# Efficient Virtual Interface Architecture Support for the IBM SP Switch-Connected NT Clusters

Mohammad Banikazemi†[1,2] Vijay Moorthy†[2] Lorraine Herger‡
Dhabaleswar K Panda† Bulent Abali‡[3]

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

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

**Abstract**

*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, Distributed Memory, Interconnection Networks, Performance Evaluation, Scalable Architecture, Low-Latency Communication.

[1] This author's work is supported in part by an IBM Cooperative Fellowship award.
[2] Work presented in this paper was performed while visiting the IBM T. J. Watson Research Center.
[3] To whom all correspondence should be addressed.

# 1 Introduction

Raw bandwidth of networks increased significantly in the past few years. Networking hardware supporting bandwidths in the order of gigabits per second have become widely available. However, bandwidths experienced by the user applications remained substantially lower and the application to application communication latency has remained high. The layered nature of the legacy networking software is one of the main reasons for this performance gap between the physical layer and the application layer. Expensive system calls and memory–to–memory copies are some of the factors responsible for degradation of networking performance.

The research community has proposed a group of communication systems [9] such AM [18], VMMC [5], FM [10], U-Net [17, 19], LAPI [12], and BIP [11] to address these issues. All of these communication systems use much simpler communication protocols in comparison with legacy protocols such as the TCP/IP. In these new systems, most layers of traditional networking software has been eliminated. The role of the operating system has been much reduced. In some cases user application is given direct access to the network interface, commonly called the *user-level network interface*. The Virtual Interface Architecture (VIA) specification was developed to standardize these user-level network interfaces and to make their ideas available in commercial systems [3]. VIA was mostly influenced by the U-Net and VMMC. Since its introduction software and hardware implementations of VIA have become available. This paper presents the results of an exercise in implementing a subset of VIA on the IBM SP systems hardware using the NT 4.0 operating system. We call our implementation *FirmVIA,* which stands for VIA implemented in firmware.

The IBM SP is one of the most successful parallel systems commercially available today. The IBM SP system consist of RS/6000 nodes running AIX and the IBM SP Switch interconnecting them. The RS/6000 SP network interface cards (NIC) support the TCP/IP protocol suite and a proprietary user–space protocol. The MPI and LAPI [12, 4] communication libraries are layered over the proprietary user–space protocol. This year IBM announced the *Netfinity SP System* which is an SP Switch–connected NT cluster. The Netfinity SP nodes are based on Intel x86 architecture and run the NT 4.0 operating system. Netfinity SP supports the TCP/IP protocol suite over the SP Switch. The implementation of a low–level, high–performance communication subsystem such as VIA for the Netfinity SP system seems to be the next logical step.

We studied approaches in implementing VIA efficiently on the Netfinity SP systems. There were many challenges in the implementation including: 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. Main contributions of this paper are:

- We explore the partition of the VIA functions among the user space, the kernel space, and the NIC firmware. A NIC processor is generally not as powerful as host processors. 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. Physical Descriptors are written to the NIC by the host instead of DMA to eliminate the NIC overhead. Therefore, in our design the so called *double indirection* of VIA is implemented efficiently.

- 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, 7, 8, 13]. The half-bandwidth is reached for messages of 864 bytes. The one-way latency of four-byte messages is 18.2 $\mu$s which is better other VIA implementation's latencies except for the hardware implementation of VIA [1]. Performance results of FirmVIA and other VIA implementations' are summarized in Table 3 in Section 6. 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. We discuss the characteristics of the SP switches and Network Interface Cards in Section 3. The design and implementation issues of the VIA implementation for SP-connected NT Clusters are discussed in Section 4. In Section 5, 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 6. In Section 7, we present our conclusions.

# 2 VI Architecture

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[3]. 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. In addition to the work queues, each VI can be associated with a completion queue. A completion queue merges the completion status of multiple work queues. Therefore, an application need not poll multiple work queues.

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

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

# 3 IBM SP Switch

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

## 3.1 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 [14]. The basic elements of the SP Switch are the 8-port switch chips and the network interface cards (NIC) interconnected by communication links. Switch chips provide means for passing data arriving at an input port to an appropriate output port. In the current implementation of Netfinity SP systems, the switch chips and NIC ports have 150 MBytes/s data bandwidth in each direction, resulting in 300 MBytes/s bidirectional bandwidth per link and 1.2 GBytes/s aggregate bandwidth per switch chip.

The switch chip, called the TBS chip, contains eight input ports and eight output ports, a buffered crossbar, and a central queue. All switch chip ports are one flit (one byte) wide. When an incoming packet encounters no contention for the selected output port, it cuts through the crossbar in wormhole fashion [14]. Cut through latency is less than 300 nanoseconds. When an incoming packet is blocked due to unavailability of an output port, flits of the packet are sent to the central queue for temporary buffering until the output port becomes available. The central queue stores up to 4KB of incoming data and this storage space is dynamically allocated for 8 output ports according to the demand.

The TBS chip is used both in RS/6000 SP and Netfinity SP systems. The TBS chips can be interconnected by links to form larger networks. Basic building block of 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/).

## 3.2 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[14, 15] and it is the PCI bus version of the NIC used in the RS/6000 SP systems.

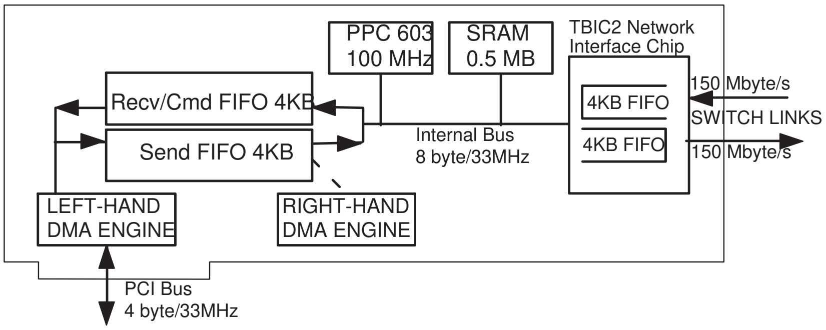


Figure 1: The Network Interface Card architecture in the Netfinity SP system.

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 603 address space. The SRAM is divided mainly into cached and non-cached regions. Cached regions contain the firmware executable and private data. Non-cached SRAM regions are used for communicating with the host processor. The host (an Intel x86 based PC with NT 4.0) typically maps the shared regions of the SRAM into its kernel or user address space. The host processor stores or loads 32-bit words (using x86 *mov* instruction) to/from SRAM to communicate with the firmware. Firmware can assert the PCI interrupt signal to notify asynchronous events to the host processor.

Two 4KB FIFO buffers are used as an intermediate storage between the PCI bus and TBIC2 (Fig. 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 5.

# 4 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.

## 4.1 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 4.2.2.

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.

## 4.2 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.

### 4.2.1 Virtual–Physical Address Translation

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

Address translation is simply done by indexing the table with the virtual page frame number.

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

Then, the next problem is how to pass the information in the TLB to the NIC. In one approach the NIC can do the TLB lookup by performing a DMA operation from the TLB in the host memory. In another approach, the host processor can do the TLB lookup on NIC's behalf and then pass the information to the NIC. In our case the first approach has a high startup cost of DMA, complicates the firmware and increases it's execution path. Queueing delays may also occur in send and receive FIFO buffers (Fig. 1) which will increase the communication latency. In the second approach the host processor needs to do the TLB lookup in kernel space, since user space applications cannot be trusted to provide valid physical addresses to 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 [16]. 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 4.2.3. Thus, we followed the second approach.

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

### 4.2.2 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 stated before, there is a high startup cost associated with DMA operations. More importantly, when receiving a message from a high speed network, there is little time for the NIC to fetch the desired descriptor from the host memory. If the NIC cannot fetch the descriptor fast enough it may need to stall the reception of the message. This can result in message packets backing up into the network which may eventually block the entire communication in the network.

Therefore, we decided to use an alternative approach and cache PDs on the NIC whenever they get posted. Fortunately, VIA descriptors exhibit high locality of reference for the VIA send and receive operations since they are consumed in sequential order and thus cache hit rate is essentially 100%. Each VI has its own caching area in the NIC memory for receive descriptors as shown in Fig. 2. This area is called Receive Queue Cache (RQC). (We will discuss caching the send descriptors in Section 4.2.3.) 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.

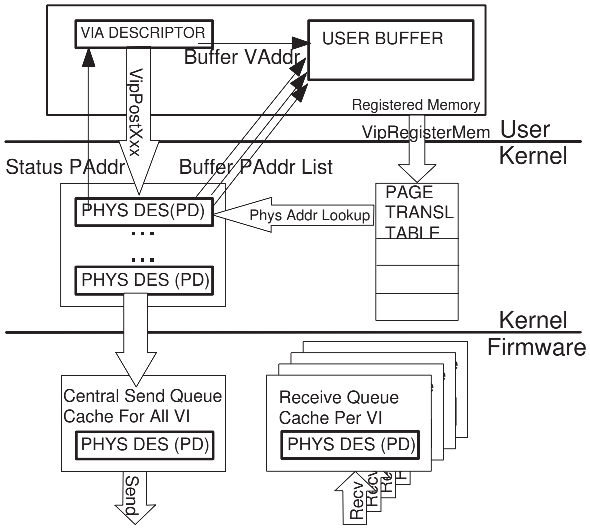


Figure 2: Sending and receiving messages using Physical Descriptors and address translation.

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. We are currently working on such a scheme.

### 4.2.3 Centralized Doorbell and Send Queue

VIA specifies that each VI is associated with a pair of doorbells. The purpose of a doorbell is to notify the NIC of existence of newly posted descriptors. Our NIC does not have hardware support for doorbells as stated before. Therefore we emulate the doorbells in the firmware. One approach for emulating doorbells is allocating space for each doorbell in the NIC memory and mapping this doorbell memory to the process' address space. The user application rings the doorbell by simply setting the corresponding bit in the NIC memory. To protect a doorbell from being tampered by other processes, doorbells of different processes need to be on separate memory pages in the NIC. An issue in emulating doorbells is the cost of polling them in the NIC. Polling will add to the message latency with increasing number of processes and active VIs. Therefore, we decided to combine doorbells and send queues in a central place on the NIC.

Considering the fact that we go through the kernel for address translation as discussed in Section 4.2.1, 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.

### 4.2.4 Immediate Data

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

Since performing 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.

#### 4.2.5 Remote Direct Memory Access (RDMA)

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

In our RDMA design, this address translation problem can be solved in two different ways: 1) In order to prevent stalling the reception of RDMA packets, the NIC can DMA all RDMA packets directly into a kernel buffer (whose physical address is known to the NIC). Then, these messages can be copied to the target user buffer by the host processor. 2) The NIC can do the TLB lookup from host memory by DMA upon message reception. The second method, as mentioned earlier in Section 4.2.2, 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.

### 4.3 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. 2). 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 finds 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. 2). 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.

# 5 Performance Evaluation

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

## 5.1 Experimental Setup

The results presented in this section were obtained on a cluster of PCs with 450 MHz Pentium III processor and 100 MHz system bus. Each node has 128 MB of SDRAM, PIII has 16 KB of L1 data cache, 16 KB of L2 instruction cache, and 512 KB of L2 cache. Each node has a 33 MHz/32-bit PCI bus and runs the NT 4.0 operating system. Table 1 shows the cost of elementary operations on this system. For all experiments, the maximum switch packet payload was set to 1032 bytes (1024 bytes of payload plus 8 bytes of VIA control header) unless otherwise stated.

Table 1: Cost of Basic Operations

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

## 5.2 Latency

We determined the message latency as one half of the measured roundtrip latency. The test application sends a message to a remote node's test application. The remote node replies back with a message of the same size. Upon receiving the reply the initiating node repeats the ping–pong test and repeats it large number of times so that the overhead of reading the timer is negligible. We aligned the send and receive buffers to the beginning of physical pages so that buffers crossing page boundaries do not influence latency measurements for small messages. Performance effects of crossing page boundaries are discussed in Section 5.3.2. 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.

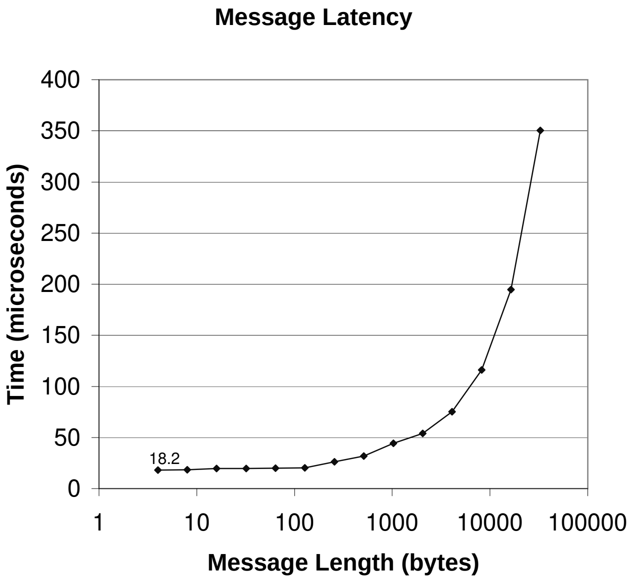


Figure 3: Message latency for different message sizes.

The latencies for different message sizes are shown in Figure 3. The one-way latency for four-byte messages is $18.2\mu s$. Figure 4 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. 4, 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 1 KB switch packets. Note that some portions of the processing and transmitting the packets are pipelined.

### 5.2.1 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 different components of latency. Each component was measured several times and the minimums were recorded. Due to this method of recording, the summation of the delays of different stages of transfer is slightly lower than the measured one-way latencies in Figs. 3–4. However, such a study provides insight to our implementation. Figure 5 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

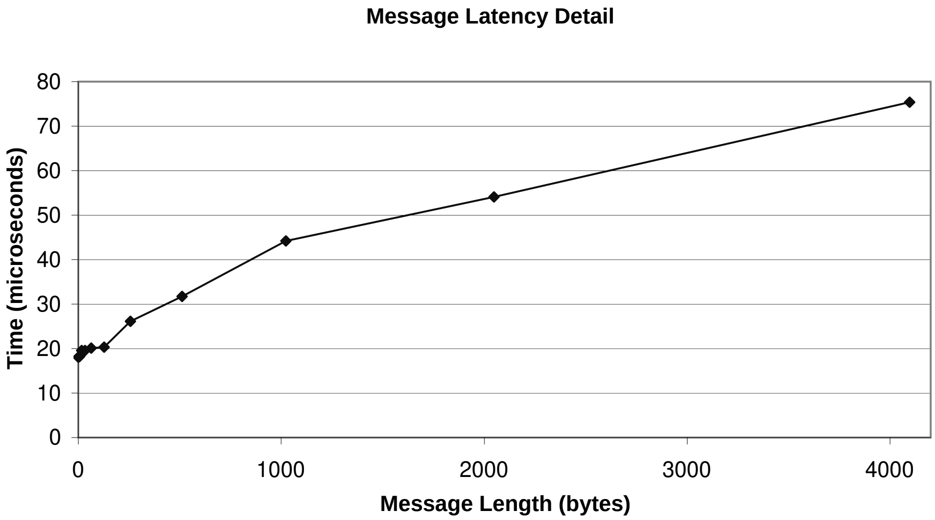


Figure 4: Latency for messages of up to 4KB.

host overhead is given in Table 2. Note that the PIO cost of writing a physical descriptor (PD) in to the NIC is the time for writing five words (three words for the PDCS and two words for the PDDS).

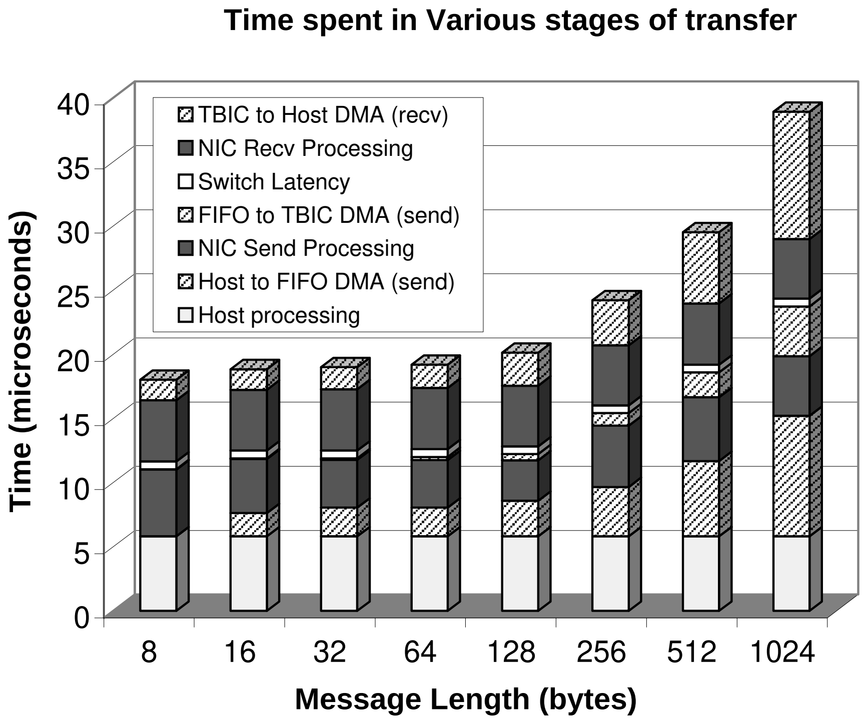


Figure 5: The breakdown of short message latencies.

The host memory to Send-FIFO transfer time is shown as the second bar from the bottom in Fig. 5. The cost of the NIC firmware processing a physical descriptor (PD) is shown as the

Table 2: Breakdown of the Host Overhead

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

third bar from the bottom. 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 5.2. It is to be noted that for messages of eight bytes or less, the data is transferred to the NIC through Programmed IO (PIO) instead of using DMA, as described in Section 4.2.4. We discuss the performance tradeoff between PIO and DMA in more detail in Section 5.2.2.

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 top most bars.

It is to be noted that on the receiving side the LHS DMA and RHS DMA engine receive operations are almost completely overlapped. While the RHS DMA engine is transferring message payload from the TBIC2 buffer to the Recv/CMD-FIFO, the LHS DMA engine is transferring that payload from the Recv/CMD-FIFO to the host memory. Since the PCI bus bandwidth is less than that of the NIC internal bus, the cost of data transmission from TBIC2 to the Recv/CMD-FIFO is masked and does not appear as a separate item in Fig. 5. 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. 5 for sends. The NIC send and receive processing costs also contain the time for marking the VIA descriptors in host memory as complete.

### 5.2.2 PIO vs. DMA

As discussed in Section 4.2.4, 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 6 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. 6). The extra space required in the CSQC was another constraint. Thus, we chose to use PIO for messages of eight bytes and less only.

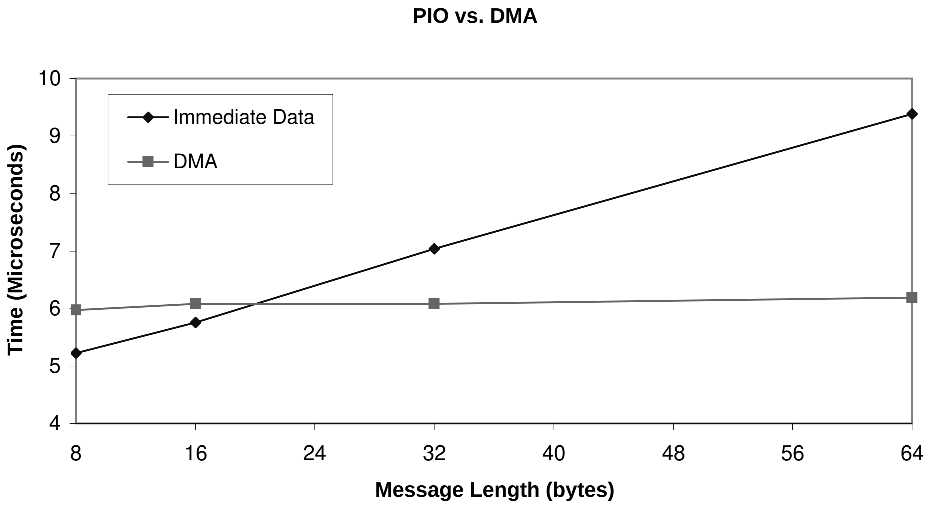


Figure 6: Delays for sending data with PIO vs. DMA. Descriptor processing delay is included.

## 5.3 Bandwidth

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

The peak measured bandwidth for different message sizes is shown in Figure 7. The maximum observed bandwidth is 101.4 MB/s. Note that the half-bandwidth is achieved for the message size of 864 bytes.

### 5.3.1 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 8 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.

### 5.3.2 Effect of Packet Size

As mentioned in Section 3.2, 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.

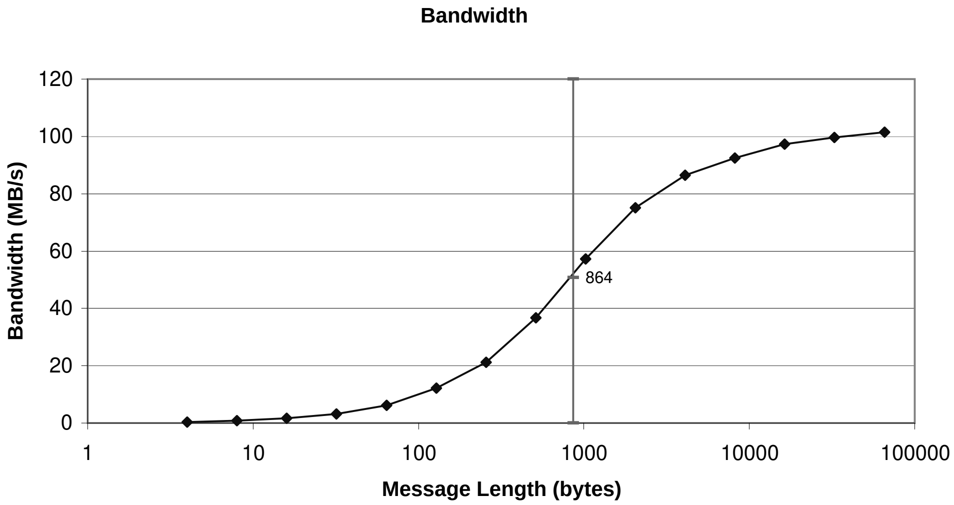


Figure 7: Measured bandwidth for different message sizes. The half-bandwidth is achieved for 864-byte messages.

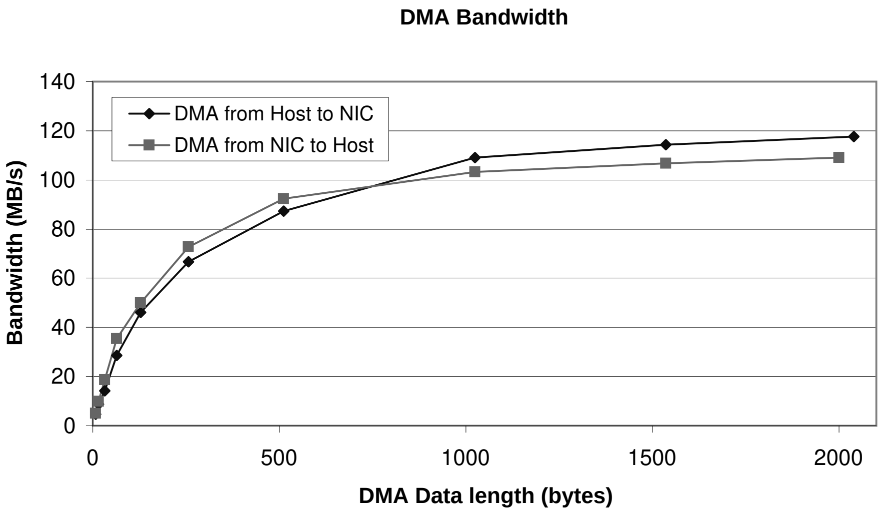


Figure 8: The raw PCI DMA bandwidth.

Figure 9 illustrates the effect of varying the packet size on the maximum possible bandwidth. It is seen that the maximum bandwidth is achieved for the switch packet size with a user payload of 1024 bytes. Increasing the size of user payload beyond 1KB does not increase the bandwidth. In fact, there is a slight decrease in the bandwidth for larger payloads. This can be attributed to the 4 KB size of the NIC Send–FIFO. When the maximum user payload in switch packets is set to 1KB the Send–FIFO can be filled with exactly 4 switch packets worth of data ($4 \times 1$ 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 9 illustrates another effect where increasing the size of the user payload from 1000 bytes to 1024 increase the bandwidth significantly. This has to do with our firmware implementation. To simplify the firmware we structured it so that each LHS DMA initiation on the NIC results in one switch packet sent out to the network (see Start_LHS_Send and Start_RHS_Send command pairs in Section 3.2.) 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 in Figure 9.

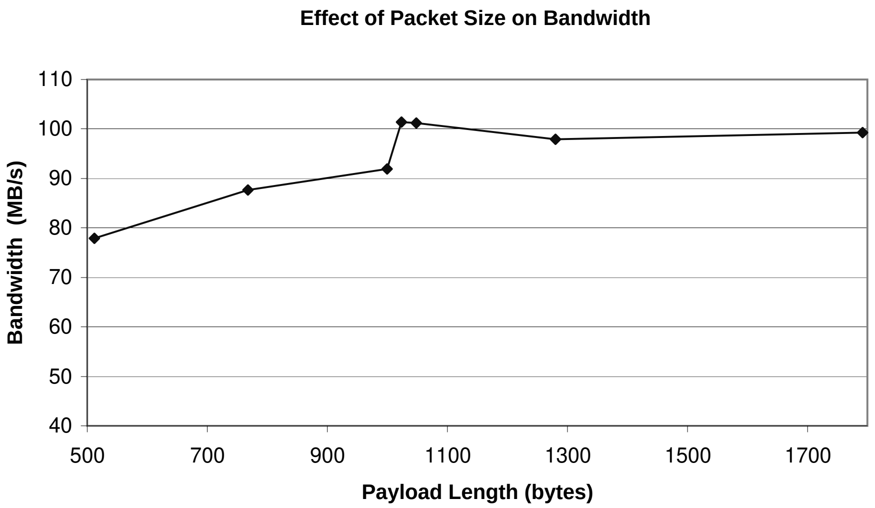


Figure 9: Effect of packet size on the bandwidth.

# 6 Related Work

Performance results of several VIA implementations are summarized in Table 6. The Berkeley VIA (Version 1) [8] 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 [6] with the LANai 4.x-based network interface card were used. The minimum reported latency for a PCI-based system is $26\mu$s. 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) [7] 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 $\mu$s. The latency for the case where TLB miss happens only at the first use of VI was shown to be as high as 34 $\mu$s (Fig. 7 of [7]). When the misses happen all the time, the latency can increase up to 40 $\mu$s. The complexity of the new firmware contributed to the increased latency which is what we tried to avoid by using Physical Descriptors. The maximum peak bandwidth was reported as 64 MBytes/s. The half-bandwidth was achieved by messages longer the 1000 bytes. Unlike our implementation, no caching of descriptors is being used in this study. The new address translation mechanism which is essentially added in response to the limited resources available on the NIC (the similar restriction that we faced in our system) increases the latency by more than $6\mu$s. In contrast, our implementation pays only the $2.27\mu$s 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.* [13] 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 [13]). 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 $\mu$s for the cLAN and around 100 $\mu$s 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 $\mu$s [13].

Table 3: Latency and Bandwidth Results of Different Communication Systems on Modern Networks

| Communication System | Latency ($\mu$s) | Bandwidth (MBytes/s) | Host | Network | OS |
|---|---|---|---|---|---|
| Berkeley VIA [8] | 23 | 29.3 | USPARC 167 | Myrinet (SBus) | Solaris 2.6 |
| Berkeley VIA [8] | 26 | 53.1 | Pentium 300 | Myrinet (PCI) | NT 4.0 |
| Berk. VIAv1 [7] | $\approx 24$ | $\approx 64$ | Dual PII 400 | Myrinet (PCI) | NT 4.0 |
| Berk. VIAv2 [7] | $\approx 32$ | $\approx 64$ | Dual PII 400 | Myrinet (PCI) | NT 4.0 |
| Giganet VIA [13] | $\approx 10$ | $\approx 70$ | Xeon 450 | cLAN | NT 4.0 |
| Servernet VIA [13] | – | 22 | Xeon 450 | ServerNet | NT 4.0 |
| LAPI on SP [12] | 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 |

# 7 Conclusions

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

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

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

[11] 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/.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

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