Design Alternatives for Virtual Interface Architecture (VIA) and an Implementation on IBM Netfinity NT Cluster *

Mohammad Banikazemi† Bulent Abali‡ Lorraine Herger ‡ Dhabaleswar K Panda†

†Network-Based Computing Laboratory
Dept. of Computer and Information Science
The Ohio State University
Columbus, OH 43210
email:{banikaze, panda}@cis.ohio-state.edu

‡System Design & Performance
IBM T.J. Watson Research Center
P.O.Box 218
Yorktown Hts, NY 10598
email:{abali, herger}@watson.ibm.com

Abstract

The Virtual Interface Architecture (VIA) specification has been developed to standardize user-level network interfaces that provide low latency, high bandwidth communications. Few hardware and software implementations of VIA exist. Since the VIA specification is flexible, different choices exist for implementing various components of VIA such as doorbells, address translation methods, and completion queues. Although previous studies have evaluated the overall performance of different VIA implementations, there has not been a comparative study on the performance of VIA components. In this paper, we evaluate and compare the performance of different implementations of essential VIA components. We discuss the pros and cons of each design approach and describe the required support for implementing each of them. The IBM SP Switch-Connected NT cluster is one the newest clustering platforms available. In this paper, we discuss an experimental implementation of the Virtual Interface Architecture for this platform. We discuss different design issues involved in this implementation. In particular, we explain how the virtual-to-physical address translation is 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. We provide a comprehensive performance evaluation study. 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 101.4 *MBytes/s and the one-way latency for short messages is* 18.2 *microseconds. It is to be noted that the VIA implementation presented in this paper is not a part of any IBM product and no assumptions should be made regarding its availability as a product in the future.*

Keywords: Cluster Computing, Virtual Interface Architecture, Performance Evaluation, Scalable Architecture.

*This research is supported in part by an IBM Cooperative Fellowship award, an NSF Career Award MIP-9502294, NSF Grant CCR-9704512, an Ameritech Faculty Fellowship award, and grants from the Ohio Board of Regents. Preliminary versions of this work has been presented at CANPC '00 [6] and has been accepted for presentation at IPDPS '00 [8].

1 Introduction

Distributed and high performance applications require a low latency, high bandwidth communication facility for exchanging data and synchronization operations. 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.

The research and industry communities have proposed a group of communication systems [13] such as AM [23], VMMC [9], FM [15], U-Net [22, 24], LAPI [17, 7], and BIP [16] to address these issues. 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 features available in commercial systems [4]. The VIA specification has been influenced mostly by the U-Net and VMMC. Since the introduction of VIA, few software and hardware implementations of VIA have become available. The Berkeley VIA [12, 11], Giganet VIA [18], Servernet VIA [18], and MVIA [2] are among these implementations. Various components of VIA have been implemented in different ways in these implementations. Although, the performance of these implementations have been evaluated, there has not been a detail study of the design choices for implementing different components of VIA.

In this paper, we first discuss the essential components of VIA and present different approaches for implementing these components. We discuss the pros and cons 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 evaluate these approaches for *IBM Netfinity clusters* which are SP Switch–connected NT clusters recently announced by IBM. Then, we present the results of an exercise in implementing VIA on the these clusters. The IBM Netfinity cluster nodes are based on Intel x86 architecture and run the NT 4.0 operating system. Implementing VIA on this platform in an efficient manner raises many challenges: 1)performing fast and efficient virtual-to-physical address translations, 2) eliminating the double indirection of VIA, and 3) fast implementation of VIA doorbells on the NIC without polling and in the absence of any hardware support. In this paper we address all of these issues.

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 a host processor. 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. 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 and results in 100% TLB hit rate for send/receive operations. PDs are written to the NIC by the host instead of DMA to eliminate the NIC overhead. Therefore, our design allows the efficient implementation of the so called *double indirection* of VIA.

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. 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, it 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 [2, 11, 12, 18]. 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 than 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 8. 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 paper is organized as follows: In Section 2, we briefly overview the Virtual Interface Architecture and discuss the VIA send and receive operations in detail and identify different important components involved in these operations. Different design alternatives for implementing various components of VIA are discussed in Section 3. We discuss the characteristics of the SP switches and Network Interface Cards in Section 4. The performance evaluation of design alternatives for VIA components on SP-connected NT clusters are presented in Section 5. The design and implementation issues of the VIA implementation for SP-connected NT Clusters are discussed in Section 6. In Section 7, 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 8. In Section 9, we present our conclusions.

2 Virtual Interface Architecture (VIA)

In this section, we first present an overview of the Virtual Interface Architecture. Then, we discuss the steps taken in sending and receiving messages in VIA and present the basic components of VIA in that regard.

2.1 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[4]. 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 the 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 paper, we focus on the send/receive messaging facilities of VIA.

2.2 Message Passing in VIA

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

2.2.1 Send Operation

For sending a message, the following major steps are taken:

Constructing the send descriptor: The application creates a send descriptor in a registered memory region. This descriptor includes the virtual address of the message (send buffer) and its length. The message buffer is also allocated from a registered memory region. The send descriptor also contains a status field which VIA provider updates upon completion of the send operation.

Posting the descriptor: The application posts the descriptor using the `VipPostSend` function call. Through the doorbell mechanism, the NIC is informed about the existence of the send descriptor.

Obtaining the descriptor by the NIC: As soon as the NIC detects the existence of a new send descriptor, it retrieves from the descriptor the information required for sending the message. This information includes the address and the length of the send buffer and the address of the status field of the descriptor.

Transferring the message to the NIC: The NIC starts DMA operation(s) for transferring the data to a staging buffer in the NIC.

Injecting the message to the network: The NIC sends out the message from staging buffer to the destination node using one or more network packets. NIC also adds a VIA control header to each packet so that at the receiving node the VI id of the message can be determined.

Marking the send as complete: After sending the message out to the network, the NIC marks the status field of the VIA send 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 send: The application can check the status of the send operation using the `VipRecvDone` in a non-blocking fashion, the `VipRecvWait` in a blocking fashion, and `VipCQDone` and `VipCQWait` if a CQ is associated with the corresponding VI.

2.2.2 Receive Operation

For receiving a message, the following major steps are taken:

Constructing the receive descriptor: The application creates a receive descriptor in a registered memory region. The virtual address of the receive buffer (where the message should be copied to) is specified in the descriptor. The receive buffer is also allocated from a registered memory region. The descriptor contains the maximum length of a message that can be received. The receive descriptor also contains a status field which VIA provider updates upon completion of the receive operation.

Posting the descriptor: The application posts the descriptor using the `VipPostRecv` function call. Through the doorbell mechanism, the NIC is informed about the existence of the receive descriptor.

Obtaining the descriptor by the NIC: The NIC retrieves from the descriptor the information required for receiving a message into the receive buffer. The information includes the address and the length of the receive buffer and the address of the status field of the descriptor.

Receiving the message at the NIC: The NIC stores the incoming messages into staging buffers on the NIC. The message header is examined for finding the VI id to which the message has been sent.

Transferring the message to the receive buffer: The NIC initiates a DMA operation to transfer the data from the staging buffer in NIC to the receive buffer in host memory. The receive buffer address is obtained from the head descriptor of the VI work queue of receive descriptors posted earlier.

Marking the receive as complete: After the message is completely transfered to the receive buffer, the NIC marks the VIA receive 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. The length field of the descriptor is also updated according to the length of the received message.

Application detecting the completion of the receive: Similar to the send operation, the application can check the status of the receive operation using the `VipRecvDone` in a non-blocking fashion, the `VipRecvWait` in a blocking fashion, and `VipCQDone` and `VipCQWait` if a CQ is associated with the corresponding VI.

2.3 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.

3 Design Alternatives

In this section, we discuss the design alternatives for implementing different components of VIA.

3.1 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, if this approach is used.

3.2 Caching Descriptors

As discussed in Section 2.2, 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.

3.3 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 paper, 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.

3.4 Completion Queues

As mentioned in Section 2, 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 [4]. 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 VIs 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.

4 Overview of IBM SP Switch and Network Interface Card

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 cards.

4.1 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 [19]. 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. 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 buffered in the central queue 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 an 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/).

4.2 SP Network Interface Card

In Netfinity SP systems each host node attaches to the SP Switch by a PCI based network interface card (NIC) illustrated in Fig. 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[19, 20] and it is the PCI bus version of the NIC used in the RS/6000 SP systems.

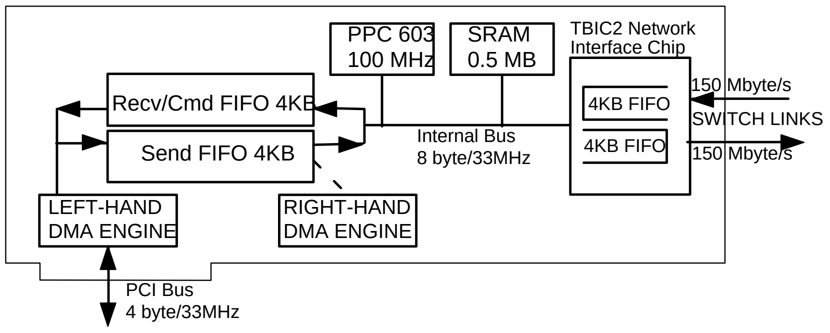


Figure 1: The Network Interface Card architecture in the Netfinity SP system.

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 PPC603 microprocessor's 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. 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.

A receive DMA operation from the SP switch includes the following steps: 1) An SP switch packet arrives at the TBIC2 buffer. Firmware decodes the packet header and decides for a destination address for the payload in the host memory. 2) Firmware inserts a 64-bit command word into the Recv/CMD-FIFO (*Start_LHS_Recv* command). The command word contains instructions for the LHS DMA engine such as a host memory physical address and the length of DMA operation. Upon receiving the command word, the LHS engine starts waiting for data to arrive at the Recv/CMD-FIFO. 3) Firmware starts the RHS engine which transfers the payload from TBIC2 to Recv/CMD-FIFO (*Start_RHS_Recv* command). 5) As soon as the first 8 bytes of the payload arrives at the FIFO, the LHS engine starts moving it to the host memory.

A send DMA operation from the host memory to the SP switch includes the following steps: 1) Firmware inserts a 64-bit command word to the Recv/CMD-FIFO (*Start_LHS_Send* command). The command word contains instructions for the LHS engine such as a host memory physical address and the length of DMA operation. 2) the LHS engine transfers payload from the desired host memory location to the Send-FIFO. 3) When the LHS DMA completes, the LHS engine increments a hardware counter. Firmware watches the counter to detect the completion of the LHS DMA for this packet. 4) Firmware inserts a switch packet header into TBIC2. Firmware then starts the RHS engine which transfers the switch packet payload from the Send-FIFO to TBIC2 (*Start_RHS_Send*).

The 4 KB FIFO buffers are used in a pipelined fashion to keep both FIFOs and both ends of the FIFOs busy. Switch packets are typically 1 KB. Firmware issues multiple (up to four) *Start_LHS_Send* commands each corresponding to a 1 KB packet to be sent. As soon as the first packet arrives, firmware starts the RHS engine (*Start_RHS_Send*) to send out the first switch packet. When sufficient space becomes available in the Send-FIFO, firmware issues more *Start_LHS_Send* commands to keep the host to Send-FIFO path busy, while streaming packets out from other end of the FIFO into the network. Receive operations are interleaved with send operations moving data through the FIFOs in a full duplex fashion.

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. The LHS engine is capable of moving data to/from PCI bus at this peak rate although this transfer rate is largely determined by the PCI chipset and system bus of the host system. On an Intel 440BX chipset-based 450 MHz PIII system, we have observed peak DMA bandwidth of 132 MBytes/s from host to NIC, and 115 MBytes/s from NIC to host. On a 450GX chipset based 200 MHz PPro system, peak DMA bandwidth was much less; we have observed peak bandwidth of 80 MBytes/s from host to NIC and 40 MBytes/s from NIC to host. These results are presented in detail in Section 7.

5 Performance Evaluation of Design Alternatives for VIA Components on IBM Netfinity System

In order to evaluate different design alternatives (as discussed in Section 3 of this paper), we implemented a subset of VIA on an IBM Netfinity SP switch-connected Cluster. 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. In the rest of this section, we first present the cost of the basic operations in this system. Then, we evaluate and compare different alternatives for implementing different components of VIA.

5.1 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 our NT testbed we used the Fast IO Dispatch method [21]. These measurements are presented in Table 1. It can be seen that the cost of PIO read operations is higher than that of PIO write operations. It can be also observed that the cost of switching from user space to kernel space is comparable to that of DMA startup.

5.2 Address Translation

Three approaches for performing the address translation were discussed in Section 3.1. 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

Table 1: Cost of basic operations in the IBM SP-connected NT testbed.

Operation	**Cost** (SP-NT)
Host PIO Write	0.33 μs/word
Host PIO Read	0.87 μs/word
User-space to Kernel-space	2.27 μs (Fast IO Dispatch)
DMA Startup (host to NIC)	1.78 μs
DMA Startup (NIC to host)	1.61 μs
NT Interrupt Latency	10-17 μs

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. The overall cost of the translation is this fixed cost plus the time required for accessing the TLB for each page frame of the send or receive buffer.

Table 2: Cost of different methods for implementing the virtual-to-physical address translation. (See Figures 2 through 5 for the value of Miss Rate for different benchmarks.)

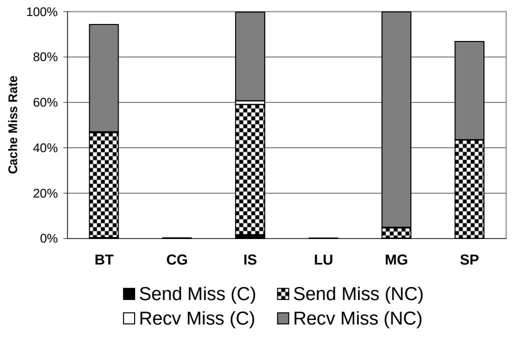
Address Translation Method	**Location/ Translator**	**NIC Memory Requirement**	**SP-NT Avg. Fixed Cost**
AT1	host/host	None	2.27
AT2	NIC/NIC	Proportional	0
AT3	host/NIC	Constant	$1.78 \times \mathit{Miss\ Rate}$

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 shows the fixed cost for performing the translation by using this approach. It can be seen that this fixed 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 this approach a more realistic and scalable approach for implementing the address translation.

In the third approach (AT3), the NIC performs 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).

In order to evaluate the effectiveness of caching and estimating the required cache size, and in the absence of the existence of a wide variety of applications/benchmarks for VIA, we used the NAS Parallel Benchmarks (NPB) [3, 5] 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 benchmarks 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 times of these programs significantly.)

Figure 2 shows the cache miss rates for the NAS benchmarks (Class A) 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. 3), the cache miss rates for all benchmarks other than IS become negligible. Increase in the number of processes results in a decrease in message sizes and this compensates 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.



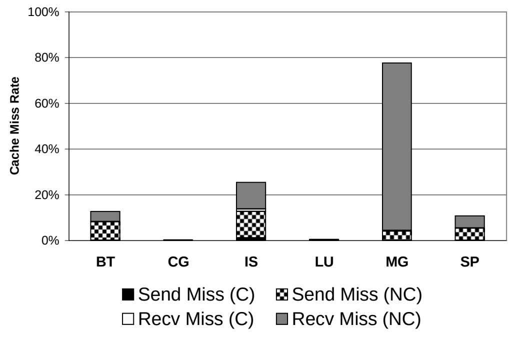
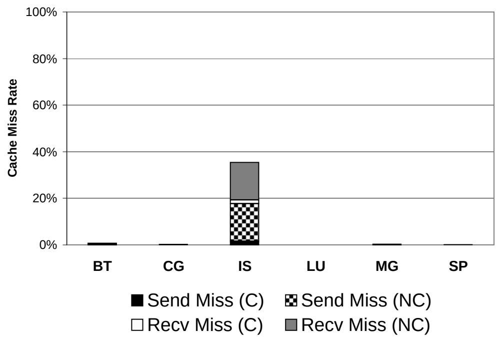


Figure 2: The cache miss rates for the NAS benchmarks (class A) using four processes (left) and 16 processes (right) with 128-entry direct-mapped caches. C and NC denote compulsory and non-compulsory misses, respectively.



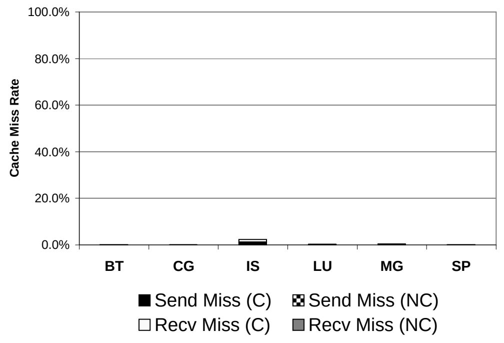


Figure 3: The cache miss rates for the NAS benchmarks (class A) using four processes (left) and 16 processes (right) with 1024-entry direct-mapped caches and 128-entry 8-way associative caches. (The miss rates are identical for both of these cache types.) C and NC denote compulsory and non-compulsory misses, respectively.

Figure 4 shows the cache miss rates for the NAS benchmarks (class B) 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 5. 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.

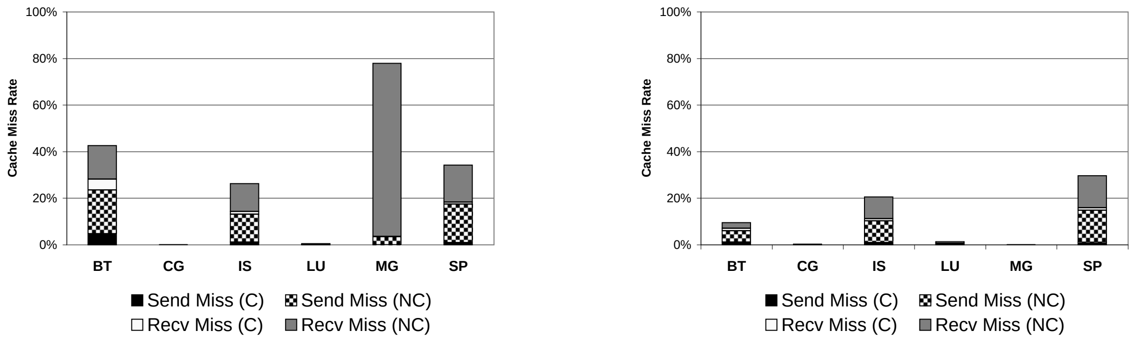


Figure 4: The cache miss rates for the NAS benchmarks (class B) using 16 processes (left) and 64 processes (right) with 128-entry direct-mapped caches. C and NC denote compulsory and non-compulsory misses, respectively.

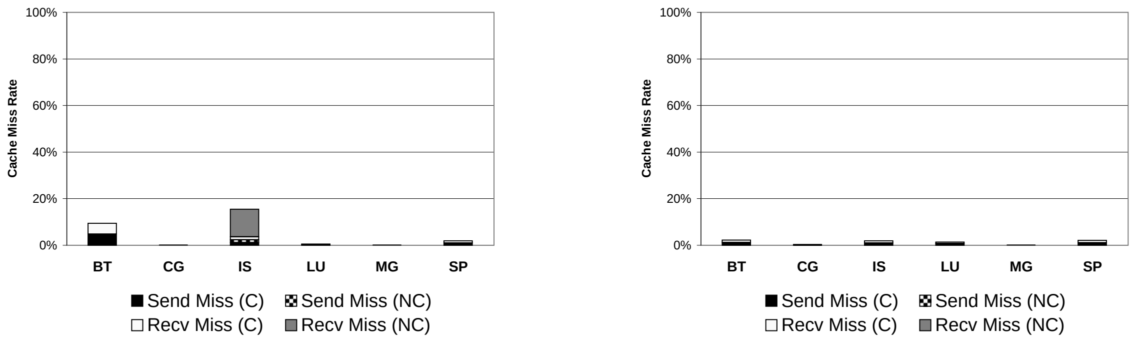


Figure 5: The cache miss rates for the NAS benchmarks (class B) using 16 processes (left) and 64 processes (right) with 1024-entry direct-mapped caches and 128-entry 8-way associative caches. (The miss rates are identical for both of these cache types.) C and NC denote compulsory and non-compulsory misses, respectively.

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. 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. (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.

5.3 Caching Descriptors

As discussed in Section 3.2, 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 Netfinity cluster testbed (Table 1), it can be seen that transferring up to five words through PIO is less time consuming than using the DMA. It should be noted that in our testbed, PCI write combining was not supported. 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), 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.

5.4 Doorbells

Two approaches for implementing doorbells (without hardware support) were discussed in Section 3.3. In the first approach (D1), in which a portion of NIC memory is associated with each doorbell and the user programs can directly ring a doorbell without the kernel intervention, the cost of ringing a doorbell is just the cost of writing a word into the NIC memory through PIO ($0.33 \mu sec$). In the second approach (D2), which requires the intervention of the kernel, the cost of ringing the doorbell is equal to that of the D1 approach plus the cost of switching from user to kernel space which is $2.27 \mu sec$ in our Netfinity cluster. Obviously, the cost of D1 is lower than that of D2. However, as mentioned in Section 3.3, if the D1 approach is used, the NIC is required to check the doorbells associated with all active send queues. Therefore, the time required for detecting the presence of a send descriptor (a rang doorbell) increases with the number of active VIs.

The increase in the cost of polling doorbells was measured to be around 0.6μsec per VI.

5.5 Completion Queues

We presented two approaches for implementing the completion queues in Section 3.4. 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 `MPI_Waitany` function. Table 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.

Table 3: Comparison between different approaches for implementing CQs.

Benchmark	**Number of Processes**	**Average Number of Receive Queues / CQ**	**Average CQ2 Cost**	**SP-NT CQ1 Cost**
Any Program	n	k	$k \times 0.08$	1.61
LU	4	2	0.16	1.61
LU	16	3	0.24	1.61
LU	64	3.5	0.28	1.61
MG	4	3	0.24	1.61
MG	16	4.6	0.37	1.61
MG	64	6.5	0.52	1.61

6 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. We also describe in detail the sequence of events taking place at the host and the NIC when sending and receiving VIA messages using the FirmVIA implementation.

6.1 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 6.2.2.

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 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.

6.2 Design Alternatives and Practical Choices for Implementation

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

6.2.1 Virtual-to–Physical Address Translation

As mentioned in Section 6.1, one of the major constraints that we were faced with while implementing VIA was the limited amount of available NIC memory. As discussed in Section 3.1, the required NIC memory for implementing the AT1 address translation method is less than that of other address translation methods. In this 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 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 [21]. As mentioned in Section 5.1, 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 result 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 in Section 6.2.3), which more than compensates the extra $2.27\mu sec$ required for switching to the kernel space. Thus, we followed the AT1 approach for performing the address translation.

To implement this 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.)

6.2.2 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 mentioned in Section 5.3, there is a high startup cost associated with DMA operations and PIO is faster than DMA for transferring up to five words from the host memory to the NIC memory. 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. 6. This area is called Receive Queue Cache (RQC). (We will discuss caching the send descriptors in Section 6.2.3.) 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.

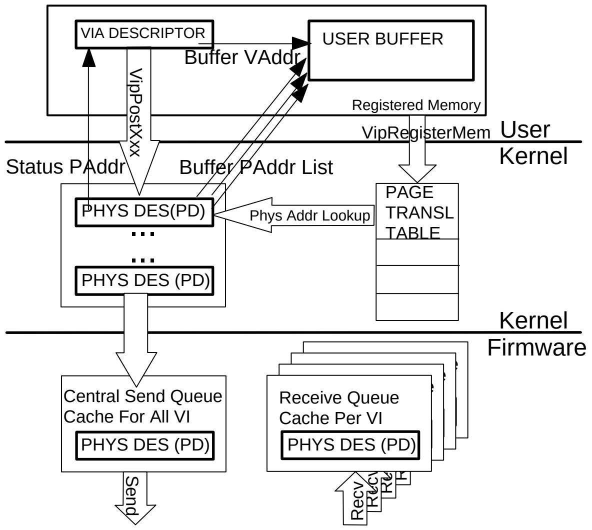


Figure 6: Sending and receiving messages using Physical Descriptors and address translation.

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. 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 to write as many 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.

6.2.3 Centralized Doorbell and Send Queue

Our NIC does not have hardware support for doorbells as stated before. Therefore we emulate the doorbells in the firmware. In the D1 approach for emulating doorbells, space is allocated for each doorbell in the NIC memory and this doorbell memory is mapped 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. The major shortcoming of this approach is the cost of polling doorbells in the NIC. Polling will add to the message latency with increasing number of processes and active VIs [14]. Therefore, we decided to follow the D2 approach in which doorbells and send queues are combined in a central place on the NIC.

Considering the fact that we go through the kernel for address translation as discussed in Section 6.2.1, 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.

6.2.4 Completion Queues

As discussed in Section 3.4, there are two major approaches for implementing completion queues. We chose the CQ2 approach for implementation in FirmVIA mostly because of its ease of implementation. Another factor which favors the implementation of CQ2 is that this approach requires no additional support from the NIC and the firmware. Keeping the complexity of the firmware has been one of the goals of the FirmVIA implementation.

6.2.5 Immediate Data

We 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 the 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. The results of this experiment are presented in Section 7.1.2.

6.2.6 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. 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 6.2.2, 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.

6.3 Events Sequences in Sending and Receiving Messages with FirmVIA

In order to gain a complete understanding of the FirmVIA implementation, we present the sequence of events that occur during the send and receive operations in this section.

When a memory region is registered (through the *VipRegisterMem* function call) the device driver pins the memory and a list of starting physical addresses of the pages is created in the kernel memory. For sending a message, the application creates a send descriptor in a registered memory region and posts the descriptor using the *VipPostSend* function call. Then, the host creates a PDCS and writes it into the CSQC on the NIC. The host converts the DS of the VIA descriptor into one or more PDDS which are also written into the CSQC (Fig. 6). Finally, the host advances the tail variable of the CSQC. As soon as the firmware on the NIC detects the existence of a new send descriptor by finding a new value for tail, it starts the send processing. The firmware first starts the DMA operation(s) for transferring the data to the NIC Send-FIFO (Fig. 1) and then sends out the message to the destination node using one or more switch packets. Firmware adds a VIA control header to each switch packet payload so that at the receiving node the VI id of the message can be determined. The size of this header is eight bytes. The firmware also marks the VIA send descriptor in the host memory as completed through a DMA operation. The user can find the status of the send operation by using the *VipSendDone* and *VipSendWait* function calls in a non-blocking or blocking manner.

For receiving a message, the application creates a receive descriptor in a registered memory region and posts the descriptor using the *VipPostRecv* function call. Then, the host creates a PDCS and writes it into the corresponding RQC on the NIC. The host converts the DS of the VIA descriptor into one or more PDDS which are also written into the RQC (Fig. 6). Finally, the host advances the tail variable of the RQC. When a message arrives at the destination NIC, firmware reads the VIA control header of the message to determine the VI id. Then, the firmware retrieves the first receive descriptor in the RQC of that VI to determine physical address(es) of the user buffer, and initiates the required DMA operations to move the data to the host memory. Finally, the firmware updates the status and length fields of the VIA receive descriptor of user application in the host memory by a DMA operation. The user can check the status of the receive operation by using the *VipRecvDone* in a non-blocking fashion. If the application thread is blocked on the completion of the receive (by using the *VipRecvWait* function), the host will be interrupted and the blocked thread will be released. In cases where a completion queue is associated with a VI, the NIC also enters an item containing the VI id and the descriptor handle into the completion queue through a DMA operation.

7 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 the FirmVIA. The results presented in this section were obtained on a cluster whose characteristics were presented in Section 5 and Table 1. 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.

7.1 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 7.2.2. 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.

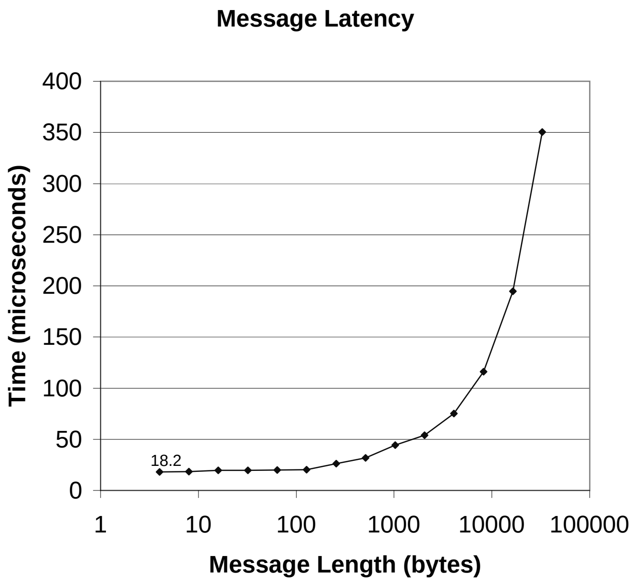


Figure 7: Message latency for different message sizes.

The latencies for different message sizes are shown in Figure 7. The one-way latency for four-byte messages is $18.2\mu s$. Figure 8 shows the latency for small messages in detail. We observe that the latencies for messages up to 128 bytes are at about $20\mu s$. There is a noticeable increase in the latency of 256 byte message. This can be attributed to the increased delay in the host to NIC Send-FIFO DMA transfer which changes the sequence of the firmware operations: when the firmware initiates DMA from host to Send-FIFO, it checks for the DMA completion soon after the initiation. For smaller than 256 byte messages the DMA completes fast enough, thus the firmware can send out the message from Send-FIFO to the network in its first try. For 256 byte messages and bigger the DMA operation doesn't complete on time for the firmware to send out the message in the first try. In this case the firmware leaves the send loop to check for any receive processing to be performed, and it may check if there are any housekeeping activities to be performed. The firmware eventually returns back to the send loop however with an increased latency at 256 bytes or longer.

From Fig. 8, it can also be observed that the slope of the latency curve is small and remains constant after the message size of 1024 bytes which is the maximum switch packet payload size. For messages longer than 1024 bytes the firmware sends the VIA messages in multiple of 1 KB switch packets. Note that some portions of the processing and transmitting the packets are pipelined.

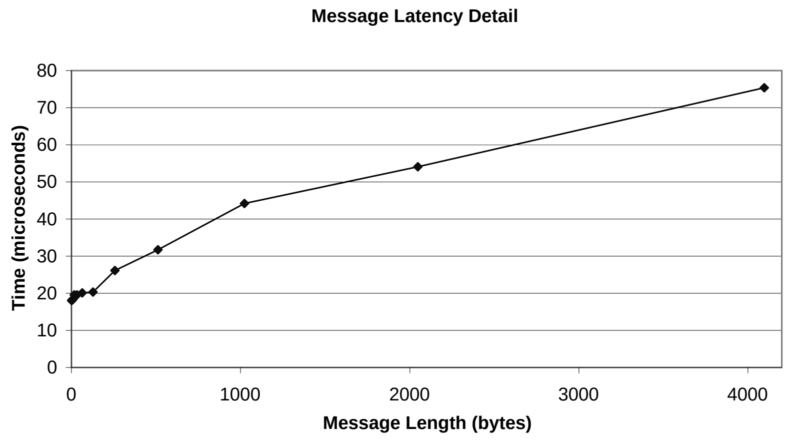


Figure 8: Latency for messages of up to 4KB.

7.1.1 Components of Latency

To find out where and how the measured time is spent, we instrumented the firmware and the device driver to measure the time spent in different phases of data transfers. Each phase 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 in Figs. 7–8. However, such a study provides insight to our implementation. Figure 9 illustrates the time spent in 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 4. 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). The time spent in kernel space includes the time required for accessing and updating the head and tail of the corresponding queue in the NIC.

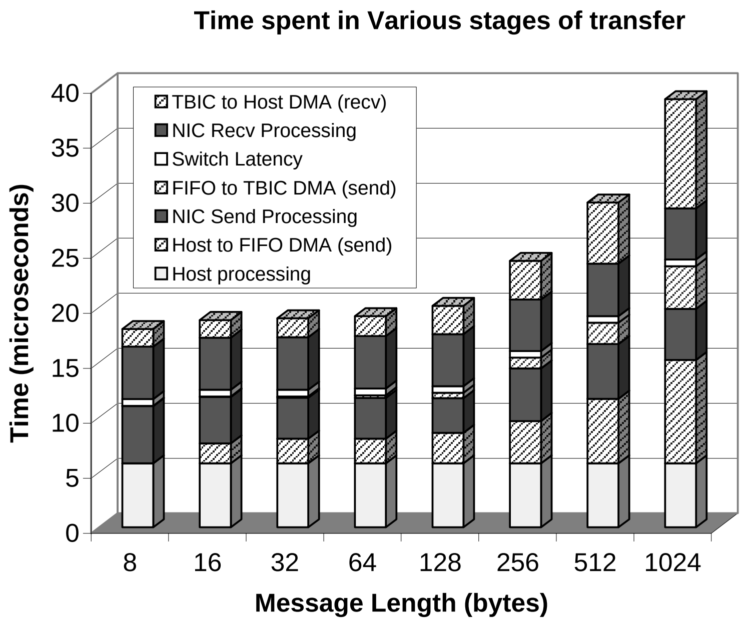


Figure 9: The breakdown of short message latencies.

The host memory to Send-FIFO transfer time is shown as the second bar from the bottom in Fig. 9. The cost of the NIC firmware processing a physical descriptor (PD) is shown as the third bar from the bottom.

Table 4: Breakdown of the Host Overhead

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

It can be observed that this cost remains almost constant for messages of 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 7.1. It is to be noted that for messages of eight bytes or less, the data is transferred to the NIC through PIO instead of using DMA, as described in Section 6.2.5. We discuss the performance tradeoff between PIO and DMA in more detail in Section 7.1.2.

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 topmost 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. 9. 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. 9 for sends. The NIC send and receive processing costs also contain the time for marking the VIA descriptors in host memory as complete.

7.1.2 PIO vs. DMA

As discussed in Section 6.2.5, 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 10 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. 10). The extra space required in the CSQC was another constraint. Thus, we chose to use PIO for messages of eight bytes and less only.

7.2 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 Figure 11. The maximum observed bandwidth is 101.4 MB/s. Note that the half-bandwidth is achieved for the message size of 864 bytes.

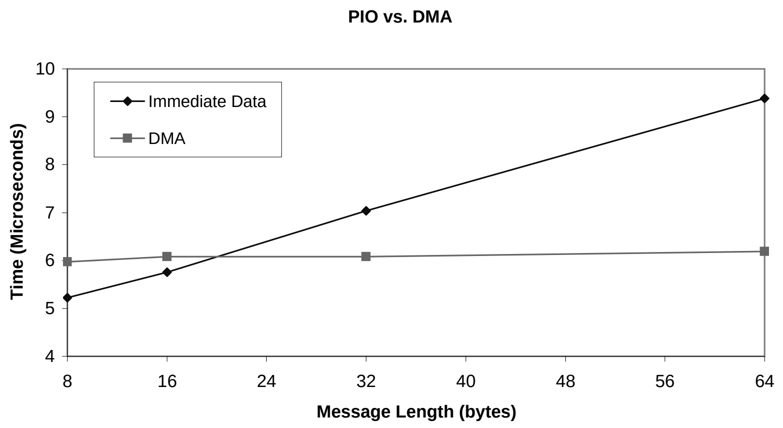


Figure 10: Delays for sending data with PIO vs. DMA. Descriptor processing delay is included.

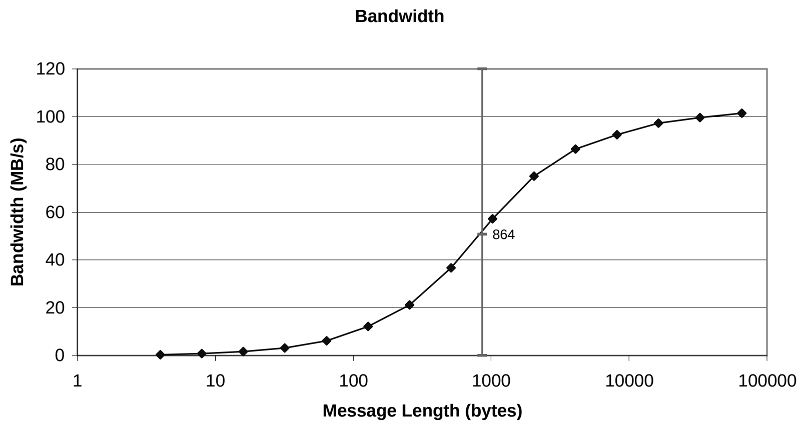


Figure 11: Measured bandwidth for different message sizes. The half-bandwidth is achieved for 864-byte messages.

7.2.1 The Bottleneck of Bandwidth

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. 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 12 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 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.

7.2.2 Effect of Packet Size

As mentioned in Section 4.2, 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 13 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–

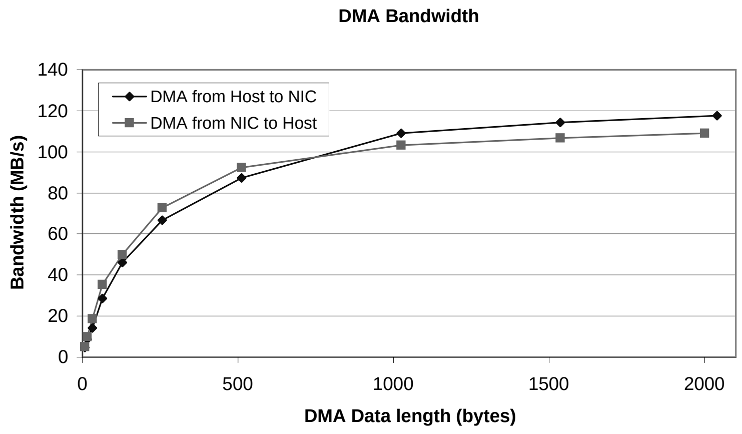


Figure 12: The raw PCI DMA bandwidth.

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 which 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 13 illustrates another effect where increasing the size of the user payload from 1000 bytes to 1024 increases 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 (see Start_LHS_Send and Start_RHS_Send command pairs in Section 4.2.) 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 as shown in Figure 13.

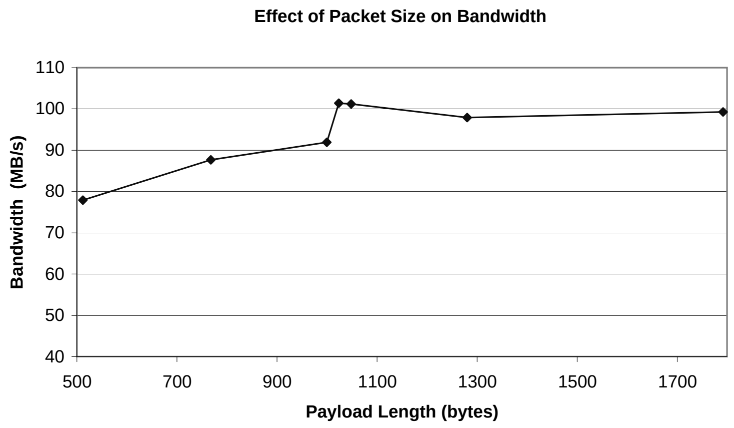


Figure 13: Effect of packet size on the bandwidth.

8 Related Work

Performance results of several VIA implementations are summarized in Table 5. The Berkeley VIA (Version 1) [12] 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. Not caching the descriptors have a graver 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 [10] 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) [11] 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. 400 MHz PCs 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 [11]). 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 avoid in our FirmVIA implementation 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.* [18] 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 [18]). 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 [18].

The Virtual Interface Benchmark (VIBe) [14] has been recently developed for evaluating the performance of VIA implementations under different communication scenarios and with respect to the implementation of different components of VIA.

Table 5: Latency and Bandwidth Results of Different Communication Systems on Modern Networks

Communication System	Latency (μs)	Bandwidth (MBytes/s)	Host	Network	OS
Berkeley VIA [12]	23	29.3	USPARC 167	Myrinet (SBus)	Solaris 2.6
Berkeley VIA [12]	26	53.1	Pentium 300	Myrinet (PCI)	NT 4.0
Berk. VIAv1 [11]	≈ 24	≈ 64	Dual PII 400	Myrinet (PCI)	NT 4.0
Berk. VIAv2 [11]	≈ 32	≈ 64	Dual PII 400	Myrinet (PCI)	NT 4.0
Giganet VIA [18]	≈ 10	≈ 70	Xeon 450	cLAN	NT 4.0
Servernet VIA [18]	≈ 100	22	Xeon 450	ServerNet	NT 4.0
LAPI on SP [17]	34	97	P2SC 120	SP (MCA)	AIX
MVIA [2]	19	60	SMP PII 400	GBit Ether	Linux 2.1
MVIA [2]	23	11.9	SMP PII 400	100MB Ether	Linux 2.1
FirmVIA (this paper)	18.2	101.4	PIII 450	SP (PCI)	NT 4.0

9 Conclusions

In this paper, we studied different components of VIA for sending and receiving messages. We also presented different approaches for implementing various components of VIA such as virtual-to-physical address translation, caching descriptors, doorbells, and completion queues. We also discussed pros and cons of each approach. We evaluated the cost and resource requirements of these approaches on our IBM Netfinity NT cluster testbed. Then, we presented an experimental VIA implementation on the IBM Netfinity NT Cluster based on these evaluations and the restrictions and requirements of our system. 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.

Disclaimer

The VIA implementation presented in this paper is not a part of any IBM product and no assumptions should be made regarding its availability as a product in the future.

References

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

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

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

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

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

[6] 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.

[7] 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.

[8] M. Banikazemi, V. Moorthy, L. Herger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP Switch-Connected NT Clusters. Accepted for presentation at International Parallel and Distributed Processing Symposium, May 2000.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] S. N. Kutlug, M. Banikazemi, D. K. Panda, and P. Sadayappan. VIBe: A Micro-benchmark Suite for Evaluating Virtual Interface Architecture (VIA) Implementations. Submitted to International Conference on Supercomputng (ICS'2000).

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

[16] 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/.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

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