Comparison and Evaluation of Design Choices for Implementing the Virtual Interface Architecture (VIA)*

Mohammad Banikazemi† Bulent Abali‡ Dhabaleswar K Panda†

†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
Yorktown Hts, NY 10598
email:{abali}@watson.ibm.com

Contact Author: Mohammad Banikazemi

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. As a user application, we use the NAS Parallel Benchmarks to study the effect of caching the address translation tables on the NIC and to study completion queues. As a hardware platform we use the IBM Netfinity SP cluster running the NT 4.0 operating system and a Myrinet connected cluster of PCs running the Linux operating system.

Keywords: Cluster Computing, Virtual Interface Architecture, Performance Evaluation, Scalable Architecture, Low-Latency Communication.

*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.

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 community has proposed a group of communication systems [11] such as AM [19], VMMC [8], FM [12], U-Net [18, 20], LAPI [15, 6], and BIP [13] 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 ideas 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 [10, 9], Giganet VIA [16], Servernet VIA [16], MVIA [2], and FirmVIA [7] are among these implementations. Different 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 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 use the NAS Parallel Benchmarks to study the effect of caching the address translation tables on the NIC and to study different Completion Queue implementations. We use a subset of VIA implemented on an IBM SP-connected Netfinity cluster [7] running the MS Windows NT operating system and a Myrinet-connected cluster of PCs running the Linux operating system to evaluate different components of VIA.

The rest of this paper is organized as follows: In Section 2, we briefly overview the Virtual Interface Architecture. In Section 3, we discuss the VIA send and receive operations in detail and identify different important components involved in these operations. Different design alternatives for implementing different components of VIA are discussed in Section 4. The performance evaluation results are presented in Section 5. Related work is discussed in Section 6. In Section 7, we present our conclusions.

2 VIA 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 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.

3 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.

3.1 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.

3.2 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.

3.3 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.

4 Design Alternatives

The first component through which the NIC is informed of the existence of an outstanding send or receive descriptor is called a doorbell as discussed in the VIA specification. Doorbells can be implemented in software or hardware. The hardware doorbells and issues involved in implementing them is the subject of our ongoing research and will not be discussed in this paper. We only discuss the implementation of doorbells in software. We also study different implementations of virtual-to-physical address translation and the possibility of caching descriptors. These two issues relate to the second basic component or the mechanism through which the NIC obtains information about the outstanding operations and corresponding user data buffers. The third component, the mechanism through which the user program is informed of the completion of send and receive operations, is also discussed with respect to the implementation of completion queues.

4.1 Doorbells

VIA specifies that each VI be associated with a pair of doorbells. The purpose of a doorbell is to notify the NIC of the existence of newly posted descriptors.

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 process' address space. 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 do not occur for receive descriptors. Since the receive descriptors are examined 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.

4.2 Caching Descriptors

As discussed in Section 3, 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. We'll have a performance evaluation of these methods in Section 5.1.1.

4.3 Address Translation

Most NICs (including the widely used PCI based NICs) use physical addresses for performing DMA operations, whereas VIA descriptor elements, e.g. user buffer addresses, are virtual addresses. Therefore virtual-to-physical address translation is required. 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.

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.

We discuss the cost of implementing these approaches for the virtual-to-physical address translation in Section 5.1.2.

4.4 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 increased in the number of VI work queues associated with a CQ. However, since in many applications each node communicates only with a small set of other processes, and therefore a limited number of work queues are associated with each CQ, this approach may be viable for implementing CQs.

We compare the cost of the implementation of these two approaches in Section 5.1.3. We also investigate how the scalability issue of the second approach can be dealt with.

5 Performance Evaluation

In order to evaluate different design choices discussed in this paper, we implemented a subset of VIA on two different systems. The first system consisted of 300 MHz Pentium II PCs with 128 MB of SDRAM and a 33 MHz/32-bit PCI bus and ran the Linux 2.0 operating system. The Myrinet switches and 33 MHz LANAI 4.3 NICs were used as an interconnect. The second system was an IBM Netfinity SP switch-connected Cluster [7]. 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. These two testbeds represent a wide range of available network-based computing platforms.

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

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 both systems. For our NT testbed we used the Fast IO Dispatch method [17] and for our Linux testbed, we used a fast trap. These measurements are presented in Table 1.

Table 1: Cost of basic operations in the Myrinet-Linux and SP-NT testbeds.

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

5.1.1 Caching Descriptors

As discussed in Section 4, the choice of caching the send descriptors when they are posted depends on the cost PIO and DMA operations. From the cost of these operations in our testbeds (Table 1), it can be seen that transferring up to ten words through PIO is less time consuming than using the DMA in the Netfinity SP system. In the Myrinet-Linux testbed, transferring up to five words can be done in a faster manner by using PIO. It should be noted that in neither of our testbeds, PCI write combining was used. If a system supports PCI write combining, a larger number of words can be transferred by PIO before the point where using DMA becomes more efficient. Another factor which affects the decision about caching send descriptors is the CPU utilization. While the host processor is not involved if DMA is used, using PIO requires the host to perform the transfer and increases the host CPU cycles used for send operations.

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

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

5.1.2 Address Translation

Three approaches for performing the address translation were discussed in Section 4. In the first approach (AT1), where the TLB is in the host memory and the host performs the translation, the cost of the address translation is essentially the one time user space to kernel space switch for each send or receive operation and the cost of the table lookup for each page frame of the send or receive buffer. In order to reduce the TLB lookup cost, one table for each registered memory can be created upon the registration of the memory region. This table includes the physical addresses of (the beginning of) all the page frames that the memory region spans over. By creating such a table, the virtual-to-physical address translation can be done by indexing the address translation table without any need for searching the table or multiple indirections. The Average cost of the address translation when the AT1 approach is used, is shown in the first row of Table 2. 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 of implementing the virtual-to-physical address translation. (See Figures 1 through 4 for the value of Miss Rate for different benchmarks.)

Address Translation Method	Location/ Translator	NIC Memory Requirement	Myrinet-Linux Avg. Fixed Cost	SP-NT Avg. Fixed Cost
AT1	host/host	None	1.06	2.27
AT2	NIC/NIC	Proportional	0	0
AT3	host/NIC	Constant	$1.72 \times Miss\ Rate$	$1.78 \times Miss\ Rate$

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 the third approach a more realistic and scalable approach for implementing the address translation.

In the third approach (AT3), the NIC perform the translation while the TLB is stored in the host memory. Since the TLB is stored in the host memory, the memory requirement on the NIC is minimal. However, if for every address translation the NIC is required to access the host memory (through DMA) this approach performs much worse than the second approach. In order to reduce the cost of the address translation while the size of required NIC memory is kept low, caching the address translations is used. If the translation of a particular physical address is found in a software cache (kept in the NIC memory), the translation can be performed quickly by accessing the corresponding cache entry. If the translation is not found in the cache, an access to the TLB in host memory (through DMA) is required (Table 2).

In order to evaluate the effectiveness of caching and estimating the required cache size, and in the absence of the existence a wide variety of applications/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 benchmark with 4, 16, and 64 processes and used two different problem sizes: class A and class B. We used different TLB cache sizes and degrees of associativity. It should be noted that the TLB cache is implemented in software and is stored in the NIC memory. (We haven't presented the data for the Embarrassingly Parallel (EP) benchmark because the number of communication operations used in this benchmark is so small that the performance of the address translation and other components of the communication subsystem does not affect the execution time of the program significantly.)

Figure 1 shows the cache miss rates for the class A NAS benchmarks on a system with 128-entry direct-mapped caches. The results for running these programs on four and 16 processes are shown and cache misses are broken down into send and receive misses (compulsory and non-compulsory). It can be seen that with such a small cache and when four processes are used, in four of the benchmarks more than 80% of memory accesses result in a cache miss. When the programs are run on 16 processes the number of cache misses reduces significantly. If the cache size is increased to 1024 (Fig. 2), the cache miss rates for all benchmarks other than IS become negligible. Increase in the number of processes result in an decrease in message sizes and this compensate the effect of the increase in the number of messages being transmitted. It is interesting to see that miss rates are identical for a 1024-entry direct mapped cache or a 1024-entry cache with the degree of associativity of eight. The access time of a software direct-mapped cache is less than that for a software associative cache. Therefore, given the same performance, using a direct-mapped cache is preferred over an associative cache when implemented in software.

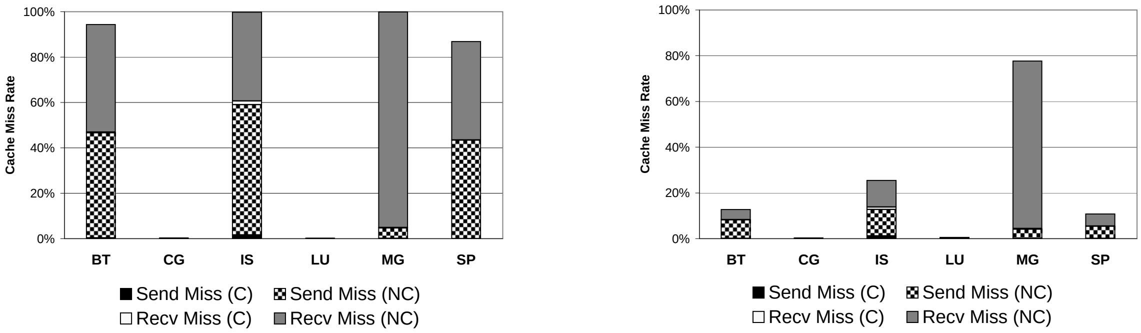

Figure 1: The cache miss rate for the NAS benchmarks (class A) using four processes (left) and 16 processes (right) with 128-entry direct-mapped caches. C and NC denote compulsory and non-compulsory misses, respectively.

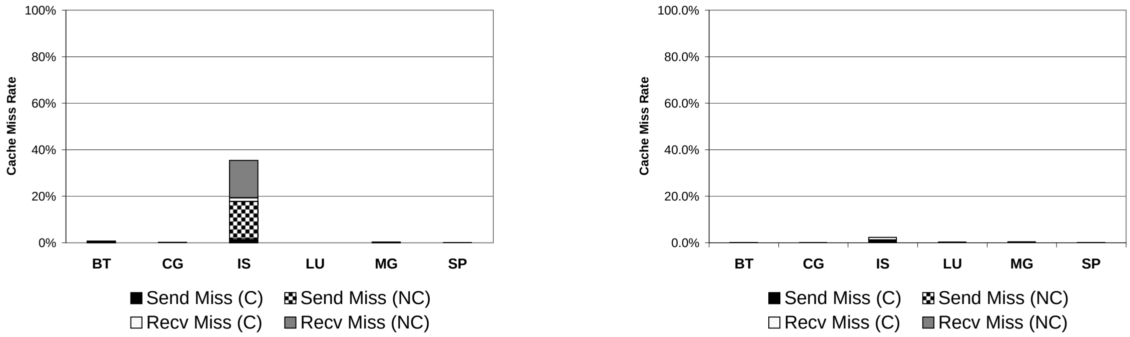

Figure 2: The cache miss rate for the NAS benchmarks (class A) using four processes (left) and 16 processes (right) with 1024-entry direct-mapped caches and 128-entry 8-way associative caches. (The miss rates are identical for both of these cache types.) C and NC denote compulsory and non-compulsory misses, respectively.

Figure 3 shows the cache miss rates for the class B NAS benchmarks on a system with 128-entry direct-mapped caches. Note that the results shown in this figure have been obtained from running these programs using 16 and 64 processes. The cache miss rates for systems with 1024-entry caches are shown in Figure 4. A similar pattern to those for class A benchmarks (smaller problem size) can be seen. It is interesting to compare the cache miss rates for these benchmarks with different problem sizes. When the benchmarks use 16 processes, increasing the problem size (from class A to class B) result in an increase in the cache miss rates. Using caches with 1024 entries are shown to be enough to make the cache miss rates for all class A benchmarks negligible. However, when the problem size is increased, the BT and IS benchmarks produce a significant number of misses.

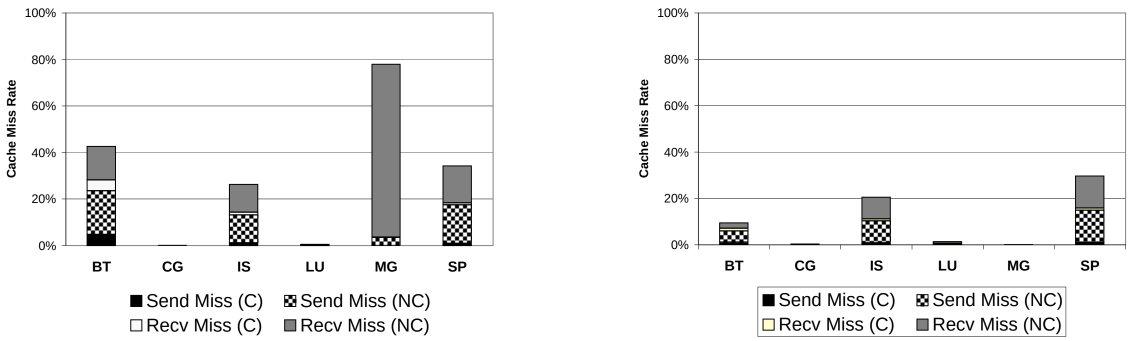

Figure 3: The cache miss rate for the NAS benchmarks (class B) using 16 processes (left) and 64 processes (right) with 128-entry direct-mapped caches. C and NC denote compulsory and non-compulsory misses, respectively.

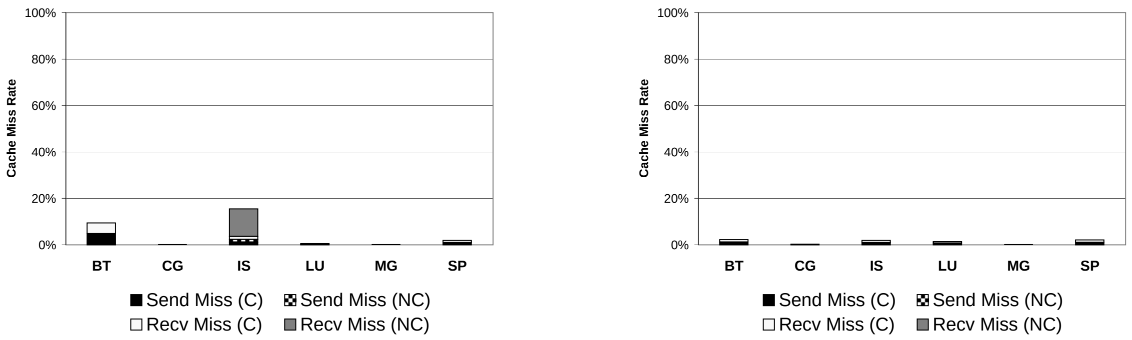

Figure 4: The cache miss rate for the NAS benchmarks (class B) using 16 processes (left) and 64 processes (right) with 1024-entry direct-mapped caches and 128-entry 8-way associative caches. (The miss rates are identical for both of these cache types.) C and NC denote compulsory and non-compulsory misses, respectively.

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

It should be noted that for receive operations, the cost of address translation might be hidden if the translation is done before the message arrives. The AT1 method can be easily used to take advantage of this characteristic. But the AT2 and AT3 methods can be implemented more easily if the translation is done when the message arrives. When the AT2 and AT3 methods are used, performing the translation before the message arrives increases the complexity of the firmware and can decrease the overall performance of the communication subsystem.

Another 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.

5.1.3 Completion Queues

We presented two approaches for implementing the completion queues in Section 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**	**Myrinet-Linux CQ1 Cost**	**SP-NT CQ1 Cost**
Any Program	n	k	$k \times 0.08$	1.47	1.61
LU	4	2	0.16	1.47	1.61
LU	16	3	0.24	1.47	1.61
LU	64	3.5	0.28	1.47	1.61
MG	4	3	0.24	1.47	1.61
MG	16	4.6	0.37	1.47	1.61
MG	64	6.5	0.52	1.47	1.61

6 Related Work

There have been several implementations of VIA. The Berkeley VIA (Version 1) [10] is one of the first software implementations of the VIA. (This implementation is a partial implementation of VIA mainly done to obtain a better insight on different aspects of the implementation of the VIA.) In this implementation, a memory page on the NIC memory has been used for the implementation of a pair of doorbells. The doorbells for send queues are polled for finding outstanding send descriptors. This polling is expensive and increases linearly with the number of active VIs. The Berkeley VIA does not perform any caching of descriptors. In other words, for sending messages NIC has to access the host memory twice: once for obtaining the descriptor and once for obtaining the data itself. In this implementation, only a subset of descriptors are moved between the host and the NIC to reduce the high cost of transferring the descriptors. The Berkeley VIA (Version 2) [9] is based on the the Berkeley VIA (Version 1) implementation and adds memory registration and increased VI/user support. In this implementation each memory page on the NIC can support up to 256 pairs of doorbells that belong to a single process. For the address translation a buffer with limited size on the NIC is used for the TLB. If the size of registered memory is bigger than what can be supported with this table, the translation of some portions of the registered memory won't be present in the NIC TLB. In these cases the host memory is accessed to obtain the translation. The location of the host buffers holding the complete translations for registered memory regions are known to the NIC. The FirmVIA [7] is an experimental implementation of the Virtual Interface Architecture for the IBM SP Switch-Connected NT cluster which is one the newest clustering platforms available. In this implementation, the address translation is performed by the host. Descriptors are also cached for improving the performance. The performance of GigaNet cLAN [1] and the Tandem ServerNet VIA implementations are studied in [16].

The effect of using caching for address translation for user-level network interfaces (and in particular U-Net) has been presented in [20]. The address translation issues have also been studied in [14] and the address translation methods are classified according to where lookup and the miss handling are performed. The major difference between the address translation in systems discussed in these papers and that in systems supporting VIA is the memory registration mechanism required by VIA. In VIA, all the memory locations used as send and receive buffers are in registered memory regions and VIA implementations are not concerned with the possibility of accessing a location which belongs to swapped page frames.

7 Conclusions

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

Acknowledgements

We would like to thank Darius Buntinas and Sencer Kutlug for their invaluable discussions and suggestions.

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, 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.

[7] M. Banikazemi, V. Moorthy, L. Hereger, D. K. panda, and B. Abali. Efficient Virtual Interface Architecture Support for IBM SP Switch-Connected NT Clusters. Submitted to International Parallel and Distributed Processing Symposium (IPDPS 2000).

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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/.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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