way latency of four-byte messages is 18.2 μs which is better other VIA implementation's latencies except for the hardware implementation of VIA [1]. Performance results of FirmVIA and other VIA implementations are summarized in Table 3 in Section 3.4. It is to be noted that our results are very general and can be easily extended to other hardware and software platforms.

The rest of this chapter is organized as follows: We discuss the characteristics of the SP switches and Network Interface Cards in Section 3.1. The design and implementation issues of the VIA implementation for SP-connected NT Clusters are discussed in Section 3.2. In Section 3.3, we present the experimental results including the latency and bandwidth of our implementation and provide a detailed discussion on different aspects of its performance. Related work is discussed in Section 3.4. In Section 3.5, we present our conclusions.

3.1 IBM SP Switch

In this section we present a brief discussion about the architecture of the IBM Scalable Parallel (SP) Switch. We also provide a brief discussion of the functional modules associated with the switch. We also discuss the architecture of the SP network interface card.

3.1.1 Elements of the SP Switch

The current generation of SP networks is called the "SP Switch". The SP Switch is a bidirectional multistage interconnect incorporating a number of features to scale aggregate bandwidth and reduce latency [51]. The basic elements of the SP Switch are the 8-port switch chips and the network interface cards (NIC) interconnected by communication links. Switch chips provide means for passing data arriving at an input port to an appropriate output port. In the current implementation of Netfinity SP systems, the switch chips and NIC ports have 150 MBytes/s data bandwidth in each direction, resulting in 300 MBytes/s bidirectional bandwidth per link and 1.2 GBytes/s aggregate bandwidth per switch chip.

The switch chip, called the TBS chip, contains eight input ports and eight output ports, a buffered crossbar, and a central queue. All switch chip ports are one flit (one byte) wide. When an incoming packet encounters no contention for the selected output port, it cuts through the crossbar in wormhole fashion [51]. Cut through latency is less than 300 nanoseconds. When an incoming packet is blocked due to unavailability of an output port, flits of the packet are sent to the central queue for temporary buffering until the output port becomes available. The central queue stores

up to 4KB of incoming data and this storage space is dynamically allocated for 8 output ports according to the demand.

The TBS chip is used both in RS/6000 SP and Netfinity SP systems. The TBS chips can be interconnected by links to form larger networks. Basic building block of Netfinity SP network is an 8-port switch board that comprises a single TBS chip. Netfinity SP software currently supports cascading of two 8-port switch boards resulting in a network of maximum of 14 nodes. However, the SP hardware technology allows larger networks to be constructed as evidenced by the 1464 node RS/6000 SP system, the ASCI Blue, in existence (http://www.llnl.gov/asci/).

3.1.2 SP Network Interface Card

In the Netfinity SP systems, each host node is attached to the SP Switch by a PCI based network interface card (NIC) illustrated in Fig. 3.1. The NIC consists of a 100 MHz PowerPC 603 microprocessor, 512 KB SRAM, an interface chip to the network called TBIC2, Left and Right DMA engines for moving data to/from PCI bus and for moving data over the internal bus. Two 4 KB speed matching FIFO buffers (called as Send-FIFO and Recv/Cmd-FIFO) also exist on the NIC for buffering data between the internal bus and the PCI bus. Architecture of this NIC is similar to the Micro Channel based SP2 adapter architecture reported in literature[51, 50] and it is the PCI bus version of the NIC used in the RS/6000 SP systems.

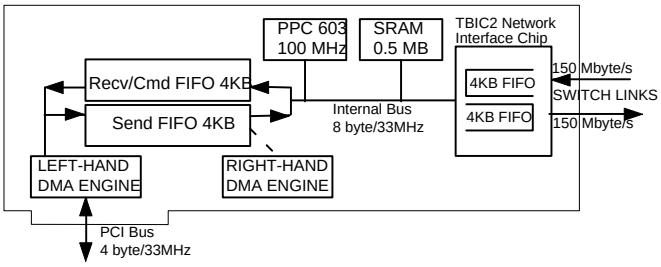

Figure 3.1: The Network Interface Card (NIC) architecture in the Netfinity SP system.

The TBIC2 interface chip has a full duplex switch link capable of moving data at a rate of 150 MBytes/s in both directions. TBIC2 supports variable size switch packets up to 2040 bytes. Each switch packet consists of a 16 byte switch header and payload. The header contains routing instructions for the SP switch chips. The header and payload are written to their respective buffers in the TBIC2. TBIC2 then transmits the packet to the SP Switch to be received at the destination TBIC2.

The 100 MHz PPC603 microprocessor runs the NIC firmware and it is responsible for managing the resources on the NIC. Firmware initiates DMA transfers to send or receive switch packets, creates or decodes switch packet headers, and communicates with the host processor through the SRAM or through interrupts. The TBIC2 registers are memory mapped in the 603 address space. The SRAM is divided mainly into cached and non-cached regions. Cached regions contain the firmware executable and private data. Non-cached SRAM regions are used for communicating with the host processor. The host (an Intel x86 based PC with NT 4.0) typically maps the shared regions of the SRAM into its kernel or user address space. The host processor stores or loads 32-bit

words (using x86 *mov* instruction) to/from SRAM to communicate with the firmware. Firmware can assert the PCI interrupt signal to notify asynchronous events to the host processor.

Two 4KB FIFO buffers are used as an intermediate storage between the PCI bus and TBIC2 (Fig. 3.1). Two DMA engines control one end of these FIFO buffers. The Right-Hand Side (RHS) DMA engine moves data from Send-FIFO to TBIC2 or from TBIC2 to Recv/Cmd-FIFO. The Left-Hand Side (LHS) DMA engine moves data from Recv/Cmd-FIFO to host memory or from host memory to Send-FIFO over the PCI bus. The PPC603 microprocessor communicates with the LHS engine by inserting command words in the Recv/CMD-FIFO. The peak internal bus bandwidth of the NIC is 264 MBytes/s (64 bit/33MHz). The NIC has a 32 bit/33 MHz PCI bus interface resulting in a peak PCI bandwidth of 132 MBytes/s.

3.2 Design and Implementation of FirmVIA

In this section, we first discuss the requirements and scope of our VIA implementation on the SP Switch-connected NT clusters. Then, we discuss the design choices we made and present the rationals for these. We focus on VIA functions which affect the latency and bandwidth and are on the critical path of sending and receiving messages.

3.2.1 Requirements and Scope

We used an RDBMS application's requirements as a guideline for our VIA design and implementation. The application requires 128 VIs per host, 256 outstanding descriptors per work queue, support for a minimum of 256 MB of registered memory and a minimum of 16 registered memory regions, and an MTU size of 4 KB with one data segment per VIA descriptor. Our design meets or exceeds all the requirements. It supports 128 VIs and a MTU of 64KB with any number of data segments. There is no inherent limit in our design for the registered memory size which is only bounded by the amount of memory that the operating system can pin. Even this limit may be exceeded as we will discuss in Section 3.2.2.

We imposed our own requirements to improve performance. VIA send and receive operations are zero–copy thereby moving data directly between the user buffer and the NIC. Status and length fields of the posted VIA descriptors are set by the NIC directly, rather than going through a host interrupt. For polled send/receive operations this results in a smaller application to application latency.

We wrote the firmware entirely in C language except for a few inlined processor control instructions. There is no operating system or run time libraries. Firmware is developed as a single threaded application that runs in an infinite loop multiplexing between various operations such as send and receive. The firmware would have been easier to implement with multiple threads. However, single threading made the firmware latency predictable.

Our design was mostly influenced by limited amount of memory on the NIC. To reduce the development time, we based our firmware on the existing firmware for Netfinity SP systems. This meant that only a small portion of the NIC memory was left to work with. NIC memory was also insufficient for storing virtual to physical address translations needed for a reasonable amount of registered memory. An additional limitation of the NIC is the lack of VIA doorbell support. There is no hardware means for host to interrupt the NIC either.

As it will become apparent in the following sections, our design generally uses the principle of keeping the firmware very simple. Operations that impact latency and bandwidth are performed by the NIC processor whereas housekeeping tasks are offloaded to the host at the expense of spending

many more host cycles. For example, NIC DMA operations generally have a high startup overhead, whereas the NIC overhead of interrupting the host is almost zero. When it is not in the critical path, replacing a NIC DMA operation with the host interrupt service routine activated through PCI interrupt gives better overall performance. There is a temptation to put more functions in the NIC, however a NIC processor is not as powerful as the host processor (or processors in SMP systems). Our experience shows that adding more functionality to the NIC increases the latency and decreases the bandwidth.

3.2.2 Design Alternatives and Practical Choices for Implementation

In this section, we discuss the different design choices we encountered for implementing VIA. We explain the advantages and disadvantages of these choices and discuss the decisions we made in implementing VIA.

Virtual–to–Physical Address Translation

PCI based NICs use physical addresses when doing DMA operations, whereas VIA descriptor elements, e.g. user buffers, are virtually addressed. Therefore virtual to physical address translation is required. VIA specifies a memory registration mechanism intended for this purpose (*VipRegisterMemory*). Registered virtual memory is pinned down in physical memory and address translation tables (commonly known as Translation Lookaside Buffer or TLB) are created. TLB lookup is later performed before data is to be moved by a DMA operation. In our design we create one TLB for each registered memory region. The table is linearly addressed. Thus, no searching is necessary. Address translation is simply done by indexing the table with the virtual page frame number.

Two critical issues in implementing a TLB for VIA are the location of the TLB and the method of accessing it. In order to support 256 MB of registered memory, a TLB of 256 KB is required. The NIC memory is too small for this purpose. An approach to circumvent this problem is to use an intermediate buffer. The buffer should be small enough so that the NIC can support an on-board TLB. However user data needs to be copied to/from this intermediate buffer when sending or receiving messages. We did not choose this option as it would not satisfy our zero–copy requirement, and we decided to put the TLB in the host memory.

Then, the next problem is how to pass the information in the TLB to the NIC. In one approach, the NIC can do the TLB lookup by performing a DMA operation from the TLB in the host memory. In another approach, the host processor can do the TLB lookup on NIC's behalf and then pass the information to the NIC. In our case the first approach has a high startup cost of DMA, complicates the firmware and increases it's execution path. Queuing delays may also occur in send and receive FIFO buffers (Fig. 3.1) which will increase the communication latency. In the second approach, the host processor needs to do the TLB lookup in kernel space, since user space applications cannot be trusted to provide valid physical addresses to the NIC. User to kernel task switch is generally an expensive operation in operating systems. However, the NT 4.0 operating system provides a relatively fast method called FAST IO Dispatch [53]. We measured the overhead of this method to be 2.27 microseconds on our host system. Therefore, we decided to go through the kernel and have the host processor perform the translation. This approach promises to significantly simplify the firmware as well as results in similar if not better latency than the first approach. There are also other reasons to go through the kernel such as for ringing VIA doorbells as we will describe later in this section. Thus, we followed the second approach.

To implement the second approach, we defined a data structure called Physical Descriptor (PD). In essence, a PD is a subset of a VIA descriptor with virtual addresses of user buffers and key control segment fields translated to physical addresses. The PD contains only the portions of the VIA descriptor needed by the NIC. The PD consists of two parts: the translated control segment (PDCS) and the translated data segments (PDDS). The host processor creates a PD by a TLB lookup of user buffer addresses specified in the data segments and the status field address in the control segment. The status field physical address is required in a PDCS so that the NIC can set the completion status directly and reduce communication latency. A single data segment may span multiple physical page frames. Therefore, a PDDS may contain a list of physical addresses. For example, a 64 KB VIA data segment may result in as many as 17 physical addresses in PDDS (or 16 if the buffer is aligned on 4KB page boundary.)

Caching Physical Descriptors

When the application posts send and receive descriptors the VIA provider queues them in send and receive work queues, respectively. Descriptors may be queued in the host memory but eventually the NIC needs each descriptor so as to transfer data to/from user buffers specified in the descriptors. In one approach, the descriptors can be queued only in the host memory and the NIC fetches the descriptors by DMA as needed. However, as stated before, there is a high startup cost associated with DMA operations. More importantly, when receiving a message from a high speed network, there is little time for the NIC to fetch the desired descriptor from the host memory. If the NIC cannot fetch the descriptor fast enough it may need to stall the reception of the message. This can result in message packets backing up into the network which may eventually block the entire communication in the network.

Therefore, we decided to use an alternative approach and cache PDs on the NIC whenever they get posted. Fortunately, VIA descriptors exhibit high locality of reference for the VIA send and receive operations since they are consumed in sequential order and thus cache hit rate is essentially 100%. Each VI has its own caching area in the NIC memory for receive descriptors as shown in Fig. 3.2. This area is called Receive Queue Cache (RQC). (We will discuss caching the send descriptors in Section 3.2.2.) The RQCs are circular FIFO queues implemented in the NIC memory. There is a tail and head associated with each RQC. When the application posts a receive descriptor, the host processor creates a PD and writes it into the RQC starting at the tail location and advances the tail to next available location. When a switch packet arrives at the NIC from the network, the firmware determines the VI id of the message and the first descriptor in the corresponding RQC is consumed. The firmware advances the head upon consuming the descriptor.

Because of the relatively small amount of NIC memory not all posted receives can be cached in an RQC. Then the host processor queues the request in the host memory. As cached descriptors are consumed by messages received, RQCs will have free space. Then two possibilities exist for caching new descriptors: 1) using DMA operations to transfer new descriptors to the NIC, or 2) interrupting the host processor to write more descriptors into the RQC, called "refill interrupt." The first approach complicates the firmware but spends no host processor cycles. The second approach of using interrupts keeps the firmware simple and minimizes NIC cycles. However, it uses many more host processor cycles due to the interrupt.

In order to keep the firmware simple we chose to implement the interrupt method at the expense of wasting host processor cycles because the RQC refill operation is not in the critical path that affects latency or bandwidth provided that the descriptors in the RQC are not depleted. If there is insufficient cache space for a descriptor a handle to the descriptor is queued in the kernel space.

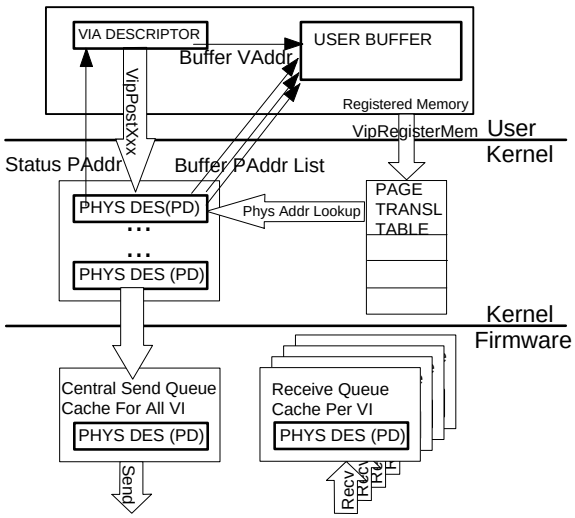

Figure 3.2: Sending and receiving messages using Physical Descriptors and address translation.

A flag in the NIC memory is set to indicate the existence of the queued descriptor(s) in the host memory. On the NIC side, a low watermark is associated with each RQC. If the amount of descriptors in the RQC goes below the low watermark and if the RQC has the queued flag set, then the firmware sends a refill interrupt to host which will dequeue the descriptors on the host and write as many of them as possible into the RQC. The low watermark is chosen such that the time required for processing a refill interrupt is less than the time it will take for arriving messages to deplete the descriptors in the RQC. Furthermore, when a new descriptor is posted, if the host finds other descriptors which have been posted earlier but not cached yet, it caches as many descriptors as possible into the NIC. The posting order of the descriptors is preserved during these operations.

Note that the operating system limitation on maximum registered memory size may be increased by taking advantage of the caching of descriptors in the NIC. In this scheme, we need to pin only the user buffers that have a corresponding descriptor cached in the NIC. And the remaining memory can be pinned on the fly as descriptors are cached. This will permit registering more memory than the amount of physical memory. However, the downside of pinning memory on the fly is the increased complexity of the device driver and a possible increase in message latency. To implement this scheme efficiently, the cached queues on the NIC need to be deeper and the low watermarks need to be higher so that page faults can be serviced in time before cached descriptor queues are depleted. We are currently working on such a scheme.

Centralized Doorbell and Send Queue

VIA specifies that each VI is associated with a pair of doorbells. The purpose of a doorbell is to notify the NIC of the existence of newly posted descriptors. Our NIC does not have hardware support for doorbells as stated before. Therefore we emulate the doorbells in the firmware. One approach for emulating doorbells is allocating space for each doorbell in the NIC memory and mapping this doorbell memory to the process' address space. The user application rings the doorbell by simply setting the corresponding bit in the NIC memory. To protect a doorbell from being tampered by other processes, doorbells of different processes need to be on separate memory pages

in the NIC. An issue in emulating doorbells is the cost of polling them in the NIC. Polling will add to the message latency with increasing number of processes and active VIs. Therefore, we decided to combine doorbells and send queues in a central place on the NIC.

Considering the fact that we go through the kernel for address translation as discussed in Section 3.2.2, combining send descriptors of all VIs in a central queue on the NIC makes more sense. We took such an approach. We call this queue as the Central Send Queue Cache (CSQC). Since descriptors go through the kernel, multiple processes can post them to the CSQC in an operating system safe manner. Effectively, the CSQC queue becomes the centralized doorbell queue. Changing the state of CSQC from empty to not empty is equivalent to ringing a doorbell. Similar to the RQCs, the CSQC is implemented as a FIFO circular buffer and it has a head and tail pointer. An advantage of a central send queue is that the firmware is required to poll only one variable, namely the tail pointer of the queue, thereby avoiding the overhead of polling of multiple VI endpoints. Situations where the CSQC is full or about to go empty is dealt using a mechanism similar to the one used for RQCs (as discussed in the previous subsection).

The VIA specifications provide a mechanism to put an upper bound on the number of outstanding descriptors associated with a particular VI. Enforcing this upper bound guarantees that no VI will suffer from starvation when using a shared queue in the NIC.

Immediate Data

We have also implemented the immediate-data mode of data transmission. On the receiving side, if the immediate data flag of a receive descriptor is set, a physical address in PD points to the immediate data field of the user VIA descriptor. On the send side, instead of writing a physical data segment address (PDDS) into the NIC, the immediate data itself is written. A flag in the control field of the PDCS is set to indicate that what follows the PDCS is the immediate data itself and not an address.

Since performing DMA operations for small messages is inefficient, we also experimented with sending messages of smaller than a certain size as if they were being sent in the immediate-data mode. In other words, instead of writing a physical address of the user buffer in the central send queue cache (CSQC), the host writes the message itself in CSQC.

Remote Direct Memory Access (RDMA)

In the VIA RDMA mode of transfer, the RDMA initiating node specifies a virtual address at the target node's memory. The issue here is how to translate this virtual address to the physical address on the target NIC. We do not expect the caching of physical addresses to have as high hit rates for RDMA as for send/receive operations. Because for send/receive operations, descriptors are consumed in sequential order hence caching works well due to the prefetching of descriptors. However for RDMA, the initiating node can specify arbitrary virtual addresses at the target node memory. Thus predicting next physical address in RDMA is difficult.

In our RDMA design, this address translation problem can be solved in two different ways: 1) In order to prevent stalling the reception of RDMA packets, the NIC can DMA all RDMA packets directly into a kernel buffer (whose physical address is known to the NIC). Then, these messages can be copied to the target user buffer by the host processor. 2) The NIC can do the TLB lookup from host memory by DMA upon message reception. The second method, as mentioned earlier in Section 3.2.2, may have a problem of stalling message reception momentarily and cause messages backing up into the network. Thus, the first method looks attractive for implementation and we

are currently incorporating this method to our implementation for supporting RDMA operations efficiently.

3.3 Performance Evaluation

In this section we present the communication latency and bandwidth measurements obtained in our experimental testbed. We discuss various aspects of our implementation and provide a detail evaluation of different components of the FirmVIA. We also discuss the impact of several hardware improvements on the performance of FirmVIA.

3.3.1 Experimental Setup

The results presented in this section were obtained on a cluster of PCs with 450 MHz Pentium III processor nodes and 100 MHz system bus. Each node has 128 MB of SDRAM, 16 KB of L1 data cache, 16 KB of L2 instruction cache, and 512 KB of L2 cache. Each node has a 33 MHz/32-bit PCI bus and runs the NT 4.0 operating system. Table 3.1 shows the cost of elementary operations on this system. For all experiments, the maximum switch packet payload was set to 1032 bytes (1024 bytes of payload plus 8 bytes of VIA control header) unless otherwise stated.

Operation	**Cost**
Host PIO Write	0.33 μs/word
Host PIO Read	0.87 μs/word
NT FAST IO (user/kernel switch)	2.27 μs
NT Interrupt Latency	10-17 μs

Table 3.1: Cost of Basic Operations

3.3.2 Latency

We determined the message latency as one half of the measured roundtrip latency. The test application sends a message to a remote node's test application. The remote node replies back with a message of the same size. Upon receiving the reply, the initiating node repeats the ping–pong test and repeats it large number of times so that the overhead of reading the timer is negligible. We aligned the send and receive buffers to the beginning of physical pages so that buffers crossing page boundaries do not influence latency measurements for small messages. Performance effects of crossing page boundaries are discussed in Section 3.3.3. The test application uses the *VipPostSend* and *VipPostRecv* function calls for posting send and receive descriptors. Messages were received using *VipRecvDone* function call and by polling on the completion status of the posted receive descriptors.

The latencies for different message sizes are shown in Fig. 3.3. The one-way latency for four-byte messages was observed to be only $18.2\mu s$.

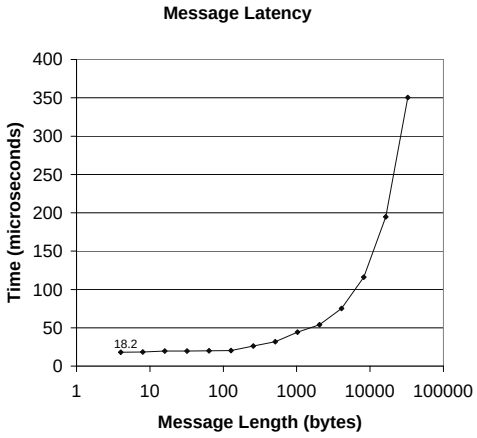

Figure 3.3: Message latency for different message sizes.

Components of Latency

To find out where and how the measured time is spent, we instrumented the firmware and the device driver to measure different components of latency. Each component was measured several times and the minimums were recorded. Due to this method of recording, the summation of the delays of different stages of transfer is slightly lower than the measured one-way latencies, shown in Fig. 3.3. However, such a study provides insight to our implementation. Figure 3.4 illustrates the time spent in different stages of data transmission from the source node data buffer to the destination node data buffer. It can be seen that the time spent by the host processor is independent of the message size (for the range shown in the figure) which is a result of the zero–copy implementation. Breakdown of the host overhead is given in Table 3.2. Note that the PIO cost of writing a physical descriptor (PD) into the NIC is the time for writing five words (three words for the PDCS and two words for the PDDS).

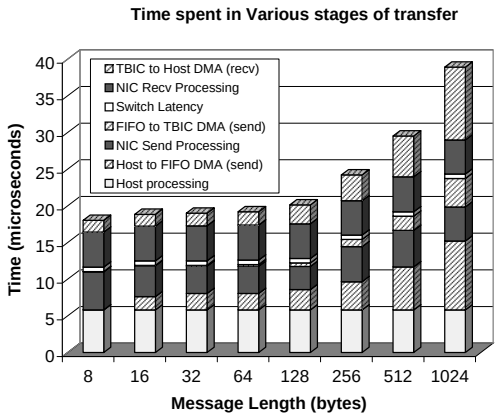

Figure 3.4: The breakdown of short message latencies.

The host memory to Send-FIFO transfer time is shown as the second bar from the bottom in Fig. 3.4. The cost of the NIC firmware processing a physical descriptor (PD) is shown as the third

Operation	Cost	
NT FAST IO (user/kernel switch)	2.27	μs
PIO write 5 words of PD to NIC	1.65	μs/word
Processing in user space	0.27	μs
Processing in kernel space	1.63	μs
Total	5.82	μs

Table 3.2: Breakdown of the Host Overhead

bar from the bottom. It can be observed that this cost remains almost constant for messages up to 128 bytes. There is a slight increase in the NIC send processing delay for larger than 128 byte messages and this can be attributed to the firmware sequencing effect as discussed in Section 3.3.2. It is to be noted that for messages of eight bytes or less, the data is transferred to the NIC through Programmed IO (PIO) instead of using DMA, as described in Section 3.2.2. We discuss the performance tradeoff between PIO and DMA in more detail later.

After the message is transferred by the LHS DMA engine into the NIC Send-FIFO, it is sent out by the RHS DMA engine into the TBIC2. This DMA transmission is performed at a rate of 264 MBytes/s and it is shown as the fourth bar from the bottom. Note that as soon as the first word of data is written into it, the TBIC2 starts sending it out to the network. The SP switch has less than $0.3\mu s$ latency. This overhead and the overhead of the injection and consumption of one word to/from TBIC2 at the sending and receiving sides are shown as the fifth bar. Finally the cost of processing the received message and transferring the message by DMA into the user buffer is shown as the two top most bars.

It is to be noted that on the receiving side the LHS DMA and RHS DMA engine receive operations are almost completely overlapped. While the RHS DMA engine is transferring message payload from the TBIC2 buffer to the Recv/CMD-FIFO, the LHS DMA engine is transferring that payload from the Recv/CMD-FIFO to the host memory. Since the PCI bus bandwidth is less than that of the NIC internal bus, the cost of data transmission from TBIC2 to the Recv/CMD-FIFO is masked and does not appear as a separate item in Fig. 3.4. This behavior is different on the send side because for the message (or more precisely the payload of a packet) to be transferred from Send-FIFO to TBIC2, the hardware requires the whole payload to be present in the Send-FIFO. Thus two separate DMA operations (bars 2 and 4) appear in Fig. 3.4 for sends. The NIC send and receive processing costs also contain the time for marking the VIA descriptors in host memory as complete.

PIO vs. DMA

As discussed in Section 3.2.2, for short messages, the message itself (instead of its address) can be directly written into the central send queue cache (CSQC) to avoid the startup cost of DMA. Figure 3.5 illustrates the cost of NIC send overhead for short messages. It can be observed that for messages of 16 bytes or less, the NIC send overhead using PIO operation is less than that of the DMA operation. The savings were less than what we anticipated. Closer examination of the firmware revealed that the C compiler for firmware was not producing efficient instructions to move

the message in the SRAM. Another constraint limiting the use of PIO for transmitting the data was turned out to be the additional cost of caching the descriptors into the NIC (not shown in Fig. 3.5). The extra space required in the CSQC was another constraint. Thus, we chose to use PIO for messages of eight bytes and less only.

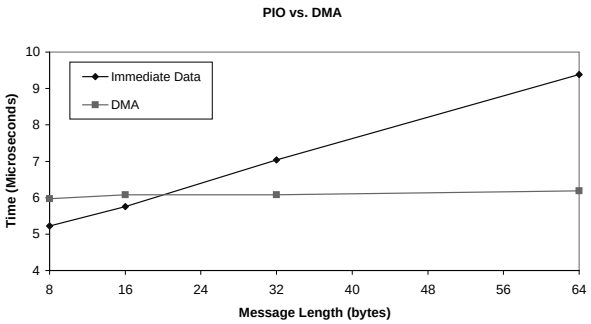

Figure 3.5: Delays for sending data with PIO vs. DMA. Descriptor processing delay is included.

3.3.3 Bandwidth

To measure the bandwidth, we sent messages from one node to another node for a number of times and then waited for the last message to be acknowledged by the destination node. We started the timer before sending these back to back messages and stopped the timer when the acknowledgment message for the last sent message was received. The number of messages was large enough to make the acknowledgment message delay negligible compared to the total measured time.

The peak measured bandwidth for different message sizes is shown in Fig. 3.6. The maximum observed bandwidth is 101.4 MB/s. Note that the half-bandwidth is achieved for a message size of 864 bytes.

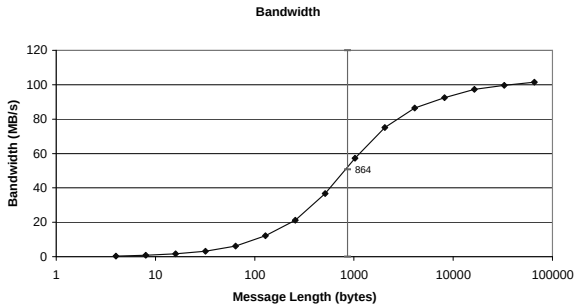

Figure 3.6: Measured bandwidth for different message sizes. The half-bandwidth is achieved for 864-byte messages.

The Bandwidth Bottleneck

The theoretical maximum bandwidth of PCI bus is 132 MBytes/s. This is less than the SP Switch link uni–directional bandwidth (150 MBytes/s) and the NIC internal bus bandwidth (264 MBytes/s). This led us to believe that the longest stage of the pipeline for sending and receiving messages is the PCI bus on which data is transferred between the host memory and the NIC FIFO buffers (Fig. 3.1). To determine the sustained bandwidth of the PCI bus we measured the DMA bandwidth from the host memory to the NIC FIFO and vice versa. Figure 3.7 shows the measurement results. Note that these numbers do not include any VIA processing overhead. It is observed that for transfer size of 1 KB and more the cost of DMA from the NIC Recv/CMD–FIFO to the host memory is more than that of moving the same amount of data in the opposite direction. Therefore, we conclude that the maximum bandwidth of our VIA implementation is limited by the receive side.

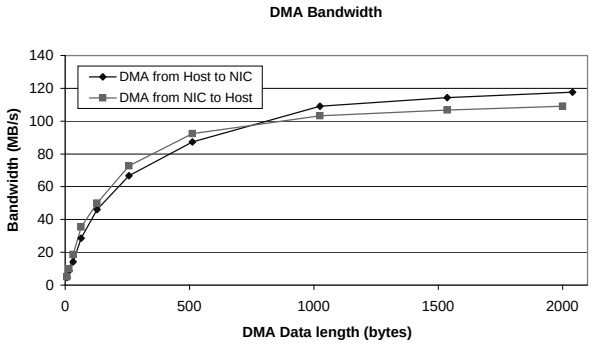

Figure 3.7: The raw PCI DMA bandwidth.

Effect of Packet Size

As mentioned in Section 3.1.2, the maximum payload of a switch packet is 2040 bytes. Since each VIA packet has a eight-byte software header, the payload for user data is 2032 bytes at maximum.

Figure 3.8 illustrates the effect of varying the packet size on the maximum possible bandwidth. It is seen that the maximum bandwidth is achieved for the switch packet size with a user payload of 1024 bytes. Increasing the size of user payload beyond 1KB does not increase the bandwidth. In fact, there is a slight decrease in the bandwidth for larger payloads. This can be attributed to the 4 KB size of the NIC Send–FIFO. When the maximum user payload in switch packets is set to 1KB, the Send–FIFO can be filled with exactly 4 switch packets worth of data (4×1 KB). When using larger payloads the Send–FIFO can take only 3 or 2 switch packets worth data. Fewer packets reduce the benefits of pipelining. Consider the fact that the PCI bandwidth is less than that of the internal bus and the switch links. This may lead to the situation where the NIC is ready to send out the next packet but the packet hasn't been completely transferred in to the Send–FIFO yet.

Figure 3.8 illustrates another effect where increasing the size of the user payload from 1000 bytes to 1024 increase the bandwidth significantly. This has to do with our firmware implementation. To simplify the firmware we structured it so that each LHS DMA initiation on the NIC results in one

switch packet sent out to the network. This means that if a section of a message buffer is crossing a physical page boundary then it is sent in two separate switch packets. For example, consider the case of a 5000 byte page aligned message to be sent. With 1000 byte packet payload, four 1000 byte packets followed by a 96 byte packet, followed by a 904 byte packet is sent (Total 5000 bytes and 6 switch packets). With 1024 byte packet payload, four 1024 byte switch packets, followed by a 904 byte packet is sent (Total 5000 bytes and 5 switch packets.) Thus for long messages the NIC has $\frac{5}{6}$ less DMA initiation overhead than for 1024-byte payloads and this results in higher bandwidth in Fig. 3.8.

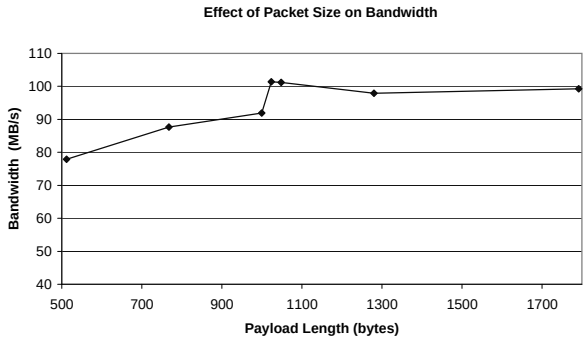

Figure 3.8: Effect of packet size on the bandwidth.

3.3.4 Estimated Performance on Future Systems

In this section, we discuss the impact of several hardware improvements on the performance of FirmVIA. In particular, we consider the impact of increasing the speed of host and NIC processors, larger NIC memory, and hardware support for doorbells on the latency of short messages. We also study the impact of using write combining. Write combining is a PCI feature that NIC may or may not support. When NIC supports write combining, multiple PCI writes to sequential addresses are in the form of one bus arbitration and one address cycle followed by multiple data cycles. This is a more efficient form of data transfer that does not require one bus arbitration and one address cycle for each word transferred. In this section, we do not consider the effect of improvement in networking technologies.

Not having the necessary hardware support for doorbells and having a limited amount of memory on the NIC were the major reasons for going through kernel to post send and receive descriptors in FirmVIA. Therefore, having an efficient hardware support for doorbells and a reasonably large NIC memory which can store the TLB eliminates the need for using system calls during send and receive operations. In such a system, descriptors can be cached into the NIC memory directly without being converted to PDs in the kernel space. Furthermore, supporting write combining and using wider and faster PCI bus can reduce the time needed for transferring VIA descriptors from host memory to NIC memory significantly.

Based on the detailed breakdown of the host and NIC overheads for FirmVIA (presented in Section 3.3.2) and in order to study the effect of described features, we estimated the oneway latency in a Netfinity cluster with the following added features: 1) Hardware support for doorbells which eliminates the need for polling at the NIC when separate send queues for VIs are used, 2)

The NIC memory is large enough to hold the entire TLB, 3) 64-bit/66-MHz PCI bus, and 4) The NIC internal bus is assumed to be working at 50 MHz resulting in an internal bandwidth of 400 MBytes/s.

Figure 3.9 shows the estimated oneway latency of 16-byte messages on such a system when the speed of the NIC processor varies. It can be observed that the NIC speed has a significant effect on the latency. With a NIC processor running at 250MHz, the oneway latency is estimated to be only $9.27\mu s$. The impact of the host processor speed was evaluated by estimating the latency while the host processor speed varied from 450MHz to 2 GHz. The estimated improvement was found to be negligible. Figure 3.10 shows the effect of changing the speed and width of the PCI bus as well as using write combining. It can be observed that the impact of these features are significant. If the PCI bus is improved from 32-bit/33-MHz to 64-bit/66-MHz and write combining is also used, the oneway latency is reduced from $13.73\mu s$ to $9.27\mu s$.

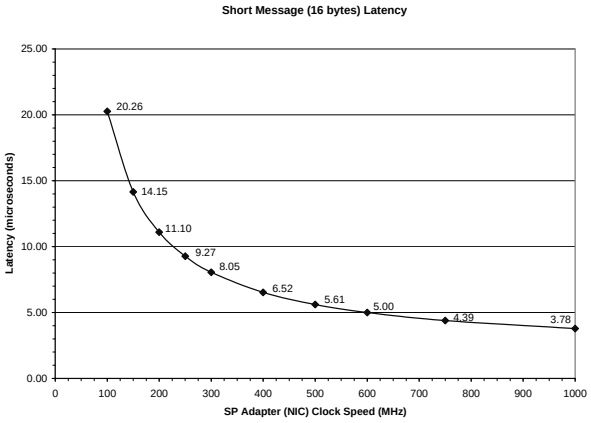

Figure 3.9: Impact of NIC cpu speed.

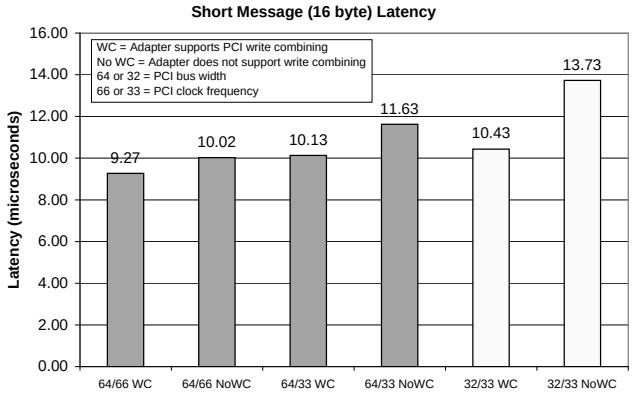

Figure 3.10: Impact of PCI speed and width.

Communication System	L	BW	Host	Network	OS
Berkeley VIA [25]	23	29.3	USPARC 167	Myrinet (SBus)	Solaris 2.6
Berkeley VIA [25]	26	53.1	Pentium 300	Myrinet (PCI)	NT 4.0
Berk. VIAv1 [24]	≈ 24	≈ 64	Dual PII 400	Myrinet (PCI)	NT 4.0
Berk. VIAv2 [24]	≈ 32	≈ 64	Dual PII 400	Myrinet (PCI)	NT 4.0
Giganet VIA [48]	≈ 10	≈ 70	Xeon 450	cLAN	NT 4.0
Servernet VIA [48]	≈ 100	22	Xeon 450	ServerNet	NT 4.0
MVIA [3]	19	60	SMP PII 400	GBit Ether	Linux 2.1
MVIA [3]	23	11.9	SMP PII 400	100MB Ether	Linux 2.1
FirmVIA	18.2	101.4	PIII 450	SP (PCI)	NT 4.0

Table 3.3: Latency (L in μsec) and Bandwidth (BW in $MBytes/sec$) Results of Different VIA Implementations

3.4 Related Work

Performance results of several VIA implementations are summarized in Table 3.3. The Berkeley VIA (Version 1) [25] is one of the first software implementations of the VIA. (This implementation is a partial implementation of VIA mainly done to obtain a better insight on different aspects of the implementation of the VIA.) In this implementation, a memory page on the NIC memory has been used for the implementation of a pair of doorbells. The doorbells for send queues are polled for finding outstanding send descriptors. This polling is expensive and increases linearly with the number of active VIs. The Berkeley VIA does not perform any caching of descriptors. In other words, for sending messages, NIC has to access the host memory twice: once for obtaining the descriptor and once for obtaining the data itself. In this implementation, only a subset of the descriptors are moved between the host and the NIC to reduce the high cost of transferring the descriptors. Not caching the descriptors have a greater impact at the receiving side. During receive operations it is required that the interface momentarily buffers or blocks the incoming message to retrieve the destination receive descriptor. One of the systems used for performance evaluation consisted of a pair of 300 MHz Pentium processors with a 33MHz PCI bus and 128 MB of memory running the Windows NT operating system. For the network, Myricom's Myrinet M2F [22] with the LANai 4.x-based network interface card were used. The minimum reported latency for a PCI-based system is 26μs. The bandwidth results are reported only for messages of up to $4K$ bytes. The peak bandwidth of 425 Mbits/s (53.13 MBytes/s) on the PCI-based system is measured. Different extensions to the original implementation have been discussed: descriptorless transfers and merged descriptors. It is reported that supporting these extensions increased the complexity of the firmware and slowed down even the standard descriptor model.

The Berkeley VIA (Version 2) [24] is based on the Berkeley VIA (Version 1) implementation and adds memory registration and increased VI/user support. In this implementation, each memory page on the NIC can support up to 256 pairs of doorbells that belong to a single process. For the address translation a buffer with limited size on the NIC is used for the TLB. If the size of registered memory is bigger than what can be supported with this table, the translation of

some portions of the registered memory won't be present in the NIC TLB. In these cases the host memory is accessed to obtain the translation. The location of the host buffers holding the complete translations for registered memory regions are known to the NIC. PCs with 400 MHz processors, running Windows NT 4.0 and interconnected by the Myrinet M2F switches were used to obtain the bandwidth and latency of this improved version of the VIA implementation. The increased latency of short messages due to the the new address translation mechanism was about 6 μs. The latency for the case where TLB miss happens only at the first use of VI was shown to be as high as 34 μs (Fig. 7 of [24]). When the misses happen all the time, the latency can increase up to 40 μs. The complexity of the new firmware contributed to the increased latency which is what we tried to avoid by using Physical Descriptors. The maximum peak bandwidth was reported as 64 MBytes/s. The half-bandwidth was achieved by messages longer the 1000 bytes. Unlike our implementation, no caching of descriptors is being used in this study. The new address translation mechanism which is essentially added in response to the limited resources available on the NIC (the similar restriction that we faced in our system) increases the latency by more than 6μs. In contrast, our implementation pays only the 2.27μs cost of the Fast IO dispatch which also gives us the chance of using central send queue on the NIC to avoid the polling of send doorbells.

Speight *et al.* [48] study the performance of GigaNet cLAN [1] and the Tandem ServerNet VIA implementations. The platform used in this study consists of a set of 450 MHz Xeon processors with a pair of 33 MHz, 32-bit PCI busses running NT 4.0. While the cLAN provides hardware support for the VIA implementation, ServerNet emulates VIA in software. The peak measured bandwidth of the VIA implementations is around 70 MBytes/s for the cLAN and just above 20 MBytes/s for the ServerNet (Fig. 2 of [48]). The maximum link bandwidths of cLAN and ServerNet switches are 125 and 50 MB/s/link, respectively. The reported small message latency for the cLAN is 24 μs for the cLAN and around 100 μs for the ServerNet. It should be noted that for the latency measurements in this study the blocking VIA calls are used for detecting the completion of the receive operations. The native VIA latency of the cLAN hardware is reported to be around 10 μs [48].

3.5 Summary

In this chapter, we presented an experimental VIA implementation on the SP Switch connected NT Cluster. The design issues and the details of the implementation have been discussed. We presented the notion of Physical Descriptors and showed how Physical Descriptors can be used to efficiently implement virtual-to-physical address translation for network interface cards with limited amount of memory. We also showed how caching descriptors can be used to provide a zero copy communication system. We presented a mechanism to implement the doorbells efficiently in the absence of any hardware support. A central send/doorbell queue in the NIC has been used to eliminate polling of multiple VI endpoints. Our design carefully distributes the work between the host and the NIC for the best performance. Our VIA implementation performs comparably or better than the other VIA implementations including hardware and software implementations.

CHAPTER 4

VIBE: A MICROBENCHMARK SUITE FOR EVALUATING VIRTUAL INTERFACE ARCHITECTURE (VIA) IMPLEMENTATIONS

As discussed in previous chapters, VIA has different components (such as doorbells, completion queues, and virtual-to-physical address translation) and attributes (such as maximum transfer unit and reliability modes). The VIA specification is not very strict with respect to the way these components need to be implemented. Modern computing systems are having programmable Network Interface Cards (NICs). The availability of these NICs lead to many alternative ways to implement the VIA components [13] by dividing the operations involved in message transfers between the host and NIC processors. The current VIA implementations demonstrate some of these alternatives.

As different VIA implementations are emerging, it is increasingly becoming a challenging task about how to report the VIA-level performance results *accurately*. The standard latency (ping-pong test) and bandwidth (consecutive sends) measurements are sensitive to the way different components of the VIA are implemented. For example, performing the address translation at the host vs. the NIC will lead to different performance results. Similarly, a latency test where buffers are reused will have significant difference in performance compared to the test where buffers are not reused at all. Implementation methodologies for other VIA components such as doorbell and completion queues also have significant impact on the performance. In the absence of any standard way to report VIA results, it is increasingly becoming difficult to understand the strengths and weakness of a VIA implementation from the standard latency and bandwidth tests.

As VIA implementations are being available on multiple networks, researchers and developers are also engaged in developing better implementations of higher-level layers (such as MPI [39], sockets [46], distributed shared memory [16]). Designing these higher level layers requires an in-depth understanding of the performance, strength, and weakness of the underlying VIA implementation. For example, knowing the impact of virtual-to-physical address translation can help higher layer developer to optimize buffer pool and memory management implementations. Understanding the impact of multiple open VIs (between a set of processes) on the latency can provide a higher layer developer insight about the number of VIs to be used in an implementation and analyze scalability studies.

Currently, there is no framework for 1) evaluating different design choices and for obtaining insight about the design choices made in a particular implementation of VIA, 2) study their impact on the overall performance, and 3) study the implication of the design choices and performance on designing higher layers. The *LogP* [28] model attempts to capture the major characteristics of communication subsystems (including the share of the host processor in data transfer operations) with a few parameters, namely, L (Latency), o (overhead), g (gap), and P (the number of processors). However, this model is not sufficient to provide answers to the three questions raised at

the beginning of this paragraph. This leads to a challenge whether a micro-benchmark suite can be designed to evaluate and compare different VIA implementations and provide guidelines to the higher-layer developers.

This work takes on such a challenge. We propose a new micro-benchmark suite called *Virtual Interface Architecture Benchmark (VIBe)*. This suite is divided into three major categories: 1) Non-Data Transfer related, 2) Data Transfer related, and 3) Higher-Level Layer related. Under the first category, we include micro-benchmarks for measuring the cost of several basic non-data transfer related operations: creating/destroying VIs, establishing/tearing down VIs, memory registration/deregistration, and creating/destroying completion queues. The second category consists of several data-transfer related micro-benchmarks. A set of micro-benchmarks under this category are designed in a systematic manner so that only one VIA component (such as address translation, multiple data segments, completion queues, multiple VIs) is changed at a time. This clearly brings out the strengths and limitations of a given VIA implementation with respect to that component. A set of other micro-benchmarks are also included in this category to study the impact of asynchronous message handling, RDMA operations, maximum transfer size, reliability, and sender pipeline lengths. For each of these data transfer related micro-benchmarks, we include latency, bandwidth, and CPU utilization numbers for transferring messages of varying size. The third category focuses on micro-benchmarks related to higher-level layers. In this chapter, we include a micro-benchmark corresponding to the client-server environment. In future, we plan to include micro-benchmarks related to other higher-level layers (such as distributed memory, distributed shared memory, and Get/Put) under this category. The micro-benchmarks under these three categories provide many insights to a VIA developer to optimize its implementation as well as provide design guidelines for the developer of a higher-layer to develop an optimized implementation on a given VIA layer. It should be noted that the impact of the performance of user-level networking protocols on that of higher layers has been studied in [37]. In contrast, the main focus of our work has been the evaluation of different implementations of VIA.

The chapter is organized as follows: In Section 4.1, we discuss in detail the motivations behind developing a micro-benchmark suite for VIA. We introduce the VIBe micro-benchmark suite in Section 4.2. The micro-benchmarks are evaluated on two implementations of VIA: Berkeley VIA [25, 24] (on Myrinet [22] connected Linux machines) and MVIA [3] (on Gigabit Ethernet connected Linux machines). The evaluation results are presented in Section 4.3. Through these evaluations we show how the VIBe suite can provide insights to the details of a VIA implementation and help higher layer software developers to develop their layers on top of the VIA implementation. In Section 4.4 we present our conclusions.

4.1 Motivation behind a Micro-benchmark Suite for VIA

As indicated above, a particular VIA implementation has several components and attributes. Developer of a VIA implementation may choose to implement these components and attributes in a certain manner based on the characteristics of the underlying node, operating systems, NIC, and network. For a given implementation, the developer may report latency and bandwidth results under an idealized condition (i.e, sending messages from the same buffer, opening only one open VI between sender and receiver, not implementing completion queue). However, in reality, a higher-level layer (MPI, socket, distributed shared memory, or client-server) may use the underlying VIA implementation under different scenarios: multiple VIs between a set of processes, send messages from different buffers, etc. Thus, it leads to the following questions:

1. How do we accurately report VIA results so that different VIA implementations can be compared with respect to their strengths and weaknesses in a standardized manner?

2. For a given VIA implementation, how to report performance numbers with respect to their components and attributes so that,

   (a) A VIA developer can get enough insights to optimize his implementation.

   (b) A higher-layer developer can get insights to a VIA implementation and come up with appropriate strategies (optimizing buffer pool, memory management implementation, scalability study, etc.) for implementing the higher layer.

By using a set of micro-benchmarks and characterizing the performance of the communication system under different conditions, insight about the internal implementation of VIA implementations can be obtained. This insight can be used as a guideline for the higher-layer developers for improving the performance of their layers. In the next section, we describe a set of such micro-benchmarks, defined as VIBe suite.

4.2 VIBe Micro-benchmark Suite

In this section, we discuss the VIBe micro-benchmarks. While developing these micro-benchmarks, we considered different design alternatives that can be used for different components of VIA and devised the methods to measure the impact of these particular decision choices. Besides quantifying the performance seen by the user under different circumstances, the benchmarks can also be useful to identify how much time is spent in each of the components in the implementation, and pinpoint the bottlenecks that can be improved. This set of benchmarks cover most important aspects of VIA implementation.

The micro-benchmarks can be categorized into three major groups: non-data transfer related micro-benchmarks, data transfer related micro-benchmarks, and higher-level layer related (client/server) micro-benchmarks. These groups and related micro-benchmarks are presented and discussed in detail in the rest of this section.

4.2.1 Non-Data Transfer Related Micro-Benchmarks

In this category there are four micro-benchmarks which measure the costs of basic non-data transfer VIA operations: 1) creating/destroying VIs, 2) establishing/tearing down VI connections, 3) memory registration/deregistration, and 4) creating/destroying completion queues.

Before any VIA data transfer can occur VIs on end nodes should be created. Furthermore, a connection between the VIs should be established. Therefore, it is important that the cost of creating/destroying VIs and establishing/tearing down connections are evaluated. All data transfers should be from/to buffers in registered memory regions. Therefore, evaluating the performance of memory registration/deregistration is important. Completion queues are frequently used in VIA applications. All of these parameters have a significant effect on the scalability of the system, suitability of the communication subsystem for large and dynamic run-time systems.

4.2.2 Data Transfer Related Micro-Benchmarks

The micro-benchmarks in this category evaluate the performance of VIA operations used for transferring data by measuring the latency, CPU utilization, and bandwidth under different conditions. For measuring latency, the standard ping-pong test is used. To measure the bandwidth, messages are sent out repeatedly from the sender node to the receiver node for a number of times and then the sender node waits for the last message to be acknowledged. The CPU utilization is measured by using the *getrusage* function. In the rest of this section we discuss the micro-benchmarks in this category in detail.

Base Latency(L_{Base}), Utilization(U_{Base}), and Bandwidth(B_{Base})

These benchmarks are used to find the latency, CPU utilization, and bandwidth for our base configuration. The base VIA setup used for these micro-benchmarks has the following properties: 1) 100% buffer reuse (all messages are sent from one single send buffer and are received in one single receive buffer), 2) one data segment, 3) no completion queue, 4) one VI connection, 5) no notify mechanism.

For measuring latency, the standard ping-pong test can be used. In this micro-benchmark, two VIs are created on two nodes and a connection is established between them. The latency is measured by measuring the time to send a number of messages (with a particular message size) from one node to another node. Each time the receiving node sends back a message of the same size. The sender node sends a new message only after receiving a message from the receiver. The number of messages being sent back and forth is long enough to make the timing error negligible. The same user buffer (which is in a registered memory region) is used as the send and receive buffers. Before posting a send descriptor, a receive descriptor is posted to avoid situations where a message arrives at a node before its corresponding receive descriptor is posted. This test is repeated for different message sizes. We report the results for two cases where polling or blocking is used for checking the completion of data transfers. The CPU utilization is measured by using the *getrusage* function in the latency micro-benchmark.

To measure the bandwidth, messages are sent out repeatedly from the sender node to the receiver node for a number of times and then the sender node waits for the last message to be acknowledged. The time for sending these back to back messages is measured and the timer is stopped when the acknowledgment of the last message is received. The number of messages being sent is kept large enough to make the time for transmission of the acknowledgment of the last message negligible in comparison with the total time.

In the following four micro-benchmarks, we change only one of the parameters of the setup to isolate and evaluate the impact of each parameter.

Impact of Virtual-to-Physical Address Translation (L_{AT}, U_{AT}, and B_{AT})

A very important component of any low-level communication subsystem is the virtual-to-physical address translation. Different methods for performing the address translation are discussed in [13]. Depending on whether the host or the NIC performs the translation and whether the address translation tables are stored in the host memory or in the NIC memory, four possible VIA implementations are possible. Since the virtual-to-physical address translation is required (in most systems) for transferring data between the host memory and the NIC memory in each data transfer, the translation method used in an implementation may have a significant effect on the performance

of the communication subsystem. Studying the impact of virtual-to-physical address translation can help higher layer developer optimize buffer pool and memory management implementations.

In order to evaluate the effect of the virtual-to-physical address translation on the performance of the communication subsystem, simple latency, CPU utilization, and bandwidth tests can be used. These micro-benchmarks are similar to the one used for measuring the base latency, CPU utilization, and bandwidth with the only difference being that different send and receive buffers are used in different iterations of the experiments. Similar to the base micro-benchmarks, two VIs are created on two nodes and a connection is established between them. Send and receive buffers for all the iterations of the experiments are allocated and registered before the measurements begin. While the L_{Base} represents the latency observed when the same send and receive buffers are used in all the iterations of the ping-pong experiment, L_{AT} represents the latency observed when a different send and receive buffer is used in each iteration. The difference between L_{AT} and L_{Base} corresponds to the cost of address translation.

Impact of Multiple Data Segments (L_{MDS}, U_{MDS}, and B_{MDS})

In most of the networking protocols, a packet is assembled from a single buffer. This requirement may result in extra data copies to put different segments of a message in a contiguous buffer. However, VIA provides a way to assemble packets using data from different buffers (gather). Similarly, received messages can be scattered into multiple buffers (scatter). In a VIA descriptor, multiple data segments may be present. Each VIA data segment has pointer to a buffer along with the size of the data to be sent from or received to that buffer. For writing layered protocols over VIA, this information can help in a design decision such as whether to use header coalescing or not.

In order to evaluate the performance of VIA implementations with respect to this feature, the latency, CPU utilization, and bandwidth tests are used in a situation where send and/or receive buffers consists of multiple data segments. For every message size, the number of segments which constitute the message is varied such that the effect of using multiple segments can be evaluated. For a given number of segments the difference between L_{Base} and L_{MDS} gives the overhead of using multiple segments.

Impact of Completion Queues (L_{CQ}, U_{CQ}, and B_{CQ})

In VIA implementation, a process can figure out the completion of a send or a receive operation by checking the completion queue in VIA. A process may have multiple VIs and it may choose to associate the work queues of these VIs with a single completion queue. By using this mechanism, the process is relieved form checking each VI to see if any operation has completed. By polling the completion queue (or blocking on it), a process can know if any of the operations has been marked as complete. Many applications require to receive messages from different nodes without the order of the receptions being important. Completion queues in VIA provide an easy method for doing so. Therefore, it is important to see how using CQs affect the latency of data transfers. This can allow an application developer writing multi-threaded applications using multiple VIs to estimate the cost of using a CQ for checking the completion of operations on multiple work queues.

In order to quantify the cost associated with using completion queues, the following micro-benchmarks can be used. In these benchmark, the completion of send and/or receive operations in latency, CPU utilization, and bandwidth tests are checked through the completion queue associated with the corresponding send and/or receive queues. Similar to the base micro-benchmarks, a VI connection established between a pair of VIs on the sending and receiving nodes are used for

performing the tests. The difference between L_{Base} and L_{CQ} indicates the overhead of using completion queues.

Impact of Multiple VI (L_{MVIs}, U_{MVIs}, and B_{MVIs})

Since VIA is a connection-oriented communication subsystem, for any pair of processes which want to communicate with each other, a VI connection should be established between two VIs of these processes. Therefore, in many applications, there are number of active VIs at each process (node). Therefore, it is important to see whether the number of active VIs on the nodes which are exchanging data has any effect on the latency, CPU utilization, and bandwidth of messages. This benchmark can provide insights into scalability of VIA implementation.

In order to evaluate the performance of the communication subsystem when a number of VIs are active, a new set of micro-benchmarks can be used. These micro-benchmarks are similar to the base micro-benchmarks with the difference being that before the tests are executed, multiple VIs are created by both of the participating processes. Then, a VI connection established between a pair of these processes is used for performing the ping-pong test. The number of active VIs on each side of the communication is varied and the same experiment is repeated such that the effect of number of VIs on the latency, utilization, and bandwidth can be quantified.

Asynchronous Messages (L_{Async}, U_{Async}, and B_{Async})

The latency of asynchronous messages can have a significant effect on the overall performance of many applications. VIA provides a Notify mechanism for handling asynchronous messages. When a notify procedure is associated with a queue and is activated, this procedure is invoked upon completion of an operation on that queue. Depending on how this mechanism is implemented the performance varies significantly [16].

In order to quantify the cost associated with using the Notify mechanism, the Base latency micro-benchmark is modified such that upon completion of every receive operation, a user procedure is executed. This procedure is responsible for sending a message back to the other node.

RDMA Operations (L_{RDMA}, U_{RDMA}, and B_{RDMA})

In addition to the Send/Receive data transfer mode, VIA provides support for RDMA operations. The implementation of the RDMA write operation is compulsory while that of RDMA read operation is not. This micro-benchmark evaluates the performance of the RDMA write operation.

Impact of Maximum Transfer Size (L_{MTS}, U_{MTS}, and B_{MTS})

VIA architecture allows different attributes to be attached to a VI. One of the attributes is Maximum Transfer Size (MTS). MTS represents the maximum amount of data that can be transferred using a single descriptor. A VIA implementation can be optimized for a certain range of MTS. It is essential to study the impact of MTS on the performance. Furthermore, studying the impact of maximum transfer size helps in assessing the cost of fragmentation/reassembly of application buffers.

Impact of Reliability Levels (L_{RL}, U_{RL}, and B_{RL})

The three levels of communication reliability supported by VIA are: Unreliable Delivery, Reliable Delivery, and Reliable Reception. A VIA implementation can guarantee these reliabilities

by using either hardware or software or combination of hardware/software. The use of different reliability levels can have significant impact on the design, implementation, and performance of the layers built on VIA. For example, the use of reliable delivery VIs can free the higher layer developers from handling out-of-order segments and retransmissions. Efficient native hardware support for reliable delivery can lead to lighter weight high level software layer such as stream sockets over VIA [46]. Thus, for a given VIA implementation, it is very important to study the impact of reliability levels on the performance.

Impact of Sender Pipeline Length (B_{SPL})

The VIA separates the data transfer initiation (post operations) from the data transfer completion (polling and blocking operations). This enables asynchronous data transfers. In [46], it was shown that deferred dequeuing of send descriptors can improve the performance of software layers built on VIA. Thus, the number of send operations that can be pipelined before checking for completion can have significant impact on the performance and CPU utilization of the system. Furthermore, the number of send operations that can be initiated in a burst can provide insights into the cost of simple credit-based flow control schemes built over VIA [46].

4.2.3 Client/Server Micro-Benchmarks

Cluster of servers connected by a SAN are being deployed today to provide reliable and scalable Internet services. The nodes within this cluster perform client/server like communications. In order to evaluate the performance of VIA in this type of environments, a set of micro-benchmarks is presented in this subsection.

Request/reply type of communication is performed in distributed object computing and RPC-like environments. A transaction test that roughly approximates synchronous request/reply is used as a micro-benchmark here. In this micro-benchmark, two VIs are created on two nodes and then a connection is established between them. The client sends some amount of bytes as a request and receives some number of bytes as a response. The client sends a new request only after receiving the entire reply from the server. Two different buffers: one for the request and the other one for the reply, are used. For experiments, the reply size is varied for a fixed request size. The number of transactions/second measured by this micro-benchmark relates to the number of RPCs or method calls/second.

A multi-threaded server version of the above micro-benchmark is described next. The server in this micro-benchmark handles multiple requests from multiple clients concurrently. A worker thread on the server provides response to a client's request. Different worker thread models such as master/slave, and worker pool will be tested. This micro-benchmark will provide insights into the scalability of the VIA and the multi-threading related issues with VIA.

4.3 Performance Evaluation

In this section, we show how our benchmarks can be used to evaluate two available implementations of VIA, namely Berkeley VIA from University of California, Berkeley [25] and MVIA from NERSC at Lawrence Berkeley National Laboratory [3]. First, we discuss our experimental testbed and the studied VIA implementations. Then, we present the results obtained from running a couple of micro-benchmarks from VIBe on our testbed. For detailed results on other micro-benchmarks, readers are requested to refer to [15].

4.3.1 Experiment Testbed

One of our experimental testbed consisted of 300 MHz Pentium II PCs with 128 MB of SDRAM, a 33 MHz/32-bit PCI bus and Linux 2.2 operating system. The PCs in the testbed were equipped with 33 MHz LANai 4.3 NICs and Packet Engines GNIC-II Gigabit Ethernet network interface cards. Myrinet and Gigabit Ethernet switches were used to construct two separate interconnection networks.

MVIA is implemented as a part of the NERSC PC cluster project. In our tests, we used Release 1.0 which was made available in September, 1999. MVIA supports some Fast Ethernet cards, and Packet Engines GNIC-I and GNIC-II Gigabit Ethernet network interface cards. Where hardware support for doorbells is not provided, MVIA uses a fast trap mechanism to trap to privileged mode, avoiding the high cost of a regular system call. Berkeley VIA is implemented as a part of the Millennium project, at the University of California, Berkeley. In our experiments we used Release 2.2 for Linux 2.2.x kernels and Myrinet 7.x network interface cards. Berkeley VIA runs on top of Myrinet NIC's from Myricom. Some parts of the Berkeley implementation is done by the firmware of the NIC.

4.3.2 Non-Data Transfer Micro-Benchmarks

The results obtained from the non-data transfer benchmarks are presented in Table 4.1. It can be observed that the costs of creating VIs and establishing connections are higher in the MVIA implementation. The cost of tearing down a connection and creating and destroying a CQ is higher in BVIA. The cost of destroying VIs are the same for both implementations.

Operation	**MVIA**	**BVIA**
Creating VI	93	28
Destroying VI	0.19	0.19
Establishing Connection	6465	496
Tearing Down Connection	2.56	9
Creating CQ	16.87	206
Destroying CQ	8.44	35

Table 4.1: Non-data transfer micro-benchmarks

4.3.3 Data Transfer Micro-Benchmarks

In this section, we present the results obtained from data transfer related micro-benchmarks.

Base Latency(L_{Base}), Utilization(U_{Base}), and Bandwidth(B_{Base})

The results of base latency and CPU utilization micro-benchmarks are presented in Figures 4.1 and 4.2. It should be noted that the latency is measured with using polling and blocking methods of checking the completion of data transfer. For both cases the CPU usage is given. It can be seen that MVIA has a lower latency for short messages. BVIA outperforms MVIA for longer messages because MVIA require extra data copies which are significant for longer messages.

It can be seen that the CPU utilization is around 100% when polling is used for checking the status of send and receive operations. The BVIA shows a much smaller CPU utilization when blocking operations are used. The reason for the high CPU utilization of MVIA even when blocking operations are used is that before the checking process is blocked the MVIA routine polls for completion for some time.

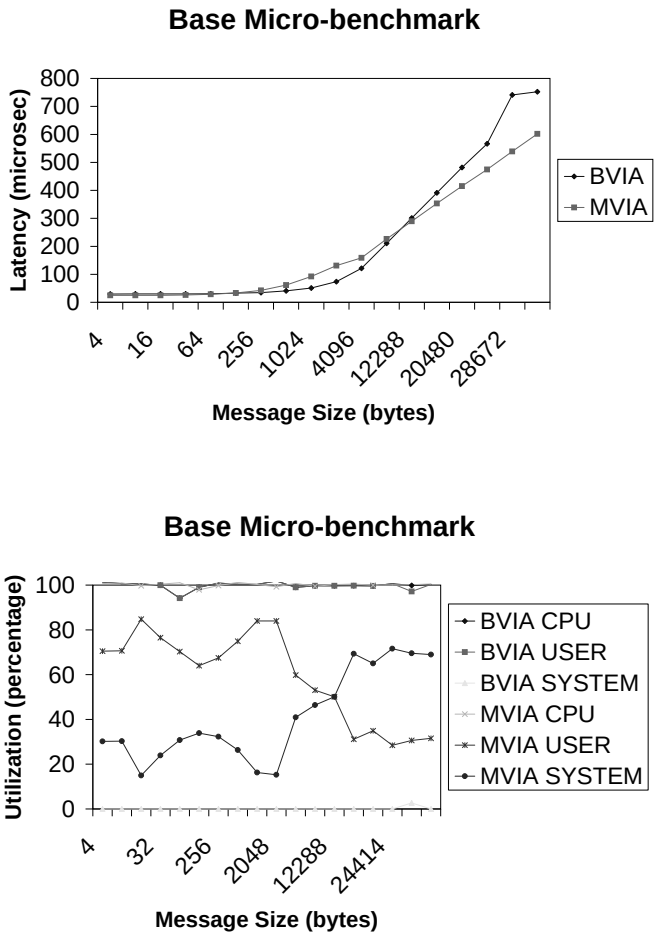

Figure 4.1: Basic latency and CPU utilization with polling.

Impact of Virtual-to-Physical Address Translation (L_{AT}, U_{AT}, and B_{AT})

The difference between latency results and the results obtained from L_{AT} micro-benchmark for BVIA is shown in Figures 4.3 and 4.4. It can be seen that changing the send and receive buffers has a significant effect on the latency of messages for Berkeley VIA. The reason for this significant effect is that in Berkeley VIA the address translation tables are kept in the host memory and the NIC performs the translation. A software cache is used for caching the translations on the NIC. When only one buffer is used for say send messages, after the first send, the required address translation

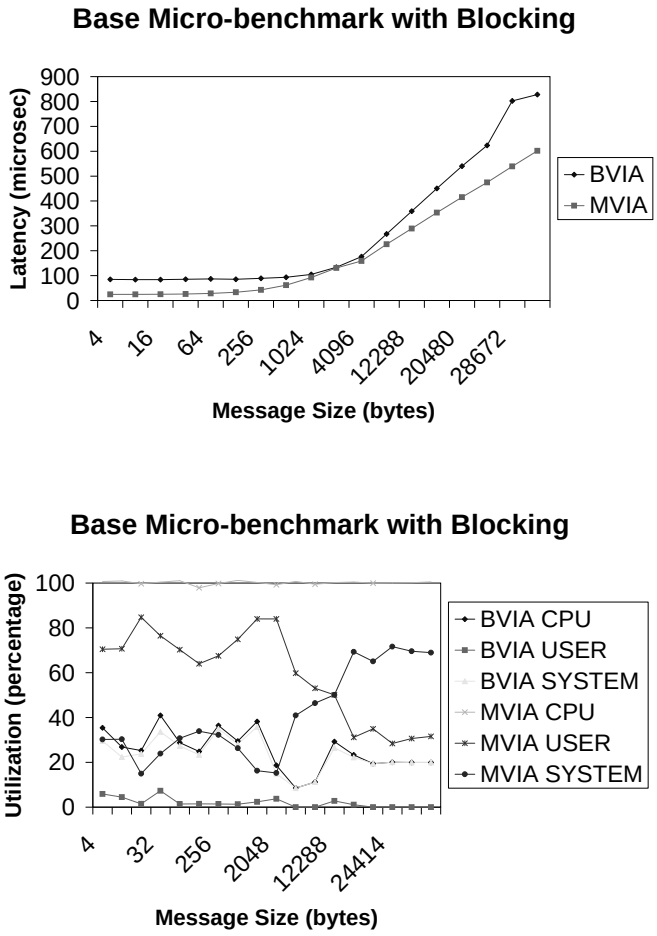

Figure 4.2: Basic latency and CPU utilization with blocking.

entry is cached on the NIC memory and consequent sends don't require the NIC to access the host memory. Since send and receive buffers in L_{AT} are used only once, the overhead of accessing the host memory for obtaining the address translation entries occur for each transfer. Depending on the application and the size and type of the software cache used by the NIC, applications may see latencies between those measured by L_{Base} and those measured by L_{AT}. Since the results for MVIA do not change significantly with the percentage of buffer reuse, we have not presented those results here.

4.4 Summary

In this chapter we proposed VIBe, a micro-benchmark suite for evaluating VIA implementations. We showed that in addition to the standard bandwidth and latency measures, other micro-benchmarks are required for obtaining a better insight into the implementations of VIA. This micro-benchmark suite has been used to evaluate two different existing VIA implementations.

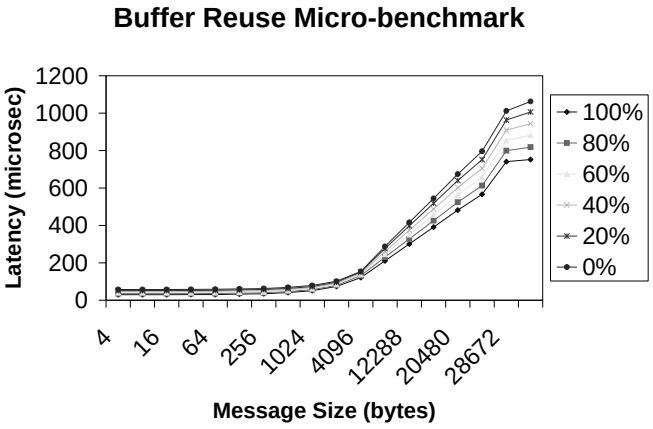

Figure 4.3: Latency for varying percentage of send/receive buffer reuse for BVIA with polling.

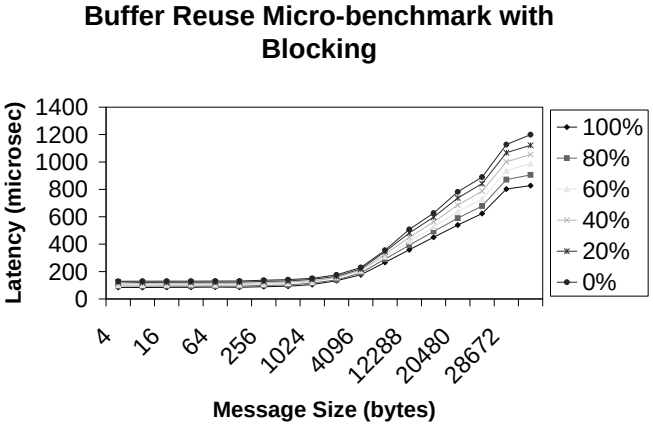

Figure 4.4: Latency for varying percentage of send/receive buffer reuse for BVIA with blocking operations.

CHAPTER 5

DESIGN ISSUES AND ALTERNATIVES FOR SUPPORTING DISTRIBUTED SHARED MEMORY APPLICATIONS IN CLUSTERS

VIA has a low-level API which provides only the basic communication primitives. However, it is difficult for user applications to directly use these primitives unless the applications are rewritten. Thus, a set of middlewares (communication libraries, infrastructures, and substrates) are needed to be built on top of VIA so that the user applications written using popular programming environments such as distributed memory, distributed shared memory, and get/put can take advantage of VIA. As the VIA communication architecture is becoming popular, a lot of research effort is currently being undertaken to design and develop such middlewares. The inherent challenge in designing such a middleware is to translate the functionalities of the programming environment to the basic services provided by the VIA layer with as little additional communication overhead as possible. Such a translation requires the following four-step approach: 1) identifying all mismatches between the functionalities of the programming environment layer and the services provided by the VIA layer, 2) deriving a set of schemes with various components to eliminate such mismatches, 3) analyzing performance trade-offs in implementing the schemes/components and selecting the best ones, and 4) implementing the best schemes/components over a VIA layer. Thus, design and development of such middlewares are complex.

During the last year, there have been many projects to support distributed memory programming environment on VIA for NOWs. All these projects have focused on efficient implementation of the commonly used middleware (Message Passing Interface (MPI) standard [39]) on top of VIA [5, 4]. Compared to distributed memory programming environment, shared memory programming environments provide a much more easier programming paradigm for application developers. In the recent years, many research projects have focused on developing software Distributed Shared Memory (DSM) systems for NOWs [49, 35, 20]. TreadMarks [36, 9] is one of the most popular software DSM systems and has been deployed on many NOWs. Unfortunately, TreadMarks and other software DSM systems do not deliver good parallel speedup and are not scalable due to the high overhead of communication in NOWs. Since VIA provides high-bandwidth and low-latency communication, it is an open challenge whether suitable middleware can be developed for software DSM systems like TreadMarks so that many application developers can enjoy the benefits of DSM environment on NOWs. To the best of our knowledge, there has not been any work on developing a middleware for supporting TreadMarks on top of VIA for NOWs.

In this chapter, we take on such a challenge and study how an efficient middleware can be developed such that applications using the popular TreadMarks DSM package can take advantage of the communication performance of VIA. Currently the communication primitives of TreadMarks are built on top of UDP protocol. Due to the high overhead associated with UDP, TreadMarks cannot exploit the maximum performance delivered by the emerging gigabit networking technologies

such as Myrinet and Gigabit Ethernet. Thus, it is essential to design and develop a middleware involving a thin communication substrate between VIA and TreadMarks as shown in Figure 5.1.

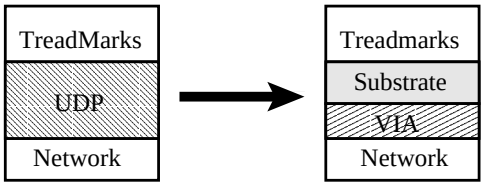

Figure 5.1: Goal: designing and implementing a thin communication substrate so that TreadMarks can run on top of VIA with little overhead. The left stack shows the existing implementation of TreadMarks. The right stack shows the new implementation discussed in this chapter.

The open challenge in designing such a substrate is that it should bridge the gap between the services provided by VIA and those required for an efficient implementation of TreadMarks. In this chapter, we take the four-step approach mentioned earlier in developing the targeted substrate. First, we identify the mismatches between the communication requirements by TreadMarks and the services provided by VIA. In particular, we show how the request-response communication model required by TreadMarks can not be directly supported by VIA. The VIA specification requires that a receive descriptor which contains the address of a receive buffer must be posted before the arrival of a message. Otherwise, the message is dropped. However, the arrival of request messages at participating nodes under the TreadMarks can not be predicted. Furthermore, VIA requires that all send and receive buffers be in registered (pinned) memory regions.

After identifying these mismatches, we propose a set of schemes to eliminate such mismatches. These schemes include connection setup, buffer management, advance posting of descriptors for unexpected messages, and alternative designs to handle asynchronous messages. Next, we analyze the performance impact of these schemes together with the VIA functions. We also propose and evaluate different design alternatives for enhancing some VIA function (such as the VIA Notify mechanism) so that the new substrate can be designed with low overhead. Finally, we derive the best set of alternatives and implement them on two enhanced implementations of VIA (MVIA and Berkeley VIA) on two different networking technologies, Gigabit Ethernet and Myrinet, respectively. These implementations are carried out in the Linux environment.

We evaluate the performance of our implementation by using several micro-benchmarks and applications. We show that the communication and wait times, and therefore the total execution times of different applications can be significantly reduced by using VIA. It is shown that factors of improvement up to 2.05 on an eight node system can be achieved in comparison with the original UDP implementation. Due to the reduced communication overhead with VIA, the applications also exhibit better parallel speedup as the system size increases. The new VIA implementation also delivers better performance for a range of application sizes.

It is to be mentioned that the goal of this work has been to analyze the challenges involved in designing the communication substrate, implementing it, and evaluating its performance without performing any major modifications to the TreadMarks layer. Our design experience and evaluation study indicate that there are multiple opportunities to improve the performance of TreadMarks over

VIA by performing modifications to the TreadMarks layer, VIA layer, or both. We are currently focusing along these directions.

The rest of this chapter is organized as follows: A brief discussion of the communication model and primitives used in TreadMarks is presented in Section 5.1. In Section 5.2, we briefly overview the relevant features of Virtual Interface Architecture, outline the services offered by VIA, and discuss the mismatches between the VIA services and the communication requirement of TreadMarks. Schemes to alleviate the mismatches and their implementation issues are discussed in Sections 5.3 and 5.4, respectively. In Section 5.5, we present the experimental results including the evaluation of different design alternatives, micro-benchmarks, and applications. Related work is discussed in Section 5.6. In Section 5.7, we present our conclusions.

5.1 Overview of TreadMarks

TreadMarks is a popular software DSM system which runs on networks of workstations without any modification to the operating system kernel. Furthermore, it does not rely on any particular compiler. TreadMarks relies on user-level memory management techniques to provide coherency among participating processes for distributed shared memory regions. TreadMarks uses the UDP communication protocol for exchange of control and data messages. Since the overhead of UDP communication is high, effort has been made in the design of TreadMarks such that the amount of communication necessary is as low as possible. In the rest of this section we briefly discuss the coherency protocol and the communication model and primitives used by TreadMarks.

5.1.1 Coherency Protocol

Shared memory accesses in TreadMarks are divided into two groups: ordinary accesses and synchronization accesses. Synchronization accesses are further categorized into acquire and release accesses. TreadMarks uses a *lazy* implementation of *release consistency*. In *lazy release consistency* (LRC), the propagation of modifications is postponed until the time of the *acquire* [36]. In order to do so, execution of each process is divided into *intervals*. An index is associated with each time interval. Upon the execution of acquire or release, a new interval begins. Intervals of different processes are partially ordered. The partial order can be represented by assigning a *vector timestamp* to each interval. For a process p to pass an acquire, *write notices* for all intervals named in the current vector time stamp of the previous releaser q but not in p's vector timestamp should become visible to p. The arrival of a write notice for a page causes the invalidation of the copy of that page. The modifications to the page are propagated to the local copy only when the next access to the page causes a miss. The coherence protocol of TreadMarks works on a memory page-level granularity. However, it supports multiple concurrent writes to modify a page to address the *false sharing* problem.

Shared pages are initially write protected. At the first write, in response to the protection violation, a copy of the page (a *twin*) is made and the write protection is removed such that further writes to the page can occur without the involvement of TreadMarks. The twin and the current version of a page can be compared to create a *diff*. TreadMarks uses a *lazy* diff creation mechanism in which the diff is created only when a request for the modifications of a page is received from another processor in the system.

5.1.2 Communication Model and Primitives

TreadMarks relies on a request-reply type of communication. As shown in Figure 5.2, Request messages (such as a request for page diffs) are sent out by using socket function calls. Upon arrival of a Request message at a node, an interrupt is issued and after the message has been processed by the kernel, the SIGIO signal is raised. The SIGIO signal handler then processes the Request message and sends a Response message if necessary (say, the requested diff). It is possible that the Request message may get forwarded to another node to prepare and send the response. Whenever a Response is expected, the node which has sent the Request message waits until the Response message is received. Then the received response message is processed (say, by incorporating a page diff).

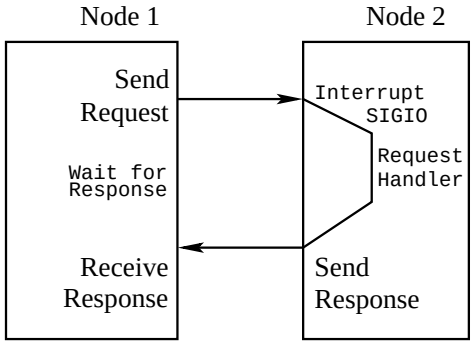

Figure 5.2: Request-response communication model used in TreadMarks with UDP support

The communication services required by TreadMarks can be divided into four major groups: sending Request messages, sending Response messages, receiving Request messages, and receiving Response messages. These services along with the corresponding UDP/TCP function calls are shown in Fig.5.3. It can be observed that in addition to the *send* and *recv* primitives, *select* (Recv-any) primitive is also used. Select is mainly used for finding out if any Request message has been received from any node. TreadMarks takes advantage of the operating system feature that allows for user level signal to be provided and the fact that completion of receive operations on sockets raises the SIGIO signal.

Since TreadMarks was developed on top of UDP/TCP primitives, the communication services required by TreadMarks matches well with the communication services provided by the UDP/TCP primitives. Examples of such UDP services which TreadMarks uses are: ability to send message from any user buffer, automatic allocation of temporary buffers for receives, and ability to receive a message from any node out of a group of nodes. As we discuss the features of VIA in the next section, we will see that significant mismatches exist between the services required by TreadMarks and those supplied by VIA.

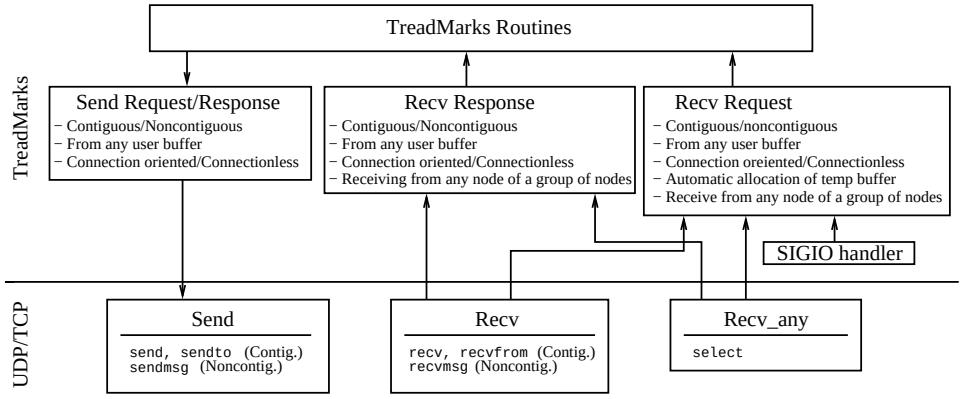

Figure 5.3: Four major groups of communication services required by TreadMarks and their implementation using UDP/TCP communication primitives.

5.2 Relevant Features of Virtual Interface Architecture (VIA)

VIA specifies two types of data transfer facilities: the traditional send/receive messaging model and the Remote Direct Memory Access (RDMA) model. In the send/receive model, there is a one to one correspondence between the send descriptors on the sending side and the receive descriptors on the receiving side. In the RDMA model, the initiator of the data transfer specifies the source and destination virtual addresses on the local and remote nodes, respectively. The RDMA write operation is a required feature of the VIA specification while the RDMA read operation is optional. There is no need for posting any descriptors at remote nodes for RDMA operations. There is one exception and that is when an RDMA write message contains immediate data.

Figure 5.4 illustrates the services provided by VIA with a brief description of their requirements and characteristics. There are five basic services: send, receive, RDAM write, Completion Queue (CQ), and Notify mechanism. One critical aspect associated with the VIA-level communication is that the communication primitives (send, receive, and RDMA write) require their associated buffers to be in registered memory. The intent of the memory registration is to give an opportunity to the VIA provider to pin (lock) down user virtual memory in physical memory so that the network interface can directly access the user buffers. This eliminates the need for copying data between the user buffers and intermediate kernel buffers typically used in traditional network transports. The other critical aspect of the VIA specification is that a receive descriptor with the addresses of user buffers must be posted before the arrival of a message. Otherwise, the message is dropped.

The CQ and Notify mechanisms help in detecting the completion of a communication operation and taking appropriate steps. Each VI work queue can also be associated with a CQ. A CQ merges the completion status of multiple work queues. Therefore, an application need not poll multiple work queues to determine if a request has been completed. The Notify mechanism allows a user handler procedure to be associated with a work queue or completion queue. Upon completion of any operation associated with such a queue, the corresponding handler procedure will be executed.

It can be observed that the services provided by VIA and their requirements are different from those of the standard UDP protocol. Thus, for implementing TreadMarks over VIA, there are inherent mismatches between the communication requirements of TreadMarks and the services

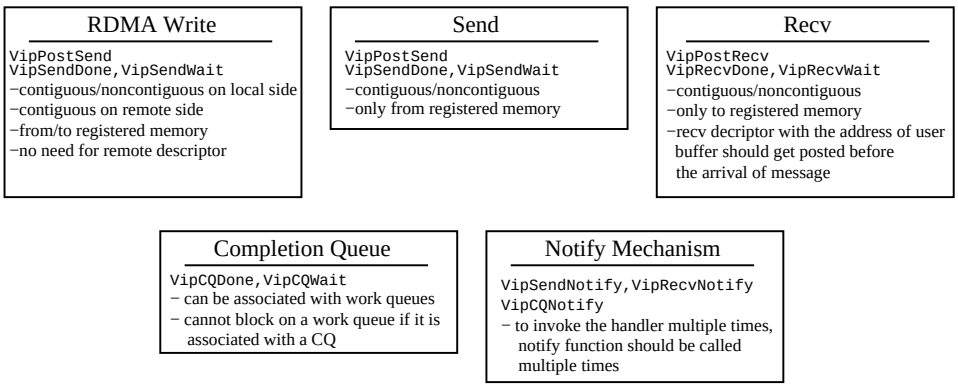

Figure 5.4: Services provided by VIA together with their requirements and characteristics

provided by VIA. In the next section, we elaborate on these mismatches and propose alternatives for designing the substrate to alleviate the mismatches with as little communication overhead as possible.

5.3 Challenges in Designing the Communication Substrate

In this section, first we outline the major issues of designing the communication substrate by identifying the mismatches between the services provided by VIA and the communication requirements of TreadMarks. Next, we present a set of components to alleviate the mismatches and evaluate design alternatives for each of these components.

5.3.1 Major Issues

Let's closely look at the requirements and characteristics of the VIA services as shown in Fig. 5.4 and compare them with those of the UDP services shown in Fig. 5.3. We have briefly analyzed some of the mismatches in [18]. The major mismatches can be identified as given below:

1. The communication model of TreadMarks on top of UDP, as shown in Fig. 5.3, handles Request and Response messages differently. In particular, it can be seen that while Request messages arrive in an asynchronous manner, Response messages are exchanged in a synchronous fashion. Under VIA, the arrival of asynchronous message on a VI connection can be identified and processed by associating a Notify handler with the receive queue of the VI. Furthermore, a single Notify handler can be used for a group of VI connections if a CQ is associated with the receive queues of those connections. However, once a CQ and Notify handler is associated with a VI connection, it is not possible to use these features (which are costly) only for a subset of messages arriving at the connection. Therefore, it is difficult to provide differentiated service (for Response and Request messages) on a single VI between any pair of processes.

2. In the UDP protocol, temporary buffers are automatically allocated on the arrival of messages. The user can look at these buffers later on. However, the VIA specification requires that recv

descriptor with the addresses of user buffer must be posted before the arrival of the message. Otherwise, the message is dropped.

3. The VIA specification requires that the applications *register* the virtual memory regions which are going to be used by VIA descriptors and user communication buffers for send, recv, or RDMA write operation. The intent of the memory registration is to give an opportunity to the VIA provider to pin (lock) down user virtual memory in physical memory so that the network interface can directly access the user buffers. This eliminates the need for copying data between the user buffers and intermediate kernel buffers typically used in traditional network transports. No such memory registration is needed for UDP.

4. TreadMarks implemented on top of the UDP protocol uses and a SIGIO handler and the 'select' call to detect the arrival of asynchronous messages and operate on them. VIA provides a mechanism to associate a CQ with a set of work queues. The VIA notify mechanism and the associated handler can be used to detect and process the arrival of a message from a group of nodes. In order to receive and process asynchronous messages by using these VIA features in an efficient manner, it is critical that the CQs and the notify mechanism are implemented efficiently. A prompt and efficient mechanism for responding to Requests can have a significant role in reducing the wait time and therefore the overall execution time of applications.

5.3.2 Components of the Substrate

In order to alleviate the above mismatches, we propose the substrate as shown in Fig. 5.5. While proposing this substrate our goal has been three-fold: 1) not making any changes to the coherency protocol of TreadMarks or its communication model, 2) not adding any new functionality to the VIA layer, and 3) making minimal changes to the communication primitives of TreadMarks so that they can be interfaced with the new substrate.

The substrate has basically four major components: connection management, pre-posting of receive descriptors, buffer management, and schemes for handling asynchronous messages. These four components solve the four mismatches mentioned in the previous subsection. These four components work together with the VIA level primitives and services leading to three new communication primitives (New Send, New Recv, and New Recv_any) at the substrate layer. The new implementation of TreadMarks uses these new primitives. The dependencies of the new components on the VIA-level primitives and services and relationships between the TreadMarks message types are shown in detail in Fig. 5.5. The design alternatives behind each of the four components and their cost-performance trade-offs are discussed in detail in the following subsections.

5.3.3 Connection Management

As mentioned earlier, under the communication model of TreadMarks, the arrival of Request messages at receiving nodes cannot be predicted. However, Response messages are exchanged in response to a Request message and therefore the process which has issued a Request expects to receive a Response message. As mentioned in Section 5.3.1, in VIA, all messages received on a given VI are treated the same way. Thus, it is difficult to provide differentiated services for messages arriving on a single VI. In order to provide such differentiated services, a scheme of using two different VI connections between each pair of processes looks attractive. One VI can be used for receiving Request messages and the other one for Response messages. Figure 5.6 illustrates such

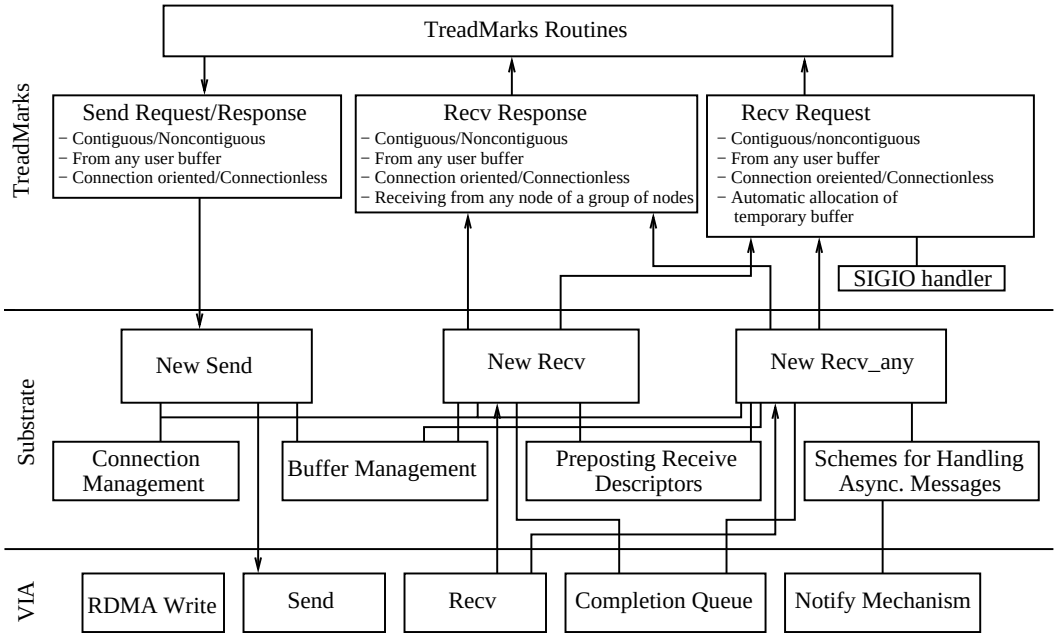

Figure 5.5: Components of the communication substrate and their relations with VIA services and TreadMarks requirements

a connection management procedure. This way, Notify handler procedures can be associated with only Request messages and the overhead involved in invoking these handlers can be avoided for Response messages.

For a system consisting of n processes, the baseline VIA setup requires $(n-1)$ VIs per process. The proposed scheme requires $2(n-1)$ VIs per process. It is to be noted that current VIA implementations easily support 1-2K VIs per node. Thus, the use of 2 VIs between each pair of process (instead of the default one) does not provide a serious threat for designing a DSM system with 512-1024 processes using the currently available VIA implementations. As the VIA implementations are moving more into the hardware level and modern NICs are supporting more memory, the number of VIs per process on future implementations will continue to rise. Thus, larger DSM configurations can be supported with the proposed approach in future.

It is to be noted that separating Request and Response messages will have an effect on how asynchronous messages are processed. We will discuss these issues in more detail in Section 5.3.6.

5.3.4 Pre-posting of Receive Descriptors

As mentioned in Section 5.3.1, in VIA, received messages for which receive descriptors are not posted are dropped. In other words, unexpected messages are dropped at the receiving nodes. However, TreadMarks (like many other systems) relies on the request-reply mechanism. Therefore, each node must be ready to receive requests from any other participating node at any time. In order to support such a feature while avoiding the need for a costly retransmission mechanism, it is crucial to guarantee that for every incoming message a corresponding receive descriptor and the associated receive buffer are posted ahead of time.

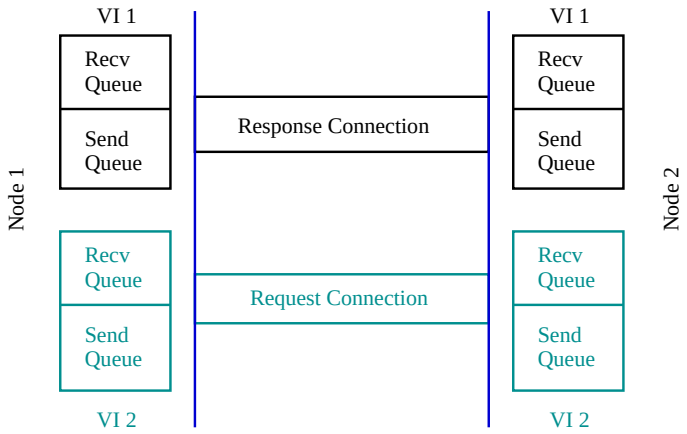

Figure 5.6: Proposed connection management scheme with two VI connections between each pair of processes.

This leads to the following questions: 1) How many descriptors/receive buffers we post in advance? and 2) When do we post these descriptors/receive buffers? As we have seen earlier, TreadMarks uses two kinds of messages: Request messages and Response messages. We provide answers to the above two questions by considering these two different kinds of messages separately.

Let us consider the Request messages first. For a given process, a Request message can come from any other process. Thus, from the point of view of a single receiver process in a system with n processes, we need at least $(n-1)$ descriptors and the associated $(n-1)$ receive buffers. The number of posted descriptors/receive buffers may need to be scaled by a factor of o, if o outstanding messages between each pair of processes are allowed. The above number of pre-posted descriptors/receive buffers are sufficient because after a request message is received and before the corresponding Response message is sent out, a new receive descriptor can be posted. In cases where the Request is forwarded to another node a new receive descriptor should be posted before the Request is forwarded.

Let us consider the amount of registered memory needed for posting the descriptors and buffers using the proposed scheme. A VIA descriptor is 48 bytes long. The size of a request size message under TreadMarks could be up to the MTU size (32K bytes). For a reasonable system size (say 256 processes) and up to one outstanding message between each pair of processes, the proposed scheme requires 8 MBytes of receive buffers and 12KBytes of receive descriptors. Since current and future systems can easily provide such amount of registered memory, we adopted this approach in our design.

For large-scale systems, if the amount of available registered memory can not support at least one descriptor/receive buffer from each of the other processes, a *rendezvous* protocol can be used. Under this scheme, a sender needs to first send a request message using immediate data (which requires a descriptor but not a receive buffer at the remote node), wait for an acknowledgment, and then send the actual Request message. The receiver, after receiving the request message (for which a descriptor is pre-posted), can post a descriptor and a receive buffer to receive the actual Request message. This scheme can work with only a fixed number of receive buffers per process. However, it will add additional communication overhead and may degrade the overall performance.

Since the earlier approach is realistic for a reasonable-sized system, we did not proceed along this rendezvous approach further.

Now let us consider the Response messages. There are two types of response messages: response to lock requests and response to non-lock requests. Response to a non-lock request comes from the node to which the request was targeted. In this case, a descriptor and a receive buffer for that node can only be pre-posted. However, response to a lock request can come from any of the other nodes. In this case, we do have to pre-post descriptors and receive buffers for each node of the system. Since responses to both types of requests (locks and non-lock) come on the same VI, we must post descriptors and receive buffers for all nodes in this system. The size of the receive buffers for response to non-lock requests could be as long at the maximum transfer unit of TreadMarks. Thus, this situation becomes identical to the handling of Request messages and requires the same amount of total memory size. As discussed earlier, the amount of memory needed is not a constraint for current and future systems. Thus, we adopted this approach. With respect to when to pre-post the descriptors, we suggest posting $(n-1)$ descriptors and receive buffers per process during the initialization. Every time a Response message arrives and gets consumed, we post a new descriptor with the associated receive buffer.

Another approach for dealing with Response messages is to use two VI connections (instead of one) for Response messages. One of these connections can be used for lock messages and one for non-lock messages. Rendezvous protocol can be used for lock messages. This way the need for assigning receive buffers for all non-lock responses can be eliminated. This approach requires a total of 3 connections between each pair of nodes. As discussed in the earlier subsection, such an increase may limit the maximum number of nodes that can be supported.

It should be noted that RDMA write operations cannot be used in order to avoid the need for posting receive descriptors in advance because there is no mechanism through which the remote node is notified about completion of the RDMA operation. Immediate data field can be used in an RDMA write operation as a mechanism for notifying the remote node. However if an RDMA write operation contains an immediate data, the remote node is required to post a receive descriptor before the RDMA write operation is executed (Section 2.3.2 of [7]).

5.3.5 Buffer Management

VIA requires that send and receive buffers to be in registered memory regions. Registered memory regions contain memory pages which are pinned down in the physical memory. Therefore, the size of registered memory is limited by the size of physical memory and the OS requirements on each node and can affect the performance of running applications.

Outgoing Request and Response messages are constructed by TreadMarks. The VIA capability for sending messages from noncontiguous buffers (by using multi-data segment descriptors) can be used whenever the outgoing message is noncontiguous. In order to avoid extra data copies, TreadMarks is required to be modified such that outgoing messages are constructed in registered memory regions. We did not follow this approach. Instead we used a pool of send buffers in the registered memory for outgoing messages and copied outgoing messages into these buffers before sending them.

The incoming messages can be received into a buffer in registered memory before being processed. Since a pointer to incoming Requests is passed to TreadMarks routines, Request messages can be processed without any extra data copies. The Response messages are required to be copied from the buffer in which they have been received to TreadMarks data structures before being processed. This approach does require an extra data copy on receive side but does not require any

changes in TreadMarks. We used this approach. Alternatively, TreadMarks can be modified such that received Response messages can be processed without making any extra copies.

5.3.6 Schemes for Handling Asynchronous Messages

Handling asynchronous messages (i.e. Request messages) is another important issue that should be addressed. VIA provides a Notify mechanism through which a user handler routine is executed whenever a communication operation is completed. Send, receive, and completion queues can be associated with handler routines. Whenever a descriptor associated with these queues is completed, the corresponding handler routine is executed. The way this mechanism is implemented can have a great impact on a system such as TreadMarks in which Request messages are unexpected and should be processed as quickly as possible. In this section, we discuss four different approaches that can be used to implement the Notify mechanism or can be directly used by applications for processing asynchronous messages.

It should be noted that if asynchronous messages (i.e. Request messages) are expected to arrive only on a subset of VI connections (as suggested in Section 5.3.3), the overhead involved in processing these messages can be avoided when other messages (i.e. Response messages) received on other connections are processed. For all of the proposed approaches, associating all asynchronous receive work queues with a Completion Queue will make the implementation easier and more scalable. In the rest of this section, we discuss these approaches.

Communication Thread: In this approach, a separate thread is created to process asynchronous messages. This thread polls or blocks for completion of receive operations on VI connections of interest. This approach can be both used by TreadMarks and by the implementation of VIA to implement the Notify mechanism. If the communication thread blocks for the completion of receive operations, using a CQ and associating it with receive queues on which asynchronous messages may arrive becomes more important. (Otherwise one thread for each such receive queue is required.) Even when the communication thread polls for completion of receive operations, using a CQ becomes more beneficial when the number of processes in the system increases. The performance of this method will greatly depend on the performance of operating system support for threads.

Timer: In this approach, a timer interrupt handler is used for checking on receive or completion queues of interest. If any receive operation is completed, the necessary action is taken (i.e. the corresponding handler is invoked). Depending on the granularity of the timer, the overhead and response time of this approach will vary. This approach can be employed by TreadMarks or by the VIA implementation of Notify mechanism.

Interrupt: In this approach, Receive queues or a completion queue which is associated with these queues are associated with a Notify handler procedure. Upon completion of a receive operation, an interrupt is raised and the interrupt handler invokes the corresponding Notify handler. This approach can be used for implementing the VIA Notify mechanism.

Polling: In this approach, the user application is responsible for detecting the arrival of asynchronous messages by polling for asynchronous messages. This approach requires that TreadMarks is modified such that it provides the user with a function which polls for Request messages. The effect of using such an approach will heavily depend on the frequency and location of the polling operations in the application. We will compare the performance of these approaches in Section 5.5.

5.4 Implementation

In this section, we present the details of our implementation of the communication substrate. We also present the modifications we have made to the VIA implementation in order to improve the performance of the VIA Notify mechanism. The implementation follows the design choices made, as presented in Section 5.3. As mentioned earlier, the primary goal of our exercise is to alleviate the mismatch between TreadMarks requirements and VIA services without changing the TreadMarks protocol. TreadMarks was minimally modified to be interfaced with our communication substrate.

We used two publicly available implementations of VIA: MVIA [3] and the VIA implementation from Millennium Project at Berkeley (Berkeley VIA) [25]. MVIA is a software implementation of VIA in which fast traps and interrupts are used for sending and receiving messages. Completion queues, notification mechanism, and RDMA operations are supported by MVIA. BVIA implements only a subset of VIA for Myrinet Network. We used the BVIA version working on Myrinet LANai 7 network interface cards. Completion queues are implemented by BVIA. But, Notification mechanism and RDMA operations are not supported.

The Connection Management component of the implemented substrate establishes connections between all pair of nodes. The number of connections between each pair of nodes can be passed to this component as an input parameter. For our implementation, the number of connections was set to two (one for Request messages and one for Response messages) as suggested in Section 5.3.3. By doing so, we can easily distinguish between the Request and Response messages based on the VI connection on which they are received. The receive queue and/or send queue of each VI connection can be associated with a CQ. Furthermore, each queue can be associated with a Notify handler procedure. In our implementation, all receive queues associated with Request connections were associated with one single CQ. Furthermore, this completion queue was associated with a Notify handler procedure. This association between the CQ and a Notify handler was done only for the cases where the Notify mechanism was used for handling asynchronous messages. We discuss this issue in detail towards the end of this section.

The Buffer Management component of the substrate was implemented such that each work queue can be associated with a registered memory region to hold descriptors and communication buffers. As discussed in Section 5.3.5, for each VI connection, this registered memory region was used for both send and receive descriptors. For each VI connection, we also used a portion of this registered memory for communication buffer. In the current implementation, all incoming and outgoing messages are sent from or received into these buffers. The size of the receive and send buffers were set to be equal of the TreadMarks maximum transfer unit (32 KBytes).

The Pre-posting Receive Descriptors unit was implemented as discussed in Section 5.3.4. This component pre-posts a number of descriptors on all work queues. Whenever a descriptor from a work queue is consumed, a new descriptor is posted to the same work queue. The number of posted descriptors to each work queue can be passed onto this component as a parameter. In the current implementation this number was set to one corresponding to one outstanding message between each pair of processes.

As mentioned earlier, MVIA supports the Notify mechanism but BVIA does not support it. We implemented several mechanisms for handling asynchronous messages as discussed in Section 5.3.6. Here we discuss our implementations. The evaluation of these approaches are discussed later in Section 5.5.2.

MVIA uses the Communication Thread approach for implementing the Notify mechanism. We also implemented the same approach at the substrate level by using a thread which blocks until a communication operation associated with a queue of interest (the completion queue associated

with receive queues of Request connections) takes place. Whenever this thread gets unblocked, it invokes the TreadMarks procedure responsible for servicing asynchronous messages. The SIGIO handler procedure is responsible for servicing asynchronous messages in TreadMarks. We used the same procedure with the only difference being that select and read statements were replaced by VIA calls for checking CQs and receiving messages. After servicing an asynchronous message, the thread blocks until the arrival of the next asynchronous message.

For the Timer approach, the communication substrate was implemented such that a SIGALRM signal is raised periodically. The TreadMarks was modified such that the handler for SIGALRM signal was set to be the TreadMarks modified SIGIO handler. In our systems the granularity of generating SIGALRM is $10ms$ and that is what we used in our implementation.

For the Interrupt approach, we modified the MVIA driver to issue a SIGUSR1 signal whenever a communication operation on a queue associated with a Notify handler is completed. (It should be noted that incoming messages cause an interrupt in the MVIA implementation.) For the BVIA implementation, the BVIA Myrinet Control Program (MCP) was modified to issue an interrupt for operations associated with a Notify handler. The interrupt handler routine was used to raise the SIGUSR1 signal. In cases where a completion queue is associated with a work queue, the BVIA interrupt handler routine for processing completion queues was used to raise the SIGUSR1 handler. The SIGUSR1 signal handler for both BVIA and MVIA were written to invoke the corresponding Notify handler. TreadMarks was minimally modified to use the modified SIGIO as the Notify handler procedure.

For the implementation of the polling approach, TreadMarks was modified to provide access to a new TreadMarks procedure called Tmk_poll. This procedure was written to call the procedure which polls for incoming Requests. If any Request message is detected, it is processed. We used this approach only to evaluate the effect of handling asynchronous messages on the system performance.

5.5 Performance Evaluation

In this section we evaluate the performance of our proposed implementation. First, we provide a brief description of our experimental testbed. Then, we evaluate the impact of different design choices for handling asynchronous messages and select the one which delivers the best performance. Using the selected design, next we compare the base implementation of TreadMarks on UDP and our new implementation on VIA. Performance results corresponding to both micro-benchmarks and application-level evaluations are presented. For application-level evaluations, first we present comparison results for the default system size and application size. Next, we study the effect of system size and the effect of application size separately.

5.5.1 Experimental Testbed and Setup

The results presented in this section were obtained on a cluster of eight PCs connected with Gigabit Ethernet and Myrinet interconnects. Each PC uses a 300 MHz dual Pentium II processor with 128 MB of SDRAM and a 33 MHz/32-bit PCI bus. Linux 2.2.5 operating system runs on all these systems. For the BVIA implementation the SMP version of the kernel is required to be used while the MVIA implementation runs on the non-SMP version of the kernel. In all of our experiments only one processor on each node was used. Regarding the Gigabit Ethernet interconnect, a switch and a set of network interface cards from Packet Engine were used. The Myrinet interconnect included a switch and a set of LANai 7 cards from Myricom.

The experimental setup consisted of TreadMarks running over four different communication subsystems. These subsystems were as follows:

1. UDP over Gigabit Ethernet (the base TreadMarks implementation)
2. UDP over Myrinet (porting the base TreadMarks implementation onto the UDP/GM layer)
3. VIA over Gigabit Ethernet (TreadMarks running over our proposed new substrate on top of MVIA [3], the standard VIA implementation for Gigabit Ethernet)
4. VIA over Myrinet (TreadMarks running over our proposed new substrate on top of the latest version of Berkeley VIA [25], the implementation developed by the Berkeley group for Myrinet).

In the remaining part of this section, these four communication systems are identified as UDP/Ethr, UDP/Myri, MVIA/Ethr, and BVIA/Ethr, respectively. The raw latency numbers for these four communication subsystems were observed to be as follows: 85.0 microsec (UDP/Ethr), 111.0 microsec (UDP/Myri), 23.0 (MVIA/Ethr), and 30.0 (BVIA/Myri).

5.5.2 Evaluation of Alternatives for Handling Asynchronous Messages

Four alternative approaches were mentioned in Section 5.3.6 to handle asynchronous messages. We implemented these alternatives as discussed in Section 5.4. It is to be noted that we implemented the TreadMarks level communication thread approach only for MVIA.

In order to evaluate the effectiveness of these four alternative approaches, we evaluated the performance of two applications from the TreadMarks Distribution: Jacobi and SOR. The characteristics of these two applications together with other applications are described in Section 5.5.4. For evaluating the polling mechanism we modified both applications and inserted calls to Tmk_poll such that we got the best execution time. We evaluated the communication thread approach (both at the TreadMarks level and the VIA level) only for the MVIA/Ethr communication subsystem. The overall performance for both these methods was found out to be comparable to that of the timer method and are not shown here. Note that we ran the applications in non-SMP mode as required by the MVIA implementation[1]. Figure 5.7 illustrates the execution times of Jacobi and SOR applications when alternative methods for handling asynchronous messages are used.

It can be observed that the interrupt approach provides the best performance. Thus, for the remaining part of our performance evaluation, we only consider the interrupt-based approach of implementing the VIA notify mechanism.

5.5.3 Micro-benchmark-level Evaluation

The micro-benchmark-level evaluation was carried out by using the four micro-benchmarks included in the TreadMarks distribution. These micro-benchmarks are: *Barrier*, *Lock*, *Diff*, and *Page*. These micro-benchmarks measure the time required for performing basic TreadMarks operations. The Barrier micro-benchmark reflects the time for performing a barrier across a set of nodes. In the Lock micro-benchmark, the cost of acquiring a lock is measured. There are two versions of this micro-benchmark: direct and indirect. The direct case reflects the situation where the lock being

[1]There are some known bugs in the MVIA implementation for SMP nodes.

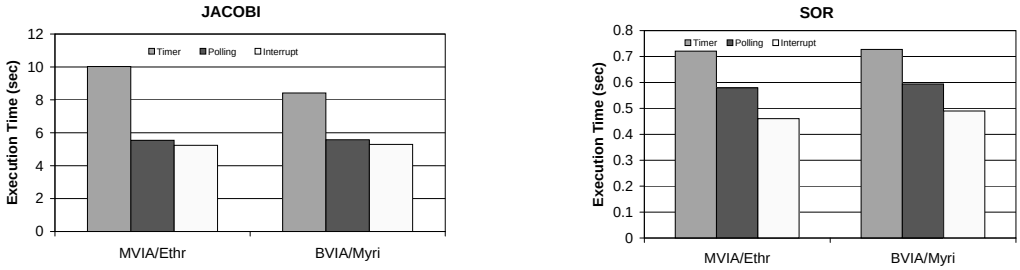

Figure 5.7: Performance impact of three alternative approaches (timer, polling, and interrupt) for handling asynchronous messages. Overall execution times for two applications (Jacobi and SOR) on the two VIA communication subsystems (MVIA/Ethr and BVIA/Myri) are shown.

acquired has already been acquired and released by its manager node. The indirect case reflects the situation where the lock being acquired has been acquired and released by a third node.

The Page and Diff micro-benchmarks are used to evaluate the performance of TreadMarks when shared memory is accessed and diffs are obtained and applied to a page. In the Page micro-benchmark, a shared memory region consisting of multiple memory pages is first created (by using Tmk_malloc) and then distributed among participating processes (by using Tmk_distribute) by process 0. After process 0 reads one word from each page, process 1 reads the same word from each page.

The Diff micro-benchmark has two cases: small and large. In the first case, one word from each page is read by one process while the same words have been written into by another process earlier. The second case is similar to the first case with the difference being that all words of the shared memory region are accessed by the writer and reader processes.

Figure 5.8 shows the performance results of four micro-benchmarks and their different cases. It can be easily seen that for all cases, the VIA implementations outperform the UDP implementations. For the barrier operation on eight nodes, the factor of improvement when UDP/Ethr is replaced by MVIA/Ethr is 1.76. When UDP/Myri is replaced by BVIA/Myri, the factor of improvement of 1.84 is achieved for barrier on eight nodes. For the lock operations, the factor of improvement is up to 3.09 and 2.76 when UDP is replaced by MVIA and BVIA, respectively. The improvement of Page and Diff operations is up to 2.31 and 2.06 for MVIA and BVIA, respectively.

Comparing the results obtained from MVIA/Ethr and BVIA/Myri shows that MVIA/Ethr outperforms BVIA/Myri by a factor of 1.90 and 1.68 for Barrier and Lock operations, respectively. While both of these systems perform comparably for Diff operations, the BVIA/Myri implementation shows an improvement of 20.0% in comparison with MVIA/Ethr for Page micro-benchmark.

5.5.4 Application-Level Evaluation

In this section, we first describe the applications we used in our evaluation and then discuss the results. It should be noted that no attempt was made to improve the performance by modifying the applications. We used these applications as they were in the TreadMarks distribution and made no attempt to improve the performance by modifying them.

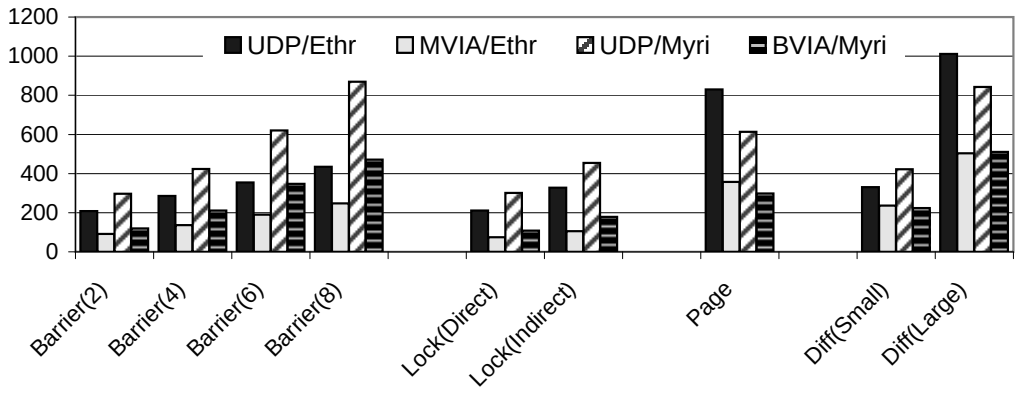

Figure 5.8: Performance results of four micro-benchmarks (Barrier, Lock, Page, and Diff). Different cases of Barrier, Lock, and Diff are shown. Barrier (x) indicates the time to achieve a barrier on x nodes. For each of the micro-benchmarks and their individual cases, the four bars (left to right) reflect the time on four different communication subsystems: UDP/Ethr, MVIA/Ethr, UDP/Myri, and BVIA/Myri.

Characteristics of Applications

We used four applications from TreadMarks distribution for evaluating our implementation. These applications are: SOR (red-black successive over-relaxation on a grid), TSP (traveling salesman problem), Jacobi, and 3D FFT. By default, SOR performs 10 iterations on a grid of 2000×1000 single-precision floating-point numbers. TSP solves the traveling salesman problem for a 18-city tour. Jacobi uses a 1022×1022 grid of real numbers. By default, FFT works on a $32 \times 32 \times 32$ array. The important statistics from the execution of these applications on our base system (eight nodes on UDP/Ethr) are shown in Table 5.1. (Note that every 8 barriers are counted as one.)

	SOR	TSP	Jacobi	3D FFT
Input	2000×1000	18-city tour	1022×1022	$32 \times 32 \times 32$
Time (s)	0.91	0.81	5.68	0.49
Barrier/s	27.63	3.85	35.51	67.08
Lock/s	553.6	34.13	0	3.2
Messages/s	4810.10	3852.6	1309.4	8328.57
Comm. rate (MB/s)	8.17	1.65	0.79	14.10
Avg. msg. size (Bytes)	1698	429	606	1709

Table 5.1: Execution statistics for an 8-processor run on TreadMarks with UDP/Ethr communication subsystem

It can be seen that Jacobi exclusively uses barriers for synchronization. On the other hand, SOR uses Locks for synchronization more than other applications. TSP and 3D FFT mostly use locks and barrier, respectively, for synchronization. Jacobi has the highest computation to communication ratio while 3D FFT has the highest volume of messages exchanged in a unit of time. The average size of exchanged messages in ascending order belongs to TSP, Jacobi, SOR, and FFT. The data exchange rates in an ascending order belong to Jacobi, TSP, SOR, and FFT.

Overall Results

The overall execution times of these applications on an 8-node system with UDP/Ethr, MVIA/Ethr, UDP/Myri, and BVIA/Myri are shown in Fig. 5.9. The execution times are normalized with respect to the execution time on UDP/Ethr communication subsystem. The breakdown of these execution times are also shown. The total execution time is divided into four categories: *Computation*, *Communication*, *Wait*, and *Tmk Protocol*. The *Computation* category is the time spent in executing the application code. The *Communication* is the time spent for sending and receiving messages. For receive operations only the time spent on receiving a message after its arrival is counted (by either using a socket *select* call or by checking the status of the corresponding descriptor). The *Wait* category represents the time spent on waiting for arrival of messages. The *Tmk Protocol* category shows the time spent for the execution of TreadMarks code excluding the Communication and Wait time.

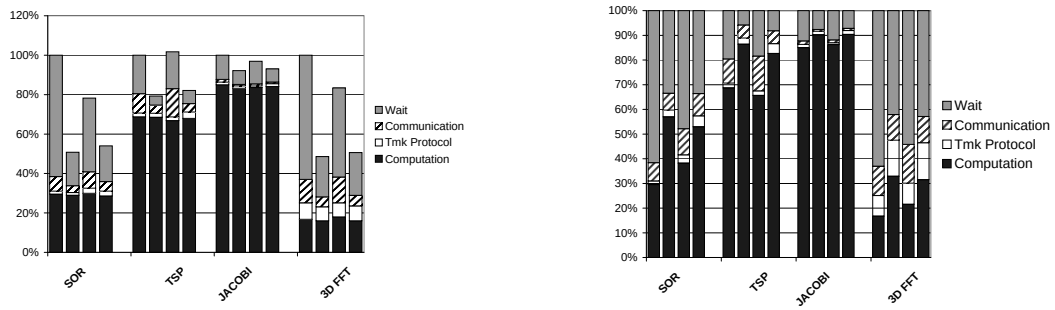

Figure 5.9: Overall execution times and their breakdowns for four applications on four implementations. For each application, the bars from left to right represent the results for UDP/Ethr, MVIA/Ethr, UDP/Myri, and BVIA/Myri communication subsystem, respectively. The left graph shows times normalized to the UDP/Ethr time. The right graph shows the percentage breakdown of different components.

It is to be noted that when the UDP/Ethr communication subsystem is replaced with MVIA/Ethr, the factors of improvement in Communication time of SOR, TSP, Jacobi, and 3D FFT are 2.12, 2.35, 1.96, 2.34, respectively. In addition to the reduction of communication time, the Wait time is also significantly reduced when MVIA replaces UDP. The factors of improvement for the overall execution time are 1.97, 1.26, 1.09, and 2.06 for SOR, TSP, Jacobi, and 3D FFT, respectively. It can be seen that there is a correlation between the rate of data exchange and the overall factor of improvement, i.e., the application exhibiting maximum data exchange rate gets maximum improvement with the new communication subsystem.

When the performance of UDP/Myri and BVIA/Myri are compared, it can be observed that the communication and wait times are also reduced. Although the communication rate is low for TSP, since the majority of exchanged messages are short (for which the UDP/Myri performs poorly) the improvement in the communication time is more significant in comparison with other applications. Factors of improvement for the overall execution time are 1.45, 1.24, 1.04, and 1.65 for SOR, TSP, Jacobi, and 3D FFT, respectively.

It can be observed that the performance of MVIA/Ethr and BVIA/Myri are comparable with MVIA/Ethr outperforming BVIA/Myri slightly. While both BVIA and MVIA perform equally for operations with large messages (as seen in the diff micro-benchmark), the latency for short messages is higher for BVIA implementation. This results in a small degradation of performance for BVIA/Myri in comparison with the MVIA/Ethr implementation.

However, UDP/Myri outperforms UDP/Ethr significantly for SOR and 3D FFT applications. The performance of these two systems for TSP and Jacobi are comparable. It can be observed that the amount of improvement or degradation when UDP/Ethr is replaced by UDP/Myri correlates with the average message size of the applications.

Effect of System Size

In order see the effect of the system size on the performance of TreadMarks implementations, we ran our applications on one to eight nodes. Figure 5.10 illustrates the execution times of our applications on different number of nodes. The speedup curves for these applications are shown in Figures 5.11 and 5.12.

It can be observed that the VIA implementations deliver better speedup compared to the UDP implementations. Sometimes the speedups achieved with the VIA implementations are quite significant. For example, when UDP/Ethr is used, SOR achieves a speedup of only 2.0 with eight nodes. It can be observed that increasing the number of nodes from four to eight doesn't increase the speedup significantly for UDP/Ethr. On the other hand, with MVIA/Ethr, the speedup is 3.95 with 8 nodes. On Myrinet network, the VIA implementation pushes the speedup for SOR to 3.72 (BVIA/Myri) compared to a speedup of 2.56 with the UDP implementation (UDP/Myri).

Across the applications, the speedups for both VIA implementations are very similar. The speedups for UDP implementations follow also a similar trend. It can be observed that Jacobi achieves a near-linear speedup. The high computation to communication ratio contributes to the scalability of Jacobi. Speedups of 7.35, 7.70, 7.24, and 7.47 are achieved on eight processors for UDP/Ethr, BVIA/Ethr, UDP/Myri, and BVIA/Myri, respectively. For TSP, speedups of 3.70 and 4.67 is observed for the UDP/Ethr and MVIA/Ethr implementations, respectively. The maximum speedups for TSP and UDP/Myri and BVIA/Myri are 3.64 and 4.51, respectively. 3D FFT has a very low computation to communication ratio with a large data exchange rate and a relatively large average message size (1709 bytes). All these contribute to a low speedup. The 3D FFT performs well when the number of nodes is a power of two. When UDP is used the speedup remains around one when up to eight nodes are used. By replacing UDP with VIA, maximum speedups of 2.36 and 2.27 are achieved for MVIA/Ethr and BVIA/Myri, respectively. It should be noted that increasing the number of nodes from four to eight increases the execution time for 3D FFT on UDP/Ethr and UDP/Myri implementations.

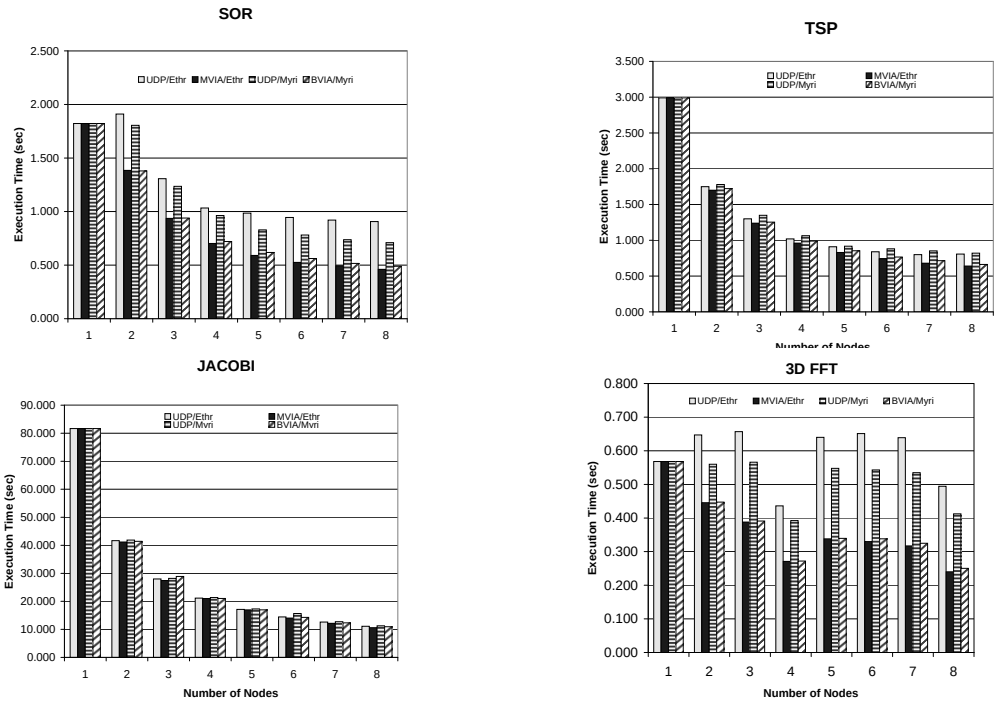

Figure 5.10: Execution times for four applications on four different communication subsystems as the number of nodes are varied from 1 to 8. The bars from left to right (for a given number of nodes) represent results for UDP/Ethr, MVIA/Ethr, UDP/Myri, and BVIA/Myri, respectively.

Effect of Application Size

The application execution times on an 8-node system for different problem sizes are shown in Fig. 5.13. The problem sizes used for SOR were: 2000×1000, 2000×2000, 2000×3000, and 2000×4000. For TSP, different problem sizes for 17, 18 and 19-city tours were used. For Jacobi the problem sizes of 500×500, 1000×1000, 1500×1500, and 2000×2000 were used. Problem sizes used for 3D FFT were $8 \times 8 \times 8$, $16 \times 16 \times 16$, $32 \times 32 \times 32$, and $64 \times 64 \times 64$.

The achieved factors of improvement in the overall execution time when VIA/Ethr is replaced by MVIA/Ethr are in the range of $1.86-2.05$, $1.10-1.26$, $1.02-1.05$, and $1.97-2.17$ for SOR, TSP, Jacobi, and 3D FFT, respectively. When UDP/Myri is used in comparison with BVIA/Myri, the factors of improvement are in the range of $1.45-1.96$, $1.18-1.28$, $1.02-1.13$, and $1.54-2.39$ for SOR, TSP, Jacobi, and 3D FFT, respectively. These numbers indicate that the VIA implementations are able to deliver better performance for a range of application sizes.

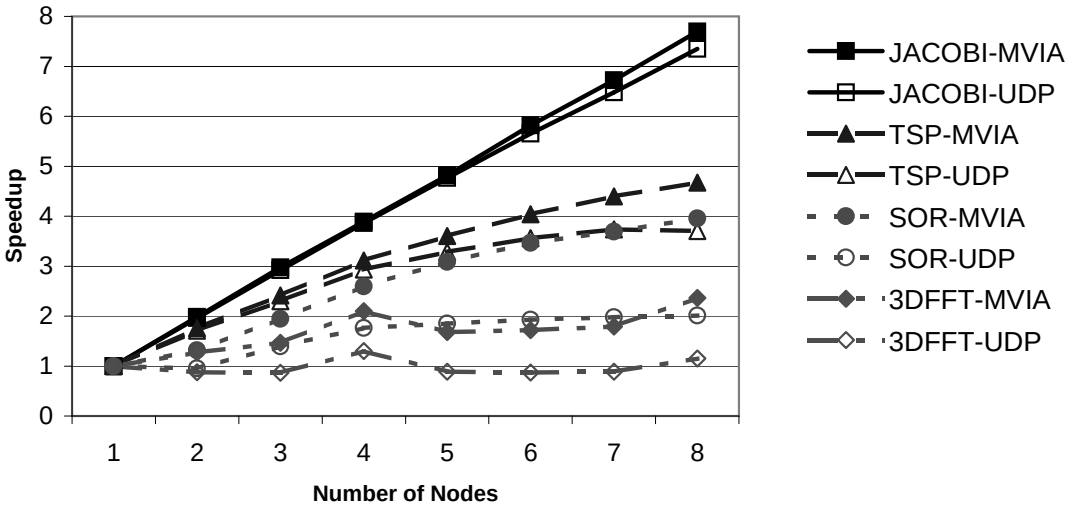

Figure 5.11: Speedups for the four applications on different number of nodes for MVIA/Ethr and UDP/Ethr (on Gigabit Ethernet).

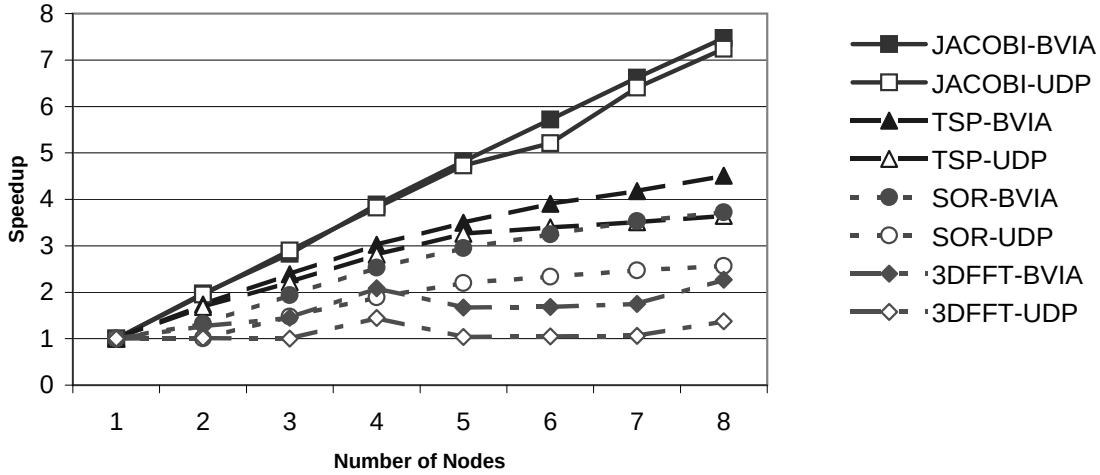

Figure 5.12: Speedups for the four applications on different number of nodes for BVIA/Myri and UDP/Myri (on Myrinet).

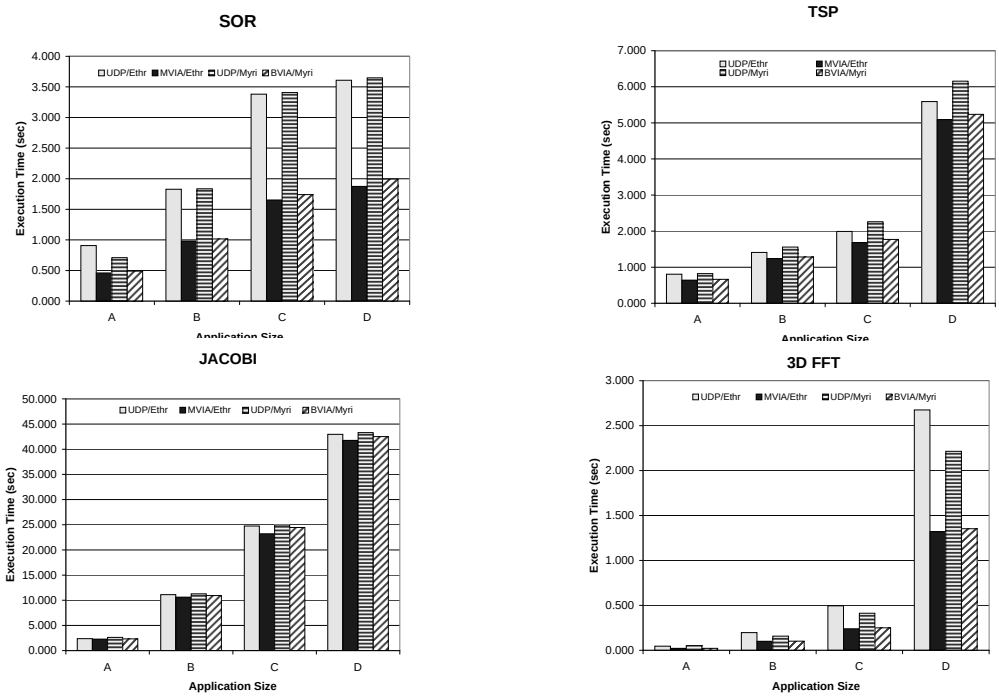

Figure 5.13: Effect of application size on execution times of applications with different problem sizes. The respective problem sizes (A, B, C, and D) are presented in the text. For each application size, the bars from left to right represent the results for UDP/Ethr, MVIA/Ethr, UDP/Myri, and BVIA/Myri communication subsystem, respectively.

5.6 Related Work

In this section we briefly discuss the related work. A comparison between the performance of PastSet Software DSM system using TCP/IP and VIA is discussed in [20]. In this work it is shown that by replacing TCP/IP by the MVIA implementation of VIA improves the performance of a few micro-benchmarks. The authors indicate that due to problems with the MVIA implementation they haven't been successful in designing and implementing the complete system. A few issues involved in taking advantage of low-latency high-bandwidth communication layers in Software DSM systems are discussed in [20]. The communication system used in this work is Fast Messages (FM) on Myrinet. In this work, a new mechanism called MultiView for providing small-size pages is proposed for avoiding false sharing, reducing the size of messages, and preventing excessive buffer copying. Methods for customizing network drivers for providing efficient support for asynchronous messages is discussed for the Windows-NT operating system.

5.7 Summary

In this chapter we discussed challenges involved in providing services required by the TreadMarks software DSM system by using the VIA features. We provided a systematic approach in bridging the gap between TreadMarks requirements and the functionality provided by VIA. We first identified all

mismatches between the functionalities of TreadMarks and the services provided by the VIA layer. Then we presented a set of schemes with various components to eliminate such mismatches. We also analyzed the performance trade-offs in implementing the schemes/components and identified the best ones. We discussed and evaluated our complete implementation of the required communication substrate on two different networks and for two different implementations of VIA. We have shown that with a careful design and implementation, the VIA communication architecture can lead to a reduction of the execution time of applications by a factor of up to 2.05 on an 8-node system.

CHAPTER 6

DESIGN ISSUES AND ALTERNATIVES IN SUPPORTING DISTRIBUTED MEMORY APPLICATIONS IN CLUSTERS

The IBM RS/6000 SP[2] system [8, 50, 51] (referred to as SP in the rest of this chapter) is a general-purpose scalable parallel system based on a distributed-memory, message-passing architecture. Configurations ranging from 2-node systems to 128-node systems are available from IBM. Larger configurations can be obtained via special order. The uniprocessor nodes are available with the latest Power2-Super (P2SC) microprocessors and the TB3 adapter. The SMP nodes are available with the 4 way, Power-PC 332MHz microprocessors and the TBMX adapter. The nodes are interconnected via a switch adapter to a high-performance, multistage, packet-switched network [30] for interprocessor communication capable of delivering bi-directional data-transfer rate of up to 160 MB/s between each node pair. Each node contains its own copy of the standard AIX operating system and other standard RS/6000 system software.

A portable parallel programming environment [29] is key to the success of high performance computing systems. Over the last few years, researchers have developed standard interfaces such as PVM [52, 57] and *Message Passing Interface* (MPI [23, 39]) to provide portability. These interfaces and standards attempt to abstract the intricate details of the hardware, software, and network characteristics from the application developer. However, the performance of applications depends heavily on the latency and bandwidth required for interprocessor communication, and synchronization across the nodes.

IBM SP systems support several communication libraries like MPI [39], MPL and LAPI [34, 45]. MPL, an IBM designed interface, was the first message passing interface developed by IBM on SP systems. Subsequently, after MPI became a standard it was implemented by reusing most of the infrastructure of MPL. This reuse allowed for SP systems to provide an implementation of MPI quite rapidly, but also imposed some inherent constraints on the MPI implementation which are discussed in detail in Section 6.1. In 1997, the LAPI library interface was designed and implemented on SP systems. The primary design goal for LAPI was to define an architecture with semantics that would allow efficient implementation on the underlying hardware and firmware infrastructure provided by SP systems. LAPI is a user space library, which provides a one-sided communication model thereby avoiding the complexities associated with two-sided protocols (like message matching, ordering, etc.).

In this chapter we describe the implementation of the MPI standard on top of LAPI (MPI-LAPI) to avoid some of the inherent performance constraints of the current implementation of MPI (native MPI) and to exploit the high performance of LAPI. There are some challenges involved

[2]IBM, RS/6000, SP, AIX, Power-PC, and Power2-Super are trademarks or registered trademarks of the IBM Corporation in the United States or other countries or both.

in implementing a 2-sided protocol such as MPI on top of a 1-sided user-level protocol such as LAPI. The major issue is finding the address of the receiving buffer. In 2-sided protocols, the sender does not have any information about the address of the receive buffer where the message should be copied into. There are some existing solutions to this problem. A temporary buffer can be used at the receiving side to store the message before the address of its destination is resolved. This solution incurs the cost of a data copy which increases the data transfer time and the protocol overhead especially for large messages. An alternative solution to this problem is using a rendezvous protocol, in which in response to the request from the sender, the receiver provides the receive buffer address to the sender, and then the sender can send the message. In this method the unnecessary data copy (into a temporary buffer) is avoided, but the cost of roundtrip control messages for providing the receive buffer address to the sender impacts the performance (especially for small messages) considerably. The impact is increased latency and control traffic. It is therefore important that a more efficient method be used for resolving the receive buffer address. In this chapter, we explain how the flexibility of the LAPI architecture is used to solve this problem in an efficient manner. Another challenge in implementing MPI on top of LAPI is to keep the cost of enforcing the semantics of MPI small so that the efficiency of LAPI is realized to the fullest. Another motivation behind our effort has been to provide better reuse by making LAPI the common reliable transport layer for other communication libraries. It should be noted that for this work we use the user-level LAPI which is the IBM propriety product (and not VIA).

This chapter is organized as follows: In Section 6.1, we detail the different messaging layers in the current implementation of MPI. In Section 6.2, we present an overview of LAPI and its functionality. In Section 6.3, we discuss different MPI communication modes and show how these modes are supported by using LAPI. In Section 6.4, we discuss different strategies that are used to implement MPI on top of LAPI and the various changes we made to improve the performance of MPI-LAPI. Experimental results including latency, bandwidth, and benchmark performance are presented in Section 6.5. Related work is discussed in Section 6.6. In Section 6.7, we outline some of our conclusions.

6.1 The Native MPI Overview

The protocol stack for the current implementation of MPI on SP systems is shown in Figure 6.1a. This protocol stack consists of several layers. The functions of each of the layers is described briefly below:

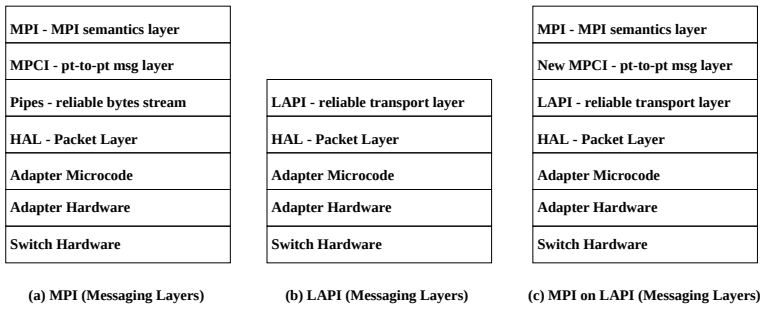

Figure 6.1: Protocol Stack Layering.

- The MPI layer enforces all MPI semantics. It breaks down all collective communication calls into a series of point-to-point message passing calls in MPCI (Message Passing Client Interface).

- The MPCI layer provides a point-to-point communication interface with message matching, buffering for early arrivals, etc. It sends data by copying data from the user buffer into the pipe buffers. The pipe layer then has responsibility for sending the data. Likewise data received by the pipe layer is matched, and if the corresponding receive has been posted, copied from the pipe buffers into the user buffer, otherwise the data is copied into an early arrival buffer (if the receive is not posted).

- The Pipes layer provides a reliable byte stream interface [47]. It ensures that data in the pipe buffers is reliably transmitted and received. This layer is also used to enforce ordering of packets at the receiving end pipe buffer if packets come out of order (the switch network has four routes between each pair of nodes and packets on some routes can take longer than other routes based on the switch congestion on the route). A sliding window flow control protocol is used. Reliability is enforced using an acknowledgment-retransmit mechanism.

- The HAL layer (packet layer, also referred to as the Hardware Abstraction Layer) provides a packet interface to the upper layers. Data from the pipe buffers are packetized in the HAL network send buffers and then injected into the switch network. Likewise packets arriving from the network are assembled in the HAL network receive buffers. The HAL network buffers are pinned down. The HAL layer handshakes with the adapter microcode to send/receive packets to/from the switch network.

- The Adapter DMAs the data from the HAL network send buffers onto the switch adapter and then injects the packet into the switch network. Likewise, packets arriving from the switch network into the switch adapter are DMAed onto the HAL network receive buffers.

The current MPI implementation, for the first and last 16K bytes of data, incurs a copy from the user buffer to the pipes buffer and from the pipe buffers to the HAL buffers for sending messages [47]. Similarly, received messages are first DMAed into HAL buffers and then copied into the pipe buffer. The extra copying of data is performed in order to simplify the communication protocol. These two extra data copies affect the performance of MPI. In the following sections we discuss LAPI (Fig. 6.1b) and explain how LAPI can replace the Pipes layer (Fig. 6.1c) in order to avoid the extra data copies and improve the performance of the message passing library.

6.2 LAPI Communication Model Overview

LAPI is a low level API designed to support efficient one-sided communication between tasks on SP systems [50]. The protocol stack of LAPI is shown in Figure 6.1b. An overview of the LAPI communication model (for LAPI_Amsend) is given in Figure 6.2 which has been captured from [45]. Different steps involved in LAPI communication functions are as follows. Each message is sent with a LAPI header, and possibly a user header (step 1). On arrival of the first packet of the message at the target machine, the header is parsed by a header handler (step 2) which is responsible for accomplishing three tasks (step 3). First, it must return the location of a data buffer where the packets of the message must be assembled. Second, it may optionally specify a pointer to a completion handler function which is called when all the packets have arrived in the buffer

location returned. Finally, if a completion handler function is provided, it also returns a pointer to data which is passed to the completion handler. The completion handler is executed after the last packet of the message has been received and copied into a buffer (step 4). In general, three counters may be used so that a programmer may determine when it is safe to reuse buffers and to indicate completion of data transfer. The first counter (org_cntr) is the origin counter, located in the address space of the sending task. This counter is incremented when it is safe for the origin task to update the origin buffer. The second counter, located in the target task's address space, is the target counter (tgt_cntr). This counter is incremented after the message has arrived at the target task. The third counter, the completion counter (cmpl_cntr) is updated on completion of the message transfer. This completion counter is similar to the target counter except it is located in the origin task's address space.

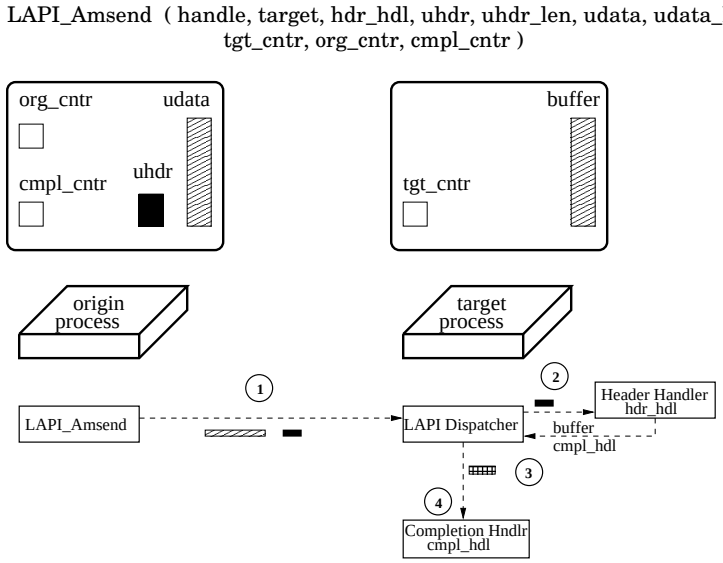

Figure 6.2: LAPI overview.

The use of LAPI functions may require that the origin task specify pointers to either functions or addresses in the target task address space. Once the address of the header handler has been determined, the sending process does not necessarily need to know the receive buffer address in the receiver's address space since the header handler is responsible for returning the receive buffer address. The header handler may, for example, interpret the header data as a set of tags which, when matched with requests on the receiving side, may be used to determine the address of the receive buffer. As we shall see, this greatly simplifies the task of implementing a two sided communication protocol with a one sided infrastructure. To avoid deadlocks, LAPI functions cannot be called from header handlers. The completion handlers are executed on a separate thread and can make LAPI calls.

LAPI functions may be broadly broken into two classes of functions. The first of these are communication functions using the infrastructure described above. In addition to these communication functions, there are a number of utility function provided so that the communication functions may

LAPI Function	Purpose
LAPI_Init	Initialize the LAPI subsystem
LAPI_Term	Terminate the LAPI subsystem
LAPI_Put	Data transfer function
LAPI_Get	Data transfer function
LAPI_Amsend	Active message send function
LAPI_Rmw	Synchronization read-modify-write
LAPI_Setcntr	Set the value of a counter
LAPI_Getcntr	Get the value of a counter
LAPI_Waitcntr	Wait for a counter to reach a value
LAPI_Address_init	Exchange addresses of interest
LAPI_Fence	Enforce ordering of messages
LAPI_Gfence	Enforce ordering of messages
LAPI_Qenv	Query the environment state
LAPI_Senv	Set the environment state

Table 6.1: LAPI Functions.

be effectively used. All the LAPI functions are shown in Table 6.1. For more information about LAPI we refer the reader to [45].

6.3 Supporting MPI on top of LAPI

The protocol stack used for the new MPI implementation is shown in Figure 6.1c. The PIPE layer is replaced by the LAPI layer. The MPCI layer used in this implementation is thinner than that of the native MPI implementation since it does not include the interface with the PIPE layer. In this section, we first discuss different communication modes defined by MPI and then explain how the new MPCI layer has been designed and implemented to support MPI on top of LAPI.

The MPI standard defines four communication modes: *Standard*, *Synchronous*, *Buffered*, and *Ready* modes [39]. These four modes are usually implemented by using two internal protocols called *Eager* and *Rendezvous* protocols. The translation of the MPI communication modes into these internal protocols in our implementation is shown in Table 6.2. The Rendezvous protocol is used for large messages to avoid the potential buffer exhaustion caused by unexpected messages (whose receives have not been posted by the time they reach the destination). The value of Eager Limit can be set by the user and has a default value of 4096 bytes. This value can be tuned based on the size of the buffer available for storing unexpected early arrival messages and the requirements of the applications.

In Eager protocol, messages are sent regardless of the state of the receiver. Arriving messages whose matching receives have not yet been posted are stored in a buffer called the *Early Arrival Buffer* until the corresponding receives is posted. If an arriving message finds a matching receive, the message is copied directly to the user buffer. In the Rendezvous protocol, a *Request_to_send* control message is first sent to the receiver which is acknowledged as soon as the matching receive gets posted. The message is sent to the receiver only after the arrival of this acknowledgment.

The blocking and nonblocking versions of the MPI communication modes have been defined in the MPI standard. In the blocking version, after a send operation, control returns to the

MPI Communication Mode	Internal Protocol
Standard	if (size < Eager Limit) Eager else Rendezvous
Ready	Eager
Synchronous	Rendezvous
Buffered	if (size < Eager Limit) Eager else Rendezvous

Table 6.2: Translation of MPI communication modes to internal protocols.

application only after the user data buffer can be reused by the application. In the blocking version of the receive operation, control returns to the application only when the message has been completely received into the application buffer. In the nonblocking version of send operations, control immediately returns to the user once the message has been submitted for transmission and it is the responsibility of the user to ensure safe reuse of its send buffer (by using MPI_WAIT or MPI_TEST operations). In the nonblocking version of receive, the receive is posted and control is returned to the user. It is the responsibility of the user to determine if the message has arrived. In the following sections we explain how the internal protocols and MPI communication modes are implemented by using LAPI.

6.3.1 Implementing the Internal Protocols

As mentioned in Section 6.2, LAPI provides one-sided operations such as LAPI_Put and LAPI_Get. LAPI also provides Active Message style operations through the LAPI_Amsend function. We decided to implement the MPI point-to-point operations on top of this LAPI active message infrastructure. The LAPI active message interface (LAPI_Amsend) function provides some enhancements to the active message semantics defined in GAM [55]. The LAPI_Amsend function allows the user to specify a header handler function to be executed at the target side once the first packet of the message arrives at the target. The header handler must return a buffer pointer to LAPI where the packets of the message must be reassembled. The ability of the target task of the LAPI_Amsend call to specify the destination address for the messages being sent, makes it ideally suited for implementing MPI-LAPI. The header handler is used to process the message matching and early arrival semantics, thereby avoiding the need for an extra copy at the target side. The header handler also allows the user to specify a completion handler function to be executed after all the packets of the message have been copied into the target buffer. The completion handler therefore serves to allow the application to incorporate the arriving message into the ongoing computation. In our MPI implementation the completion handler serves to update local state of marking messages complete, and possibly sending control message back to the sender. The LAPI_Amsend therefore provides the hooks to allow applications to get control when the first packet of a message arrives and when the complete message has arrived at the target buffer, making it ideal to be used as a basis for implementing MPI-LAPI. We explain below how the Eager and Rendezvous protocols have been implemented.

Implementing the Eager Protocol

In the MPI-LAPI implementation, LAPI_Amsend is used to send the message to the receiver (Fig. 6.3a). The message descriptions (such as message TAG and Communicator) are encoded in the user header which is passed to the header handler (Fig. 6.3b). Using the message description, the posted "Receive Queue" (Receive_queue) is searched to see if a matching receive has already been posted. If such a receive has been posted, the address of the user buffer is returned to LAPI and LAPI assembles the data into the user buffer. It should be noted that LAPI will take care of out of order packets and copy the data into the correct offset in the user buffer. If the header handler doesn't find a matching receive, it will return the address of an "Early Arrival Buffer" (EA_buffer) for LAPI to assemble the message into. (The buffer space is allocated if needed.) The header handler also posts the arrival of the message into the "Early Arrival Queue" (EA_queue). If the message being received is a Ready-mode message and its matching receive has not yet been posted, a fatal error is raised and the job is terminated. If the matching receive is found, the header handler also sets the function Eager_cmpl_hdl to be executed as the completion handler. The completion handler is executed, when the whole message has been copied into the user buffer, and the corresponding receive is marked as complete (Fig. 6.3c). In order to make the description of the implementation more readable, we have omitted some of the required parameters of the LAPI functions from the outlines.

(a)

```
Function Eager_send
    LAPI_Amsend(eager_hdr_hdl, msg_description, msg)
end Eager_send
```

(b)

```
Function Eager_hdr_hdl(msg_description)
    if (matching_receive_posted(msg_description)) begin
      completion_handler = Eager_cmpl_hdl
      return (user_buffer)
    end else begin
      if (Ready_Mode)
        Error_handler(Fatal, "Recv not posted")
      post msg_description in EA_queue
      completion_handler = NULL
      return (EA_buffer)
    endif
end Eager_hdr_hdl
```

(c)

```
Function Eager_cmpl_hdl(msg_description)
    Mark the msg as COMPLETE
end Eager_cmpl_hdl
```

Figure 6.3: Outline of the Eager protocol: (a) Eager send, (b) the header handler for the Eager send and (c) the completion handler for the Eager send.

```
(a)
    Function Request_to_send
        LAPI_Amsend(Request_to_send_hdr_hdl,
                    msg_description, NULL)
    end Request_to_send
(b)
    Function Request_to_send_hdr_hdl(msg_description)
        if (matching_receive_posted(msg_description)) begin
          completion_handler = Request_to_send_cmpl_hdl
          return (NULL)
        end else begin
          post msg_description in EA queue
          completion_handler = NULL
          return (NULL)
        endif
    end Request_to_send_hdr_hdl
(c)
    Function Request_to_send_cmpl_hdl(msg_description)
        LAPI_Amsend(Request_to_send_acked_hdr_hdl,
                    msg_description, NULL)
    end Request_to_send_cmpl_hdl
```

Figure 6.4: Outline of the first phase of the Rendezvous protocol: (a) Request to Send, (b) The Header handler for the request to send and (c) the completion handler for the request to send.

Implementing the Rendezvous Protocol

The Rendezvous protocol is implemented in two steps. In the first step a request_to_send control message is sent to the receiver by using LAPI_Amsend (Fig. 6.4). The second step is executed when the acknowledgment of this message is received (indicating that the corresponding receive has been posted). The message is sent by using LAPI_Amsend the same way the message is transmitted in Eager protocol (Fig. 6.3a). In the next section, we explain how these protocols are employed to implement different communication modes as defined in the MPI standard.

6.3.2 Implementing the MPI Communication Modes

Standard-mode messages which are smaller than the Eager Limit and Ready-mode messages are sent by using the Eager protocol (Fig. 6.5). Depending on whether the send is blocking or not, a wait statement (LAPI_Waitcntr) might be used to ensure that the user buffer can be reused.

Standard-mode messages which are longer than the Eager Limit and Synchronous-mode messages are transmitted by using the 2-phase Rendezvous protocol. Figure 6.6 illustrates how these sends are implemented. In the non-blocking version, the second phase of the send is executed in the completion handler which is specified in the header handler corresponding to the active message sent for acknowledging the Request_to_send message as shown in Figure 6.7.

```
Function StndShort_ready_send
    Eager_send
    if (blocking)
      Wait until Origin counter is set
end StndShort_ready_send
```

Figure 6.5: Outline of the standard send for messages shorter than the Eager Limit and the ready-mode send.

Buffered mode messages are transmitted using the same procedure as used for sending nonblocking standard messages. The only difference is that messages are first copied into a user specified buffer (defined by MPI_Buffer_attach). The receiver informs the sender when the whole message has been received so that the sender can free the buffer used for transmitting the message (Figure 6.8).

Figure 6.9 shows how blocking and non-blocking receive operations are implemented. It should be noted that in response to a Request_to_send message, a LAPI_Amsendis used to acknowledge the request. When this acknowledgment is received at the sender side of the original communication, the entire message will be transmitted to the receiver. If the original send operation is a blocking send, the sender is blocked until the Request_to_send message is marked as acknowledged and the blocking send will send out the message. If the original message is a nonblocking send, the message is sent out in the completion handler specified in the header handler of Request_to_send_acked (Fig. 6.7). When a message is found marked as COMPLETE, if the message has been stored in the EA_buffer, message will be copied into the user buffer.

```
Function StndLong_sync_send
    Request_to_send
    if (blocking) begin
      Wait until request_to_send is acknowledged
      Eager_send
      Wait until Origin counter is set
    endif
end StndLong_sync_send
```

Figure 6.6: Outline of the standard send for messages longer than the Eager Limit and the synchronous-mode send.

6.3.3 A Closer Look at the Implementation of MPI_Send and MPI_Recv

In this section, we examine how the procedures discussed in Section 6.3.2 are used to implement two major MPI communication primitives: MPI_Send and MPI_Recv. We also discuss the sequence

```
Function Request_to_send_acked_hdr_hdl
    if (blocking(msg_description))
      mark the request as acknowledged
    else
      completion_handler =
        Request_to_send_acked_cmpl_hdl
end Request_to_send_acked_hdr_hdl
Function Request_to_send_acked_cmpl_hdl
    Eager_send
end Request_to_send_acked_cmpl_hdl
```

Figure 6.7: Outline of receive for messages sent using the Rendezvous protocol.

```
Function Buffered_send
    Copy the msg to the attached buffer
    if (msg_size ≤ EagerLimit)
      Eager_send
    else
      Request_to_send
end Buffered_send
```

Figure 6.8: Outline of the buffered-mode send.

```
Function Receive
    if (found_matching_msg(EA_queue, msg_description))
      if (request_to_send) begin
          LAPI_Amsend(Request_to_send_acked,
                      msg_description, NULL)
      endif
    else
      Post the receive in Receive_queue
    if (blocking)
      Wait until msg is marked as COMPLETE
    if (this_msg_in_EA_buffer)
      Copy the msg from the EA_buffer to the user buffer
end Receive
```

Figure 6.9: Outline of receive for messages sent by the Eager protocol.

of actions taken at the sending and receiving tasks in detail. For clarity, we present the details of MPI_Send for messages shorter than the Eager Limit and other messages separately.

Figure 6.10 illustrates how the MPI_Send function is implemented for messages shorter than the Eager Limit. MPI_Send calls the Eager_send routine. The Eager_send routine uses a LAPI_Amsend call to send the message to the destination task. The corresponding header handler routine at the destination is set to be the Eager_hdr_hdl routine. The message description (which consists of information such as the communicator, tag, and the sender rank in the communicator) is also sent with the message to be passed to this header handler. When the message arrives at the destination task, Eager_hdr_hdl is executed. By using the message description passed to this routine, the queue of posted receives is searched to see if a matching receive has been already posted. If a matching receive is found, the address of the user buffer into which the message should be copied is returned to the LAPI subsystem. Otherwise, the address of an EA_buffer is returned and an entry in the EA_queue is created for this message. The Eager_cmpl_hdl routine is set to be executed as the completion handler. The buffer address returned from the header handler is used by the LAPI communication subsystem to copy the data. After the whole message has been copied (into the user buffer or an early arrival buffer) Eager_cmpl_hdl is executed. The only action taken in this completion handler is that the message is marked as COMPLETE (such that the matching receive can detect this condition).

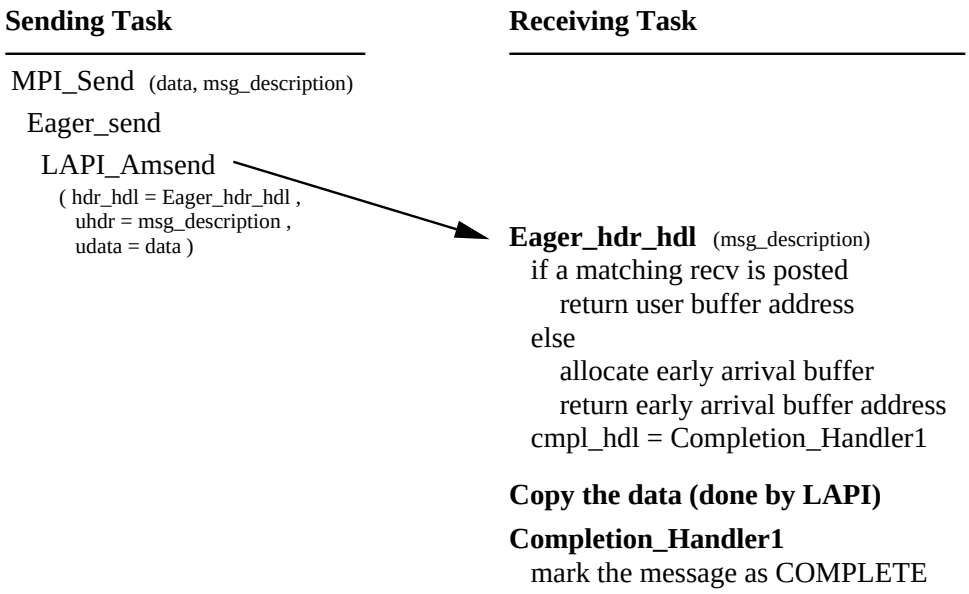

Figure 6.10: Outline of MPI_Send and the sequence of actions taken at the sending and receiving tasks when the message size is less than the Eager Limit.

Figure 6.11 illustrates how the MPI_Send function is implemented for messages whose size is greater than or equal to the Eager Limit. For these messages, MPI_Send calls the function Request_to_send to initiate the first phase of the Rendezvous protocol. The Request_to_send routine

uses a LAPI_Amsend call to send the description of the message to the destination task. It should be noted that the user data is not being sent in this message. The header handler to be executed at the destination task is set to the Request_to_send_hdr_hdl routine. When this routine is executed at the destination task, the posted messages queue is searched for a matching receive. If a matching receive is found, an acknowledgment is sent to the sender task. However, since calling LAPI functions in header handlers is not allowed, the completion handler needs to perform this operation. Therefore, the Request_to_send_cmpl_hdl completion handler is set to be executed as the completion handler. Since there is no data to be copied, a NULL pointer is returned to the LAPI subsystem. The Request_to_send_cmpl_hdl sends the acknowledgment by using a LAPI_Amsend call. When this acknowledgment arrives at the sender task, the user data is sent using a method similar to the one used for sending short messages. In cases where no matching receive is found, an entry in the EA_queue is created for the message. Since no other action needs to be taken and the completion handler and buffer address returned by the header handler are NULL. In these cases, the acknowledgment is sent to the sender task when a matching receive get posted. This point becomes clearer when MPI_Recv is discussed.

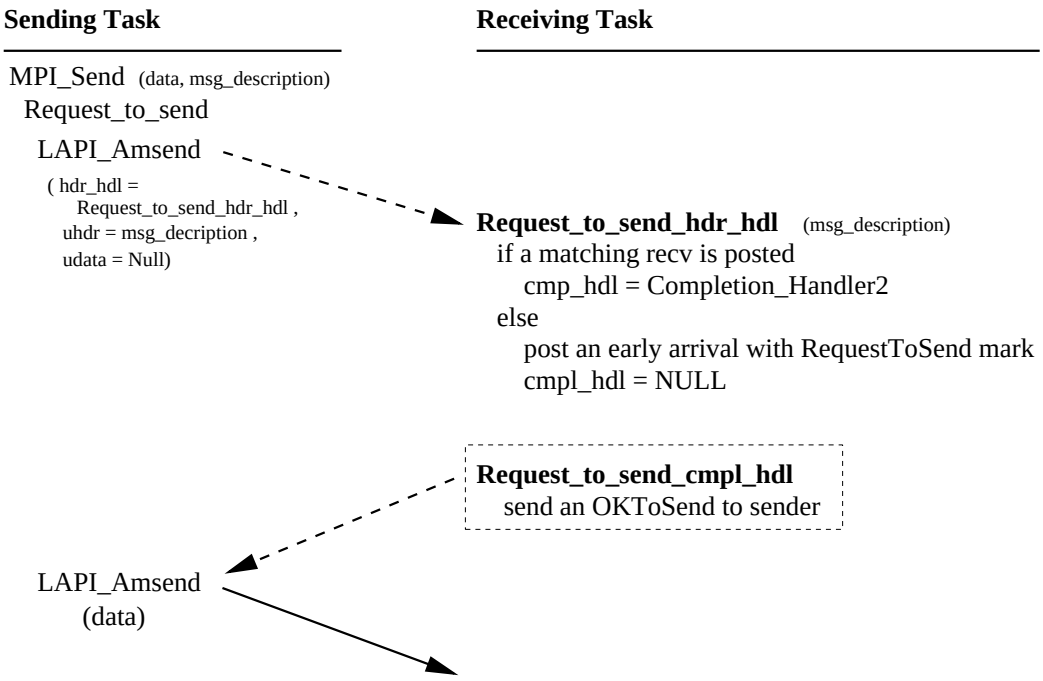

Figure 6.11: Outline of MPI_Send and the sequence of actions taken at the sending and receiving tasks when the message size is equal to or greater than the Eager Limit.

As illustrated in Fig. 6.12, whenever MPI_Recv is called, the EA_queue is searched for a matching message. If a message with matching descriptions is not found, a receive is posted (i.e., an entry in the Receive_queue with the description of the message is created). If a matching message is found, the description of the received message is checked to see if the received message is a Request_to_send message. If that's the case, an acknowledgment is sent back to the sender task by

Receiving Task

```
MPI_Recv (user_buffer, msg_description)

  if found a matching message
    if the message is a Request_to_send message
      send an acknowledgment message to the sender
    endif  /* Rendezvous */
  else
    post the receive
  wait until the message is marked as COMPLETE
  if message is in EA_buffer
    copy the message to the user buffer
```

Figure 6.12: Outline of MPI_Recv.

using a LAPI_Amsend call to initiate the second phase of the Rendezvous protocol at the sending task. In all cases, the status of the message is checked until it is marked as COMPLETE. If the message has been received in an EA_buffer, it is copied into the user buffer.

6.4 Optimizing the MPI-LAPI Implementation

In this section we first discuss the performance of the base implementation of MPI-LAPI which is based on the description outlined in Section 6.3. After discussing the shortcomings of this implementation, we present two methods to improve the performance of MPI-LAPI.

6.4.1 The Base MPI-LAPI

We compared the performance of our base implementation with that of LAPI itself. We measured the time to send a number of messages (with a particular message size) from one node to another node. Each time the receiving node would send back a message of the same size, and the sender node will send a new message only after receiving a message from the receiver. The number of messages being sent back and forth was long enough to make the timer error negligible. The granularity of the timer was less than a microsecond. LAPI_Put and LAPI_Waitcntr were used to send the message and to wait for the reply, respectively. The time for the MPI-LAPI implementation was measured in a similar fashion. MPI_Send and MPI_Recv were the communication functions used for this experiment. It should be noted that in all cases, the Rendezvous protocol was used for messages larger than the Eager Limit ($4K$ bytes). Figure 6.13 shows the measured time for messages of different sizes. We observed that message transfer time of the MPI-LAPI implementation was too high to be attributed only to the cost of protocol processing like message matching which are required for the MPI implementation but not for the 1-sided LAPI primitives.

6.4.2 MPI-LAPI with Counters

Careful study of the design and profiling of the base implementation showed that the cost of thread context switching required from the header handler to the completion handler was the major source of increase in the data transfer time. It should be noted that completion handlers are

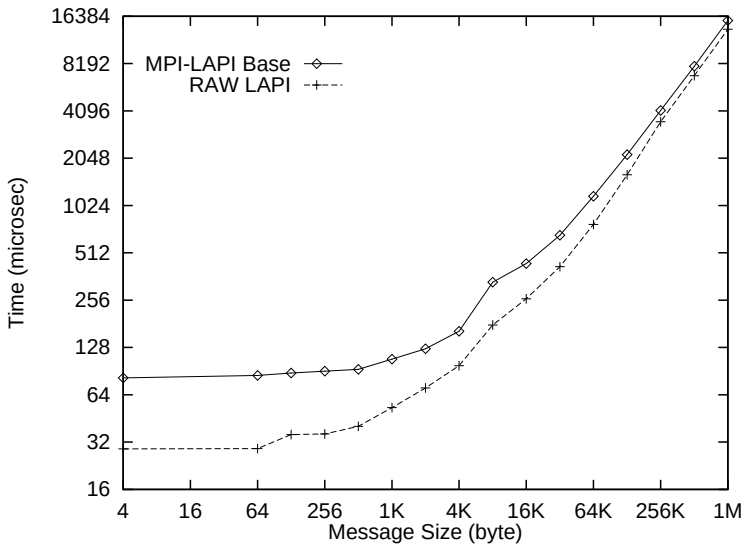

Figure 6.13: Comparison between the performance of raw LAPI and MPI-LAPI.

executed on a separate thread (Section 6.2) in LAPI. To verify this hypothesis, we modified the design such that we do not require the execution of completion handlers. As described in Section 6.3, when the Eager protocol is used, the only action taken in the completion handler is marking the message as completed (Fig. 6.3) such that the receive (or MPI_WAIT or MPI_TEST) can recognize the completion of the receipt of the message. LAPI provides a set of counters to signal the completion of LAPI operations. The target counter specified in LAPI_Amsend is updated (incremented by one) after the message is completely received (and the completion handler, if there exist any, has executed). We used this counter to indicate that message has been completely received. However, the address of this counter which resides at the receiving side of the operation should be specified at the sender side of the operation (where LAPI_Amsend is called). In order to take advantage of this feature, we modified the base implementation to use a set of counters whose addresses are exchanged among the participating MPI processes during initialization. By using these counters we avoided using the completion handler of messages sent through the Eager protocol. We could not employ the same strategy for the first phase of the Rendezvous protocol. The reception of the Request_to_send control messages at the receiving side does not imply that the message can be sent. If the receive has not yet been posted, the sender cannot start sending the message even though the Request_to_send message has been already received at the target. The time for the message transfer of this modified version is shown in Figure 6.14. As it can be observed, this implementation provided better performance for short messages (which are sent in Eager mode) compared to the base implementation. This experiment was solely performed to verify the correctness of our hypothesis.

6.4.3 MPI-LAPI Enhanced

The results in Figure 6.14 confirmed our hypothesis that the major source of overhead was the cost of context switching required for the execution of the completion handlers. We showed how

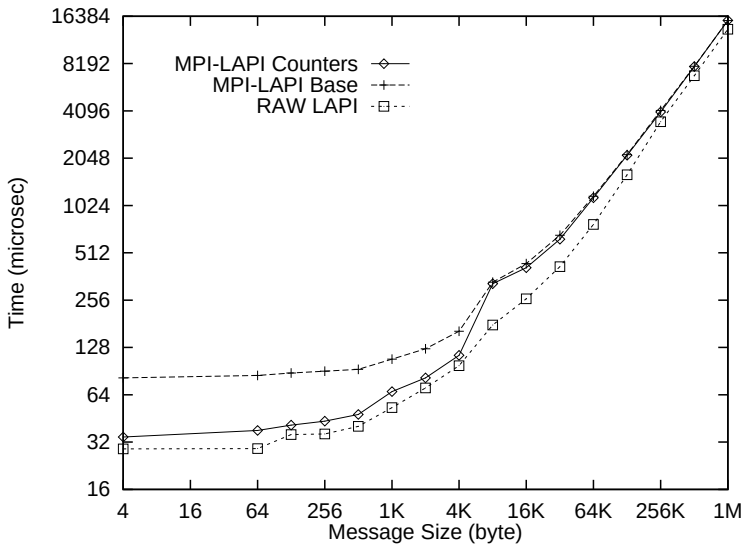

Figure 6.14: Comparison between the performance of raw LAPI and improved version of MPI-LAPI

we can avoid using completion handlers for messages which are sent in Eager mode. However, we still need to use completion handlers for larger messages (sent in Rendezvous mode). In order to avoid the high cost of context switching for all messages, we enhanced LAPI to include pre-defined completion handlers in the same context. In this modified version of LAPI, operations such as updating a local variable or a remote variable (which requires a LAPI function call), indicating the occurrence of certain events, were executed in the same context. The results of this version is shown in Figure 6.15. The time of this version of MPI-LAPI comes very close to that of the bare LAPI itself. The difference between the curves can be attributed to the cost of posting and matching receives required by MPI, and also the cost of locking and unlocking of the data structures used for these functions at the MPI level.

In the following section, we compare the latency and bandwidth of our MPI-LAPI Enhanced implementation with that of the native MPI implementation. We also explain the difference between the performance of these two implementations.

6.5 Performance Evaluation

In this section, we first present a comparison between the native MPI and MPI-LAPI (the Enhanced version) in latency and bandwidth. Then we compare the results obtained from running the NAS benchmarks using MPI-LAPI with those obtained from running NAS benchmarks using the native MPI. In all of our experiments we used a SP system with Power-PC 332MHz nodes and the TBMX adapter. The Eager Limit was set to $4K$ bytes for all experiments.

6.5.1 Latency and Bandwidth

We compared the performance of MPI-LAPI with that of the native MPI available on SP systems. The time for message transfer was measured by sending messages back and forth between two

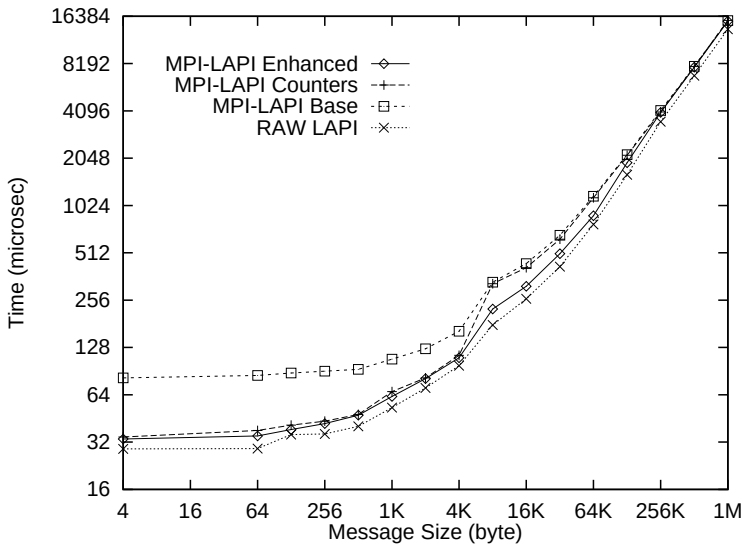

Figure 6.15: Comparison between the performance of raw LAPI and different versions of MPI-LAPI.

nodes as described in Section 6.4. The MPI primitives used for these experiments were MPI_Send and MPI_Recv. The Eager Limit for both systems was set to $4K$ bytes. To measure the bandwidth, we repeatedly sent messages out from one node to another node for a number of times and then waited for the last message to be acknowledged. We measure the time for sending these back to back messages and stop the timer when the acknowledgment of the last message is received. The number of messages being sent is large enough to make the time for transmission of the acknowledgment of the last message negligible in comparison with the total time. For this experiment we used MPI_Isend and MPI_Irecv primitives.

Figure 6.16 illustrates the time of MPI-LAPI and the native MPI for different message sizes. The time of MPI-LAPI for very short messages is slightly higher than that of the native MPI. This increase is in part due to the extra parameter checking by LAPI which, unlike the internal Pipes interface, is an exposed interface. The difference between the size of the packet headers in these two implementations is another factor which contributes to the slightly increased latency. The size of headers in the native MPI is 16 bytes, and the size of headers for MPI-LAPI is 48 bytes. It can be also observed that for messages larger than 256 bytes, the latency of MPI-LAPI becomes less than that of the native MPI. An improvement of up to 17.3% was measured. As mentioned earlier, unlike the native implementation of MPI, in the MPI-LAPI implementation messages are copied directly from the user buffer into the NIC buffer and vice versa. Avoiding the extra data copying helps improve the performance of the MPI-LAPI implementation.

The obtainable bandwidth of the native MPI and MPI-LAPI is shown in Figure 6.17. It can be seen that, for a wide range of message sizes, the bandwidth of MPI-LAPI is higher than that of the native MPI. For $64K$ byte messages, MPI-LAPI achieves a bandwidth of $83.35MB/sec$ which indicates a 20.9% improvement in comparison with the $68.93MB/sec$ bandwidth obtained by using the native MPI.

For measuring the time required for sending messages from one node to another node in interrupt mode, we used a method similar to the one used for measuring latency. The only difference was

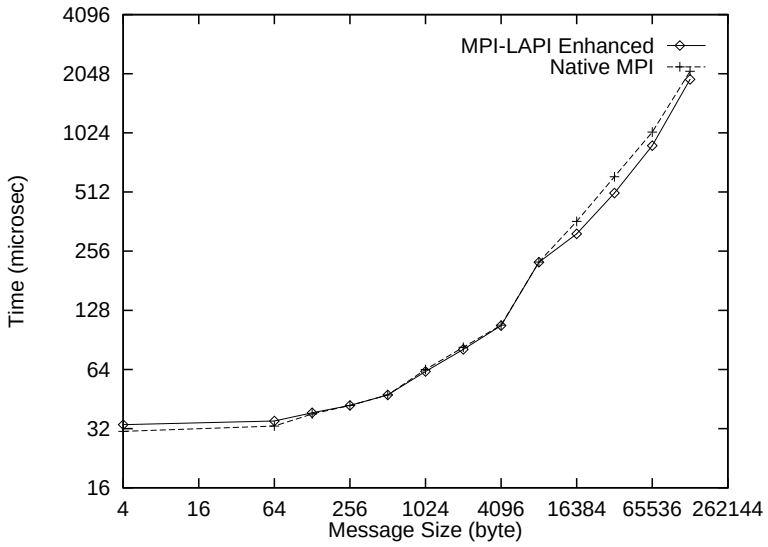

Figure 6.16: Comparison between the performance of the native MPI and MPI-LAPI.

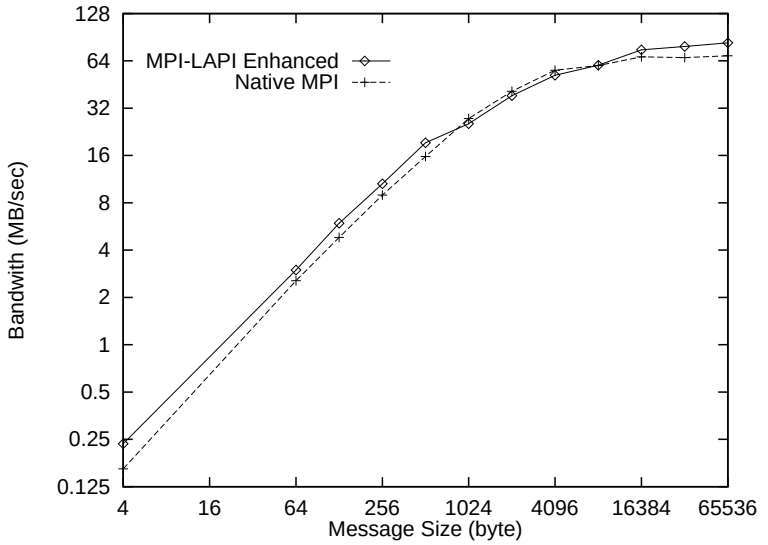

Figure 6.17: Comparison between the performance of the native MPI and MPI-LAPI.

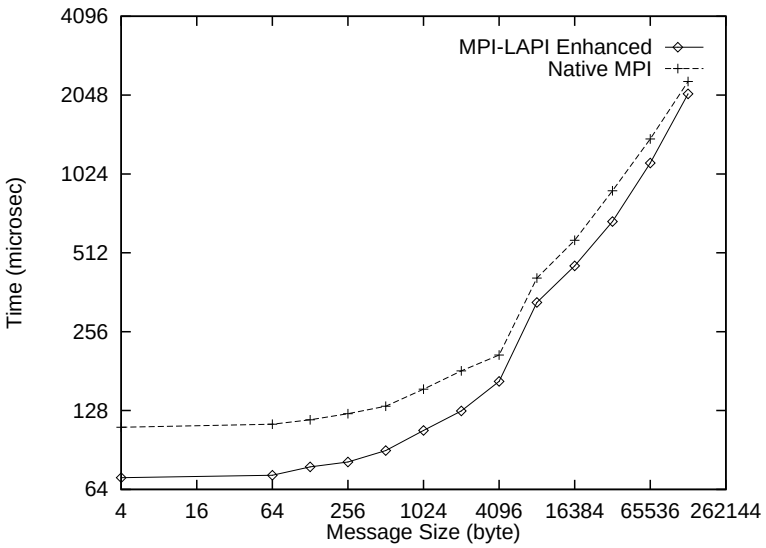

Figure 6.18: Comparison between the performance of the native MPI and MPI-LAPI in interrupt mode.

that the receiver would post the receive (using MPI_Irecv) and check the content of the receive buffer until the message has arrived. Then it would send back a message with the same size. The results of our measurements are shown in Figure 6.18. It can be seen that MPI-LAPI performs consistently and considerably better than the native MPI implementation. For short messages of 4 bytes an improvement of 35.8% is observed. The native MPI performs poorly in this experiment. One reason behind the poor performance of the native MPI is the hysteresis scheme used in it. In the interrupt handler of the native MPI, the interrupt handler waits for a certain period of time to see if more packets are coming to avoid further interrupts. If more are coming then they increase the time the interrupt handler waits in the loop. The value of this waiting period can be set by the user. LAPI does not use any such hysteresis in its interrupt handler and thus, provides better performance.

6.5.2 NAS Benchmarks

In this section we present the execution times of programs from the NAS benchmarks for the native MPI and MPI-LAPI. NAS Parallel Benchmarks (version 2.3) consist of eight benchmarks written in MPI. These benchmarks were used to evaluate the performance of our MPI implementation in a more realistic environment. We used the native implementation of MPI and MPI-LAPI to compare the execution times of these benchmarks on a four-node SP system. The benchmarks were executed several times. The best execution time for each application was recorded.

As presented in Table 6.3, the MPI-LAPI performs consistently better than the native MPI. Improvements of 1.9%, 4.1%, 4.6%, 5.1% and 13.8% were obtained for LU, IS, CG, BT and FT benchmarks, respectively. The percentages of improvement for EP, MG, and SP were less than 1.0%.

NAS Benchmark	Improvement(%)
BT	5.1
CG	4.6
EP	0.1
FT	13.8
IS	4.1
LU	1.9
MG	0.3
SP	0.0

Table 6.3: The percentage of improvement for NAS Benchmarks

6.6 Related Work

Previous work on implementing MPI on top of low-level one-sided communication interfaces include (a) the effort at Cornell in porting MPICH on top of their GAM (generic active message) implementation on the SP [26], and (b) the effort at University of Illinois in porting MPICH on top of the FM (fast messages) communication interface on a workstation cluster connected with the Myrinet network [38]. In both cases the public domain version of MPI (MPICH [32]) has been the starting point of these implementations. In the MPI implementation on top of AM, short messages are copied into a retransmission buffer after they are injected into the network. Lost messages are retransmitted from the retransmission buffers. The retransmission buffers are freed when a corresponding acknowledged is received from the target. Short messages therefore require a copy at the sender side. The other problem is that for each pair of nodes in the system a buffer should be allocated which limits scalability of the protocol. MPI-LAPI implementation avoids these problems (which degrade the performance) by using the header handler feature of LAPI. Unlike MPI-LAPI, the implementation of MPI on AM described in [26] does not support packet arrival interrupts which impacts performance of applications with communication behavior that is asynchronous. In the implementation of MPI on top of FM [27, 38], FM was modified to avoid extra copying at the sender side (gather) as well as the receive side(upcall). FM has been optimized for short messages.

MPI on the I-WAY [33] is another available implementation of the MPI standard. However, in this implementation the emphasis has been on features such as authentication and multi-method communication rather than the pure performance of the implementation. MPI-BIP is another implementation of the MPI standard on top of BIP [43]. BIP is an API designed, and implemented for the Myrinet network. BIP ensures reliable and ordered transmission of messages in the absence of network fault.

6.7 Summary

In this chapter, we have presented how the the MPI standard is implemented on top of LAPI for the IBM SP system. The details of this implementation and the mismatches between the MPI standard requirements and LAPI functionality have been discussed. We have also shown how LAPI can be enhanced in order to make the MPI implementation more efficient. The flexibility provided

by having header handlers and completion handlers makes it possible to avoid any unnecessary data copies. The performance of MPI-LAPI is shown to be very close to that of bare LAPI and the cost added because of the MPI standard semantics enforcement is shown to be minimal. MPI-LAPI performs comparably or better than the native MPI in terms of latency and bandwidth. MPI-LAPI also outperforms the native MPI for NAS benchmarks.

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

In this chapter we summarize the contributions of this thesis and suggest some directions for future research.

7.1 Summary of Research Contributions

This thesis makes the following major contributions for design and development of efficient communication subsystems for clusters.

1. **Design, Development, and Performance Evaluation of Several Important Components of the Virtual Interface Architecture (VIA):** High bandwidth, low latency communication subsystems in general and the Virtual Interface Architecture in particular are made of several components such as virtual-to-physical address translation component, mechanisms for hosts informing NICs about outstanding operations (i.e. doorbells), and mechanisms for waiting for completion of any data transfer operation among a given group of operations (i.e. completion queues). Several design alternatives for implementing these components are proposed. These design alternatives are evaluated on different clusters.

2. **Design, Development, and Performance Evaluation of a Prototype Implementation of the Virtual Interface Architecture (VIA):** A prototype implementation of VIA is developed on IBM SP-connected NT clusters. Impact of design choices for components of VIA on the overall performance is evaluated. It is shown how VIA can be implemented in the most efficient manner under certain requirements and restrictions. The bottlenecks for reaching a better performance are identified. Furthermore, it is shown how adding several hardware improvements affect the overall performance of the communication subsystem.

3. **The VIBe Micro-Benchmark Suite for Evaluating Communication Subsystems:** A framework for evaluating different VIA design choices and for obtaining insight about the design choices made in a particular implementation of VIA and their impact on the performance is developed. A new micro-benchmark suite called VIBe is designed and implemented. This suite consists of several micro-benchmarks which are divided into three major categories: non-data transfer related micro-benchmarks, data transfer related micro-benchmarks, and client/server micro-benchmarks. By using the new benchmark suite, the performance of VIA implementations is evaluated under different communication scenarios and with respect to the implementation of different components and attributes of VIA.

4. **Design and Implementation of TreadMarks over VIA:** A thin and efficient substrate over VIA is developed such that applications using the popular TreadMarks DSM package can take advantage of the enhanced communication performance of VIA. The mismatches between the communication requirements by TreadMarks and the services provided by VIA are identified. A set of schemes to eliminate such mismatches is proposed. These schemes include connection setup, buffer management, advance posting of descriptors for unexpected messages, and alternative designs to handle asynchronous messages. Different design alternatives for enhancing some VIA functions (such as the VIA Notify mechanism) are proposed such that the new substrate is implemented with low overhead. The best set of alternatives are derived and implemented on two enhanced implementations of VIA (MVIA and Berkeley VIA) on two different networking technologies, Gigabit Ethernet and Myrinet, respectively.

5. **Design and Implementation of MPI over LAPI:** Since a large number of high performance applications are written (and being written) in the distributed memory programming model by using the communication primitives provided by the MPI standard, it is crucial to implement MPI on top of high performance communication subsystems. It is explained how the high performance of the LAPI communication library is exploited in order to implement the MPI standard more efficiently than the existing MPI. Unnecessary data copies at both the sending and receiving sides are avoided in order to improve the performance. The resolution of problems arising from the mismatches between the requirements of the MPI standard and the features of LAPI is discussed. As a result of this exercise, certain enhancements to LAPI are identified and implemented to enable an efficient implementation of MPI on LAPI.

7.2 Suggestions for Future Research

The topic discussed in this thesis is a fertile area for further research with immediate impact in the computer industry. Many interesting and practical problems in this area still wait for solutions. In this section, we describe some of the problems and provide a few suggestions.

- **Coherence Protocols on top of VIA:** We studied how a popular software DSM system (i.e. TreadMarks) can be implemented on top of VIA. However, in our work we focused on supporting DSM systems on top of VIA without modifying the software DSM system and related coherence protocols. Because of the importance of VIA and its potential in becoming the communication standard for System Area Networks (SANs) it is very important to study how coherence protocols can be developed such that they can take advantage of the features of VIA. How to cost-effectively support a coherence protocol on top of VIA is a new and exciting area for future study with potentially a strong impact on the computer industry.

- **Other Programming Models:** Distributed memory and distributed shared memory programming models are the two most important programming models for clusters. We studied how these programming models can be supported on top of high performance user-level communication protocols. However, there are other programming models such as the Get/Put model [19] and Global Arrays [41]. It is interesting to see how VIA can be used for supporting such programming models. VIA Remote Direct Memory Access (RDMA) operations (i.e. RDMA Read and RDMA Write) seem to be the features that can be used for implementing operations supported by Get/Put and Global Arrays models.

- **Application Traces for VIBe:** VIBe micro-benchmarks suite is made of synthetic micro-benchmarks. It evaluates the performance of VIA implementations under various scenarios

which can be set by the user. It is very interesting to add a micro-benchmark to the suite which uses traces of real applications. The communication patterns of various applications can be gathered and then fed to this micro-benchmark. Various traces can be obtained from different types of applications. By using such an approach VIA implementations can be evaluated under a more realistic set of conditions.

- **InfiniBand Architecture (IBA):**

  "Today's computing model is becoming more distributed as companies work to meet the growing demands of the Internet economy. The demands of the Internet and distributed computing are challenging the scalability, reliability, availability, and performance of servers. To meet this demand a balanced system architecture with equally good performance in the memory, processor, and input/output (I/O) subsystems is required. The InfiniBand Architecture (IBA) will de-couple the I/O subsystem from memory by utilizing channel based point to point connections rather than a shared bus, load and store configuration" [2].

  The basic IBA concepts are very similar to those of VIA. The same concepts such as work queues, doorbells, completion queues, and descriptors are used in both VIA and IBA. In addition to the features present in the VIA specification, there are several new features presented in the IBA specifications: RDMA atomic operations, support for connection-less data transfers, virtual lanes, service levels, and global addressing. We believe that many design alternatives proposed in our work can be applied to IBA implementations. However, the performance and requirements of these alternatives should be reevaluated under new scenarios and with respect to the impact of the proposed approaches on the implementation of new features.

  The high performance communication subsystems as discussed in this dissertation have been used only for inter-processor communication. With the introduction of IBA, the need for evaluation of these communication subsystems for I/O operations is felt more than ever. Although a similar communication architecture is going to be used for I/O data transfers, the difference between characteristics of I/O data transfers and those of inter-processor data transfers may lead to different solutions. It is interesting to study the Characteristics of I/O data transfers in this context.

- **Providing Quality of Service (QoS) on top of VIA and IBA:** VIA does not support any form of quality of service. In order to support applications with such requirements it's essential to provide some kind of quality of service on top of VIA. If the semantics of VIA is to preserved and no additional features are added, a substrate on top of VIA is required to provide QoS. Enhancing VIA such that it provides QoS needs to be investigated. The results of such a study can be used to influence the future versions of the protocol. Similarly, providing QoS on IBA needs to be investigated. It is interesting to see if the service levels specified by IBA are sufficient for providing QoS.

- **Data Centers:** Clusters have become increasingly popular for building Data Centers. Data centers are essentially a collection of servers and personal computers which serve the requests of the Internet applications. These data centers contain hundreds of computing nodes and storage devices. High performance communication subsystems can be deployed in such centers to improve the performance. Some of the important related research issues which should be addressed are the topology of the interconnections and routing schemes used in such systems. It will be interesting to study and see if current topologies and routing schemes are

efficient enough for interconnecting hundreds of computing nodes and storage devices in such environments.

- **Commercial Workloads:** High performance communication subsystems have been studied and evaluated by the industrial and research communities. However, most of these studies have been carried out for high performance applications. By the increasing popularity of the high performance communication subsystems for data centers, it is crucial that the performance of these systems is evaluated under commercial workloads which are seen in such environments. It is interesting to see if such workloads show any characteristic which may differentiate them from high performance computing workloads.

BIBLIOGRAPHY

[1] GigaNet Corporations. http://www.giganet.com/.

[2] InfiniBand Trade Association. http://www.infinibandta.org/.

[3] M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via/.

[4] MPI/PRO. http://www.mpi-softtech.com/.

[5] MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/.

[6] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[7] Virtual Interface Architecture Specification. http://www.viarch.org/.

[8] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M.Snir. SP2 System Architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[9] C. Amza, A. L. Cox, et al. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[10] T. Anderson, D. Culler, and Dave Patterson. A Case for Networks of Workstations (NOW). *IEEE Micro*, pages 54–64, Feb 1995.

[11] ATM Forum. *ATM User-Network Interface Specification, Version 3.1*, September 1994.

[12] D. H. Bailey, E. Barszcz, L. Dagum, and H.D. Simon. NAS Parallel Benchmark Results. Technical Report 94-006, RNR, 1994.

[13] M. Banikazemi, B. Abali, and D. K. Panda. Comparison and Evaluation of Design Choices for Implementing the Virtual Interface Architecture (VIA). In *Proceedings of the CANPC workshop (held in conjunction with HPCA Conference)*, Jan. 2000.

[14] M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda. Implementing Efficient MPI on LAPI for IBM RS/6000 SP Systems: Experiences and Performance Evaluation. In *Proceedings of the 13th International Parallel Processing Symposium*, pages 183–190, April 1999.

[15] M. Banikazemi, J. Liu, S. Kutlug, A. Ramakrishnan, P. Sadayappan, and H. Shahand D. K. Panad. VIBe: A Microbenchmark Suie for Evaluating Virtual Interface Architecture (VIA) Implementations. Technical Report OSU-CISRC-10/00-TR20, Dept. of Computer and Information Science, The Ohio State University, Oct 2000. Submitted to International Parallel and Distributed Processing Symposium (IPDPS'2001).

[16] M. Banikazemi, J. Liu, D. K. Panda, and P. Sadayappan. Implementing TreadMarks over VIA on Myrinet and Gigabit Ethernet: Challenges, Design Experience, and Performance Evaluation. Technical Report OSU-CISRC-07/00-TR15, Dept. of Computer and Information Science, The Ohio State University, July 2000.

[17] M. Banikazemi, V. Moorthy, L. Herger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP Switch-Connected NT Clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 33–42, May 2000.

[18] M. Banikazemi, D. K. Panda, and P. Sadayappan. Implementing treadmarks on virtual interface architecture (via): Design issues and alternatives. In *Proceedings of the Nineth Workshop on Scalable Shared Memory Multiprocessors*, 2000.

[19] R. Barriuso and A. Knies. *SHMEM User's Guide.* Cray Research Inc., 1994.

[20] John Markus Bjoerndalen, Otto J. Anshus, Brian Vinter, and Tore Larsen. Comparing the performance of the pastset distributed memory system using tcp/ip and m-via. In *Proceedings of the 2nd International Workshop on Software Distributed Shared Memory*, 2000.

[21] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.

[22] N. J. Boden, D. Cohen, et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–35, Feb 1995.

[23] J. Bruck et al. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. *Journal of Parallel and Distributed Computing*, pages 19–34, Jan 1997.

[24] P. Buonadonna, J. Coates, S. Low, and D.E. Culler. Millennium Sort: A Cluster-Based Application for Windows NT using DCOM, River Primitives and the Virtual Interface Architecture. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[25] P. Buonadonnaa, A. Geweke, and D.E. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of the Supercomputing (SC)* , pages 7–13, Nov. 1998.

[26] C. Chang, G. Czajkowski, C. Hawblitzel, and T. Von Eicken. Low Latency Communication on the IBM RISC System/6000 SP. *Supercomputing 96*, 1996.

[27] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[28] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 262–273, 1993.

[29] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware-Software Approach*. Morgan Kaufmann, March 1998.

[30] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. The IEEE Computer Society Press, 1997.

[31] E. W. Felten, R. A. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. N. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *International Symposium on Computer Architecture (ISCA)*, pages 296–307, 1996.

[32] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[33] J. Geisler I. Foster and S. Tuecke. MPI on the I-WAY: A Wide-Area, Multimethod Implementation of the Message Passing Interface. In *Proceedings of the Second MPI Developer's Conference*, pages 10–17, 1996.

[34] IBM. *PSSP Command and Technical Reference - LAPI Chapter*. IBM, 1997.

[35] Ayal Itzkovitz, Assaf Schuster, and Yoram Talmor. Harnassing the power of fast low-latency networks for software dsms. In *First Workshop on Software Distributed Shared Memory*, 1999.

[36] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, Jan. 1994.

[37] Sencer Kutlug. Performance Evaluation and Analysis of User Level Networking Protocols in Clusters. MS Thesis, The Ohio State University, June 2000.

[38] M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, pages 4–18, Jan 1997.

[39] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[40] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[41] J. Nieplocha, R. J. Harrison, and R. L. Littlefield. Global arrays: A portable "shared memory" programming model for distributed memory computers. In *Supercomputing 94*, 1994.

[42] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[43] Loc Prylli and Bernard Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet. *In Proceedings of the International Parallel Processing Symposium Workshop on Personal Computer Based Networks of Workstations*, 1998. http://lhpca.univ-lyon1.fr/.

[44] Ioannis Schoinas and Mark D. Hill. Address Trnslation Mechanisms in Network Interfaces. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, Feb. 1998.

[45] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and Experience with LAPI - a New High-Performance Communication Library for the IBM RS/6000 SP. In *Proceedings of the International Parallel Processing Symposium*, March 1998.

[46] Hemal V. Shah, Calton Pu, and Rajesh S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *Proceedings of the CANPC workshop (held in conjunction with HPCA Conference)*, pages 91–107, 1999.

[47] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[48] E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the Performance Potential of the Virtual Interface Architecture. In *Proceedings of the International Conference on Supercomputing*, June 1999.

[49] R. Stets, S. Dwarkadas, L. Kontothanassis, U. Rencuzogullari, and M. L. Scott. The Effect of Network Total Order, Broadcast, and Remote-Write Capability on Network-Based Shared Memory Computing. In *Proceedings of International Symposium on High-Performance Computer Architecture*, 2000.

[50] C. B. Stunkel, D. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao. The SP1 High Performance Switch. In *Scalable High Performance Computing Conference*, pages 150–157, 1994.

[51] C. B. Stunkel, D. G. Shea, B. Abali, et al. The SP2 High-Performance Switch. *IBM System Journal*, 34(2):185–204, 1995.

[52] V. S. Sunderam. PVM: A Framework for Parallel and Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[53] P. G. Viscarola and W. A. Mason. *Windows NT Device Driver Development.* Macmillan Technical Publishing, 1999.

[54] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.

[55] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.

[56] M. Welsh, A. Basu, and T. von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of Hot Interconnects V*, Aug. 1997.

[57] H. Zhou and A. Geist. LPVM: A Step Towards Multithread PVM. Technical report, Oak Ridge National Laboratory, 1995.