An Overview of Virtual Interface Architecture (VIA)

Hassan Shojania
shojania@ieee.org

Abstract

Cluster of standard servers are rapidly gaining popularity as an alternative for mainframe systems. Distributed and high performance applications running over these clusters require high performance communication subsystem. The proprietary network configurations developed for Systems Area Networks (SAN) so far have remedied this but have limited the scalability of the system because of high cost and lack of interoperability. Virtual Interface Architecture (VIA) is an initiative to achieve high performance communication through minimizing software overhead.

1 Introduction

Over the last decade, physical networks have been increasing in speed at a much greater rate than processors. Networks are increasing by orders of magnitude, while CPU speeds are following "Moore's Law", which states that computer processing power doubles every eighteen months [1]. And networking hardware supporting bandwidths in the order of gigabits per second have become widely available. But this is a misconception to equally associate high data rate of physical networks with high communication performance.

The layered nature of the legacy networking software, the usage of expensive system calls, and extra memory copies are some of the factors responsible for degradation of the communication subsystem performance as seen by the application. Figure 1 compares the increasing portion of software overhead stack against the hardware wire time when increasing physical network speed.

Clusters of standard servers are becoming more popular as a cost-effective alternative to large mainframe computers. But at the same time, software overhead has become the bottleneck for cluster computing where high performance of communication for inter-server communication is essential for scalability of cluster. To solve this, several high speed Systems Area Networks (SAN) had been developed prior to VIA. But the interface and operation of these networks were unique and proprietary, which limited the number of applications developed for them. This has lead to market fragmentation and limited return on investment.

To fill this gap between standardization and high performance cluster communications, Intel, Compaq and Microsoft initiated Virtual Interface Architecture (VIA) in 1996 and published its first specifications in December 1997. Since then, the standard has been accepted by the industry; hardware vendors have developed VI-compliant products and new commercial applications developed or adapted to use VIA.

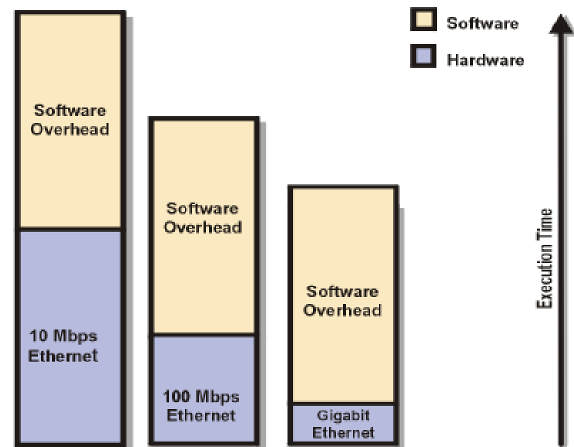

Figure 1: Software overhead vs. HW wire time (from [2])

VIA has roots in previous academic and industrial researches. Active Messages at UC Berkeley, Fast Messages at UIUC, Shrimp at Princeton, and specially U-Net at Cornell have all affected VIA. VIA is independent of OS, CPU and network media and targets portability and compatibility between different HW/SW platforms. By employing user-level messaging, VIA provides direct access to the network interface (minimizing number of instructions to execute), avoids intermediate copies (by bypassing OS virtualization/protection) and interrupts. At the end, it provides a protected and zero-copy form of User-Level Networking (ULN).

In this paper we intend to present VIA's main concepts and how low software overhead is achieved. We first start with overview of software overhead in Section 2. Then main concepts and operation of VIA is explained in Section

3. VIA's flexibility allows designer to explore different design alternatives when implementing VIA components for a particular HW/SW platform. Two examples are provided in Section 4. Two open VIA implementations (*M-VIA* and *Berkeley VIA*) are briefly overviewed in Section 5 as case studies. InfiniBand Architecture (a new standard superseding VIA in many areas) is briefly compared with VIA in Section 6. In Section 7 we present our conclusions and future of VIA.

The framework of this paper is based on [1] and [3]. They and [4] should be referenced for more information about VIA fundamentals and VIA APIs.

2 Software Overhead

There are several factors contributing to software overhead in a message transfer:

- Many layers of software boundaries crossed from application, protocol stack and OS kernel before reaching the Network Interface Card (NIC) device driver.
- Several potential data copies when crossing boundaries of above layers.
- Number of interrupts and context switches incurring for transfer of a message.

Note that the time to traverse the software stack layers, latency of interrupts and the time to complete context switches do not depend on the number of bytes sent or received. Only the time to complete the data copies depends on number of bytes being transferred. The problem is that it is not only a single layer responsible for most of the overhead, so tuning a single layer alone can hardly lead to significant decrease in latency.

Since most of software overhead comes from "code execution", faster processors decrease the overhead. But the penalty paid for context switches and handling interrupts have not decreased linearly with increasing processor performance[5]. Larger penalties must be tolerated for reshuffling CPU state at context switches and interrupts because of the higher depths of pipelines. Further, these extra executed code is wasting CPU time that can be used for other applications on the system.

At the end all these overheads lead to high latency and low bandwidth as we will see below. The time to transfer a message can be approximated by

$$t(m) = t_0 + m/r_\infty \tag{1}$$

where t_0 is the fixed, non-overlapped time per message mainly contributed by software overhead (i.e. transfer time of a message of zero size), m is the message length and r_∞ is the asymptotic bandwidth of physical network.

Let's take the effect of message size on software overhead t_0 negligible for a while (assuming it to be constant). Figure 2 shows how the graph of total latency $t(m)$ versus message size m changes with three values of software overhead t_0 on a 1 Gbps network (derived from equ. 1). It shows that for small messages, software overhead dominates the total latency incurred (as $t_{wiretime} = m/r_\infty$ is small). Also software overhead has significant impact on total latency throughout the range of message sizes. (at $t_0 = 1\mu s$, and $m = 8192$ bytes, $t_{wiretime} = m/r_\infty = 8192 * 8 * 1Gbps = 65.5\mu s => t(m) = 66.5\mu s$, while for $t_0 = 62\mu s$ it'll end up with $t(m) = 127.5\mu s$; an almost double-fold increase.)

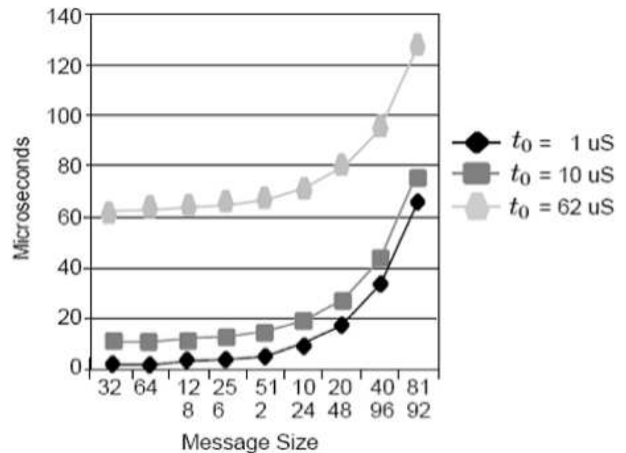

Figure 2: Effect of software overhead on latency (from [1])

Figure 3 shows the effect of software overhead on the best achievable bandwidth for the same one-gigabit network (using the total latency equation 1). The severe impact of software overhead is obvious. Even with only $t_0 = 10\mu s$ and $m = 1024$ bytes, less than half of the available bandwidth can be achieved ($t_{wiretime} = 1024 * 8 * 1Gbps = 8.1\mu s$; $Bandwidth = (8.1/(8.1 + 10)) * 1Gbps = 450.3Mbps$).

Now the situation becomes worse if we consider the effect of message size on software overhead t_0 (increasing it with message size because message copy portion of the software overhead depends on the message size). Note that above bandwidth graph calculation does not assume pipelining the message sends (sending another message as soon as software hands in the message to the NIC), ignoring the fact that it can execute another send while the previous message is in transit over wire. This pipelining can help the effective bandwidth for larger messages (e.g. with sizes above half-message size $N_{1/2}$ where wire-time m/r_∞ is more than the overhead t_0). But it doesn't help

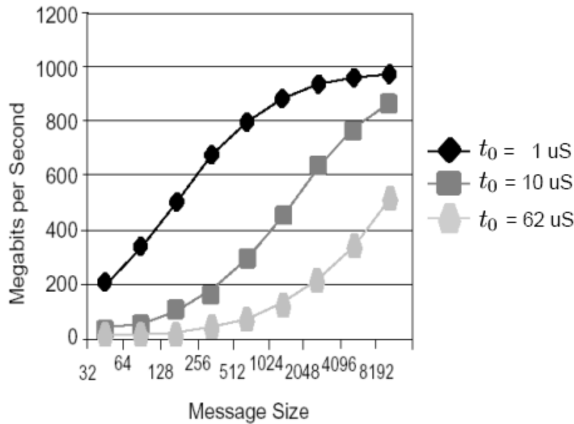

Figure 3: Effect of software overhead on bandwidth (from [1])

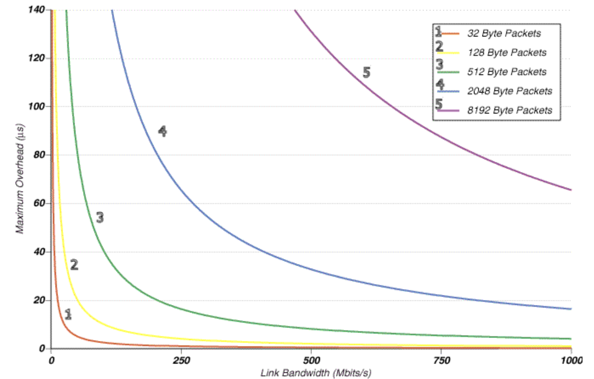

Figure 4: Maximum allowable overhead to achieve a throughput of one-half of the link rate for message sizes of 32, 128, 512, 2048 and 8192 bytes (from [7])

with short messages as $t_{wiretime}$ is much less than t_0.

What makes the matter worse is a well-known rule of thumb known as *80/80 rule of network distribution*. This rule states that 80% of messages transmitted over a network have less than 256 bytes while 80% of the total data is delivered by messages of greater than 8192 bytes in length. For example a study of a network running Network File System (NFS) traffic at University of California, San Diego (UCSD) shows this bi-model behavior: 86% of the NFS messages were less than 200 bytes, while only 9% were more than 8192 bytes. Most of the small messages are control and synchronization related, while the long messages are bulk data exchanges. This high frequency of short messages amplifies the importance of minimizing software overhead as even pipelining messages can not help anymore (overhead is larger than wire time) and good portion of physical bandwidth remains unused.

Now let's see what can be a tolerable overhead for short messages. If we want to match overhead time and wire time of a 200 bytes message on a 1 Gbps network, overhead must be equal to wire time of $200*8*1Gbps = 1.6\mu s$. This time translates to only 320 instructions on a 200 MIPS processor, or 1600 instructions on a 1000 MIPS processor. With all the layers we saw already, even an overhead of 1600 instructions is hard to achieve. Now if a message transfer incurs an interrupt or context switch the situation becomes much worse. Figure 4 shows maximum tolerable overhead to achieve equal overhead and wire time for a range of bandwidth and different message size. As graph 3 of the figure shows, even for a message size of 512 bytes overhead can't be more than $5\mu s$, an almost impossible overhead for traditional networks. And the situation is worse for lower size messages or when bandwidth increases.

By bypassing the kernel, the software overhead can be decreased so much that a study [10] showed much less round-trip time can be achieved for a remote procedure call (RPC) with distributed COM (DCOM) between two systems over VIA compared to a local inter-process call (local COM) under Windows NT.

Figures 5 and 6 compare bandwidth and latency of running a test application over GigaNet cLAN with VIA support (using VIPL) with the same application over GigaBit Ethernet (using Sockets). In this configuration two servers (P3 800 MHz) were directly connected.

As Figure 5 shows that round-trip latency of a single message is significantly decreased with VIA. For example the wire time to total latency of a 1024 bytes message is increased to half of the total latency, meaning less overhead ($1024*8*1Gbps*2_{round-trip} = 16.5\mu s$ out of total $35\mu s$). Figure 6 shows while TCP/IP networking fails to achieve half of the network bandwidth even for large messages, VIA case can achieve half of the network bandwidth with message sizes as low as 512.

3 VI Architecture Overview

In the traditional network architectures, the operating system (OS) virtualizes the network hardware into a set of logical communication endpoints available to network consumers and multiplexes access to the hardware among these endpoints. In most cases, the operating system also implements the protocol stack (usually in kernel). This model allows a simple interface between the network hardware and the operating system. But the drawback is that all communication operations require a call or trap into the operating system kernel, which can be quite expensive to execute. The demultiplexing process and reliability protocols also tend to be computationally expensive.

To reduce software overhead, VIA bypasses the oper-

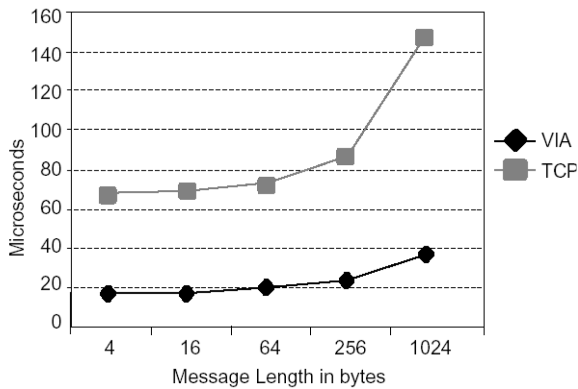

Figure 5: Measured round-trip latency (from [1])

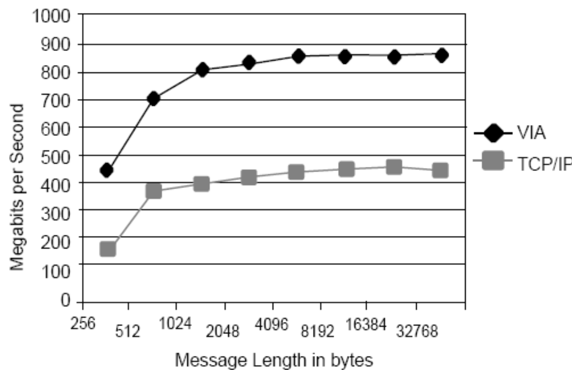

Figure 6: Measured bandwidth (from [1])

ating system kernel in the performance critical data path. This leads to some challenges itself as VIA needs to provide each consumer process with a protected, directly accessible interface to the network hardware, a Virtual Interface (VI). Each VI is owned by a single process and represents a communication endpoint where pairs of them can be logically connected to support bidirectional, point-to-point data transfer. A process may own multiple VIs exported by one or more network adapters. A network adapter directly performs the endpoint virtualization, multiplexing, de-multiplexing, and data transfer scheduling normally performed by OS kernel and device driver.

VIA separates the control and data paths to enable OS bypass. The control path is responsible for setup of connections, memory buffer management and protection. VIA is optimized for the data path where the actual message-passing operations, such as Send, Receive and Remote DMA (RDMA) happens without buffer allocation. This is in contrast to legacy networking stacks where they prepare transmit buffers in-line with transmission operation itself (i.e. preparing the buffer after the transmit request started). For Receive operations, legacy network stacks use intermediate receive buffers that are subsequently copied to the destination buffers of the application. VIA's memory registration in the control path eliminates the need for buffer preparation, virtual to physical address translation and memory locking at transmission time.

3.1 Data transfer model

VIA provides three operations for data movement:

Send/Receive A source application sends a sequence of bytes by issuing a Send operation. On the destination, the destination application executes a Receive operation to receive the data in a pre-allocated location. Send/Receive operations can use scatter/gather lists for their buffers. Basically source buffer of the Send operation can gather the data from multiple dis-contiguous buffers. On the destination, the data can be scattered between multiple dis-contiguous buffers. The scatter/gather mechanism can increase performance of packet header/trailer manipulation of layered protocols by eliminating extra data copies.

RDMA-Write The RDMA-Write operation allows data to be transferred into a buffer at the destination application. By executing a RDMA-Write, sending application *pushes* data from possibly dis-contiguous memory locations to a specific contiguous memory location at the destination side (gather but no scatter). The destination process must communicate this destination address to the sender process prior to issue of RDMA-Write by sender.

RDMA-Read By executing a RDMA-Read, the initiating application (receiver) *pulls* data from a specific contiguous memory location at the remote sender application (which already communicated its buffer address to the receiver). This data can be potentially scattered into multiple dis-contiguous destination buffers at the receiver but data can not be gathered from sender.

The Receive Assumption: The VIA eliminates the need to copy messages at the receiving side by enabling incoming data to be placed directly into the application buffers. To do this, NIC of the receiver must have been told about the location of application buffers in advance; meaning there must be a Receive message posted by destination already.

3.2 VI Architecture Components

The VI Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI

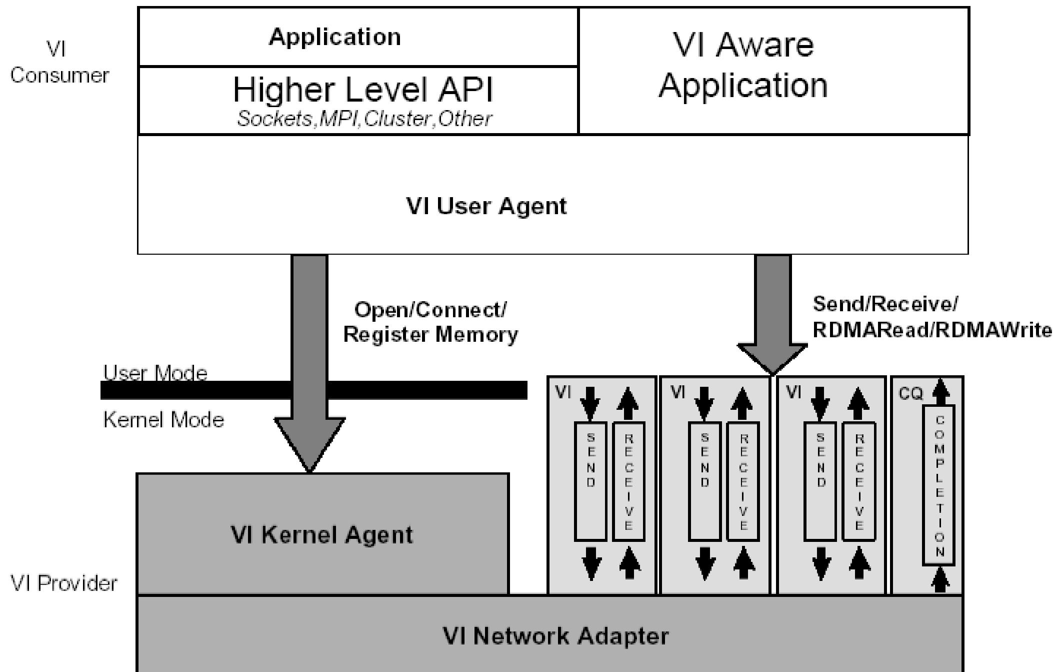

Figure 7: VIA Architectural Model (modified from [4])

Providers, and VI Consumers. The VI Provider is composed of a physical network adapter and a software Kernel Agent. The VI Consumer is generally composed of an application program and an operating system communication facility. The organization of these components is illustrated in Figure 7.

3.2.1 Functional partitioning

VIA partitions the functionality (normally provided by OS) to the VI Consumer and the VI Provider as shown in Figure 7. VI Provider is the software (Kernel Agent) and hardware (VI NIC) components implementing the VIA. The users of a VI Provider are VI Consumers. They use the software interface provided by ”VI User Agent” block. An example implementation of ”VI User Agent” block called Virtual Interface Provider Layer/Library (VIPL) is provided in VIA specifications. Though VIPL interface is not a de-jour standard, it has become the de-facto API standard. Applications can be *VI-aware* and directly use VIPL interface or can use a higher-level abstract (like a middleware).

A VIA Consumer (VI-aware applications or another higher-level API) allocates and manages its own buffers, allowing it to optimize its message buffering strategy in contrast to kernel-based network stacks. It also directly notifies the VI-NIC for transmitting or receiving new messages, while this normally handled by operating system. On the other hand, a VI Provider manages protected sharing of the network controller, virtual to physical translation of buffer addresses and synchronization of completed work via interrupts, all normally handled by OS. It also provides a reliable transport service with a level depending on the underlying network.

3.2.2 VI Instances and its work queues

As already described, a VI is a communication endpoint for the point-to-point communication. It provides the illusion to each process that it owns the interface to network (see Figure 7). Different VIs (owned by different or single process(es)) are protected against each other. Each VI consists of one send queue and one receive queue and is owned and maintained by a single process. Each VI queue is formed by a linked list of variable-length descriptors. To add a descriptor to a queue, the user builds the descriptor and posts it onto the tail of the appropriate work queue using VipPostSend() or VipPostReceive(). Send, RDMA-Write and RDMA-Read operations are posted to the send queue (by the operation initiator), while Receive operation is posted to the Receive queue.

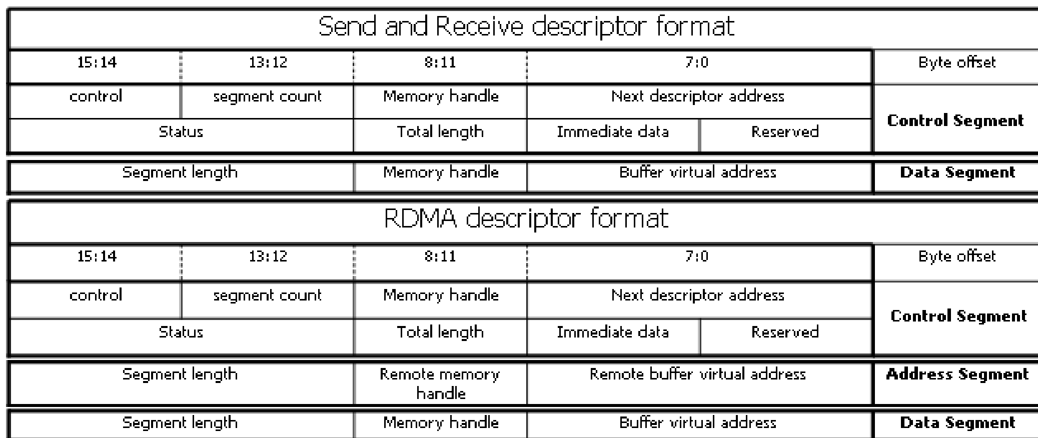

Figure 8: Format of Send/Receive and RDMA descriptors

Descriptors A data movement operation is started by creating a descriptor specifying the operation and posting it to the appropriate work queue. After being posted, the descriptor is linked to the tail of the queue. Descriptors reside in a host memory that can be accessed by both the application and VI NIC (see Section 3.2.5). There are two types of descriptors: Send/Receive descriptors for Send/Receive operations and RDMA descriptors for RDMA-W/R operations. Descriptors always consist of a *control segment* and a variable number of *data segments*. Appearing at the beginning of a descriptors, the control segment contains control and status information, and fields used for queuing. The data segments form a scatter/gather list describing the address and length of the *local buffers* of a Send, Receive, RDMA-Write or RDMA-Read operation.

The descriptor for a RDMA-Read or RDMA-Write operation includes an extra segment called *address segment*. An address segment specifies the length of a single *remote buffer*. This buffer is the source of an RDMA-Read and the target of an RDMA-Write. Since only one address segment is allowed, the remote buffer is always a single contiguous buffer preventing buffer scatter for RDMA-Write or gather for RDMA-Read. Figure 8 shows the format of both descriptor types. See [1] or [4] for more information about each fields.

When VI-NIC has finished the operation, it sets the *done* bit of the status field. Applications can either poll for completion by calling VipSendDone()/VipReceiveDone() or do a blocking wait by calling VipSendWait()/VipReceiveWait(). Since descriptors are kept in the host memory, polling is not expensive at all. But blocking wait usually causes a couple of system calls, a thread switch and an interrupt; so it is much more expensive than polling. When VI NIC updates the "done" bit, it usually has to use a DMA to update the host memory.

Doorbells When an application posts a descriptor to a work queue, VIPL uses a *doorbell* to notify VI NIC about availability of the new task on related work queue. Each Send/Receive work queue has its own associated doorbell. A doorbell can be a VI NIC register memory mapped to the application address space. VI NIC keeps track of the head descriptor of the queue and the count of posted descriptors at any time.

Notification through doorbell is done as below steps:

1. VIPL adds the descriptor to the work queue (by updating the tail's descriptor "next descriptor" field) after application posted the request.

2. VIPL rings the doorbell associated with the queue (e.g. by writing to the VI NIC register or memory location). The NIC increments the count of outstanding descriptors and if this is the first arriving descriptor to the queue, it also updates its head pointer.

VI NIC processes the descriptors like this:

1. It uses its pointer to the head descriptor to pick the next request.

2. After finishing with the request, it updates the status field of the descriptor.

3. VI NIC decrements the count of outstanding descriptors and follows the link to the next descriptor by updating its pointer.

3.2.3 Completion Queues

It is often convenient for a process to poll or block for completion on more than one work queue at a time. When a VI is created, its work queues can be associated to one or

different completion queues. A completion queue belongs to a process and can aggregate completions from multiple work queues (even queues belonging to different VIs of the process). So the process can wait on completion queue(s) instead of using a thread for waiting on each one of the work queues separately.

Both polling and blocking functions are supported on completion queues (VipCqDone()/VipCqWait()). They return the VI handle and its work queue identifier that completion notification has arrived on, so the application can figure out which one of its work queues has a completed task.

3.2.4 Protection

VI-NIC must ensure that a process is not writing to or reading from another process memory. Protection-tag is the mechanism used for this purpose and is created by calling VIPL's VipCreatePtag(). It is an opaque data type containing a unique value assigned by the Kernel Agent. When creating a VI through VipCreateVI() or registering a memory location through VipRegisterMemory(), they must be associated with a protection tag (multiple VIs/memory regions within a process can use the same tag). Other VIPL calls crossing the Kernel Agent are verified against the protection tag associated with the VI/memory region to reject calls using incorrect protection tags.

3.2.5 Memory registration

VIA allows applications to use virtual addresses when referencing memory locations. But VI NIC needs to access the memory with physical addresses, so translation is required. Also, VI NIC is the body to enforce the memory protection among applications and between application and OS. The mechanism is called Translation and Protection Table (TPT). Its implementation is beyond the spec. and only a reference implementation provided. Each table entry keeps track of a single virtual memory page, so the granularity of memory registration is page-based that could lead to waste of memory. A memory region being registered may span multiple pages but must be contiguous. The entry assigns a memory handle to the virtual address and records its physical address, protection attributes and the protection tag passed by VIPL.

Here is the sequence of memory registration when VipRegisterMemory() is called:

1. Kernel Agent locks (pins down/wires) the memory to prevent it from being paged out.

2. Kernel Agent allocates one TPT entry for each page in the memory region and sets its protection tag and associated attributes.

   Protection attributes can be a combination of:

   Read access: By default all memory has Read access.

   Write access: The memory can be destination of a Send operation.

   RDMA-Read access: The memory may be the source of a RDMA-Read.

   RDMA-Write access: The memory may be the destination of a RDMA-Write.

3. Kernel Agent creates and returns a memory handle associated with the region.

Later, application will use the virtual address and its associated handle to reference the memory region used in transmit operation (i.e. in descriptors). VI INC will retrieve the proper TPT entry knowing both virtual address and its handle.

A sample translation mechanism can do like below. For associating a handle to a virtual address can use:

$$MemoryHandle = (VirtualAddress >> 12) - TPTIndex$$

And to find the proper TPT entry after receiving the virtual address and memory handle pair:

$$TPTIndex = (VirtualAddress >> 12) - MemoryHandle$$

Figure 9 shows a sample translation from a handle, virtual memory pair back to TPT entry to retrieve the physical address.

Assume:

1. Two memory regions from an application one with virtual address 0x40006000 and length of one page and the other region starting at 0x40008000 and spanning two pages where they've been already assigned TPT entries 0x101, 0x126 and 0x127.

2. First region has a protection attributes of 0x02 (e.g. Write access only) and its associated memory handle is 0x3FF05 while second region has an attributes of 0x06 (e.g. both Write and RDMA-Write access) and its associated handle is 0x3FEE2.

3. And the application has registered both regions with the same protection tag of 0x78 to use both with a single VI.

The drawback of the above simple method is that it can not detect a bad pair of virtual address and memory handle. For example if somebody uses virtual address 0x40006000 with the handle 0x3FF004 he will end up with TPT entry 0x102 which is not valid. Also references made to a location beyond the registered memory will not be caught. For example if somebody passes virtual address 0x40007000 with a handle that falls into a valid TPT entry for the same process, a wrong physical memory can be accessed.

Of course above problems can be caught by improving the scheme like adding more fields to TPT entry (like the

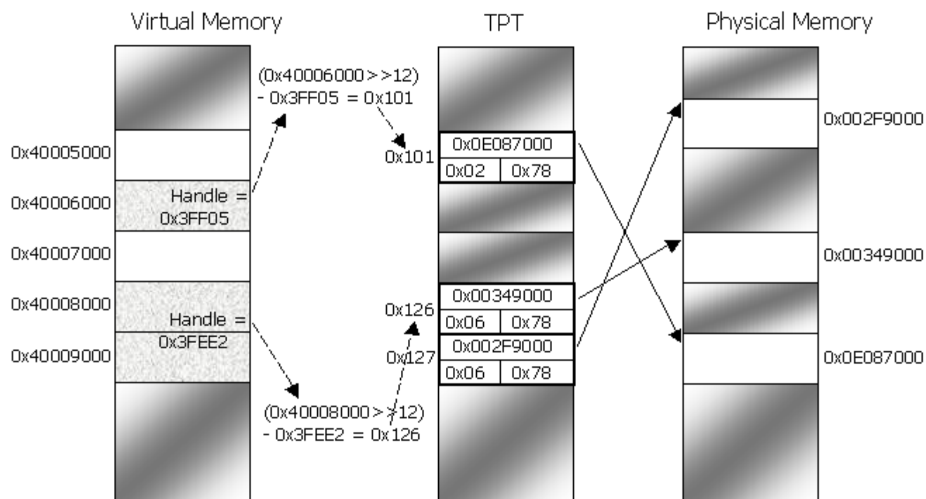

Figure 9: Memory registration and TPT

virtual address of memory the entry was allocated for) and better mechanism for handle generation. Note that even in the above simple scheme, a bad behaving application can not access memory locations registered by another application because their protection tag (which is VI dependent and as the result process dependent) will be different. So this simple TPT scheme may let a process "shoot itself in the foot", but protection tags prevent it from harming another process.

3.3 Managing VI Components

This section discusses how other components of a Virtual Interface are created, destroyed, and managed.

3.3.1 Accessing a VI NIC

A VI Consumer gains access to the VI Provider by retrieving a handle to the Kernel Agent (before creating any VI) using standard operating system mechanisms. The VI Consumer uses this handle to perform general management operations such as registering memory regions, creating Completion Queues and creating VIs. This mechanism would also be used to retrieve information about the VI NIC, such as the reliability levels it supports and its transfer size limits. A Kernel Agent must use standard operating system mechanisms to detect when a VI Consumer process exits so that it can cleanup any resources used by the process. And it must also keep track of all resources associated with a VI Consumers use of a VI NIC.

3.3.2 Connection model

VIA provides connection-oriented data transfer service. A VI must be connected with another VI through a deliberate process in order to transfer data. When data transfer is completed, the associated VIs must be disconnected. The endpoint association model can be client-server or peer-to-peer.

At VI creation, the network address of the other party is passed to VipCreateVI(). This network address has two parts: A unique host address, and an endpoint discriminator. The format is implementation specific (e.g. it can be like [Node 1, foo server]).

Client-Server model This model is asymmetric. The server's VI Consumer issues a VipConnectWait() request to its VI Provider specifying the address discrimination values that are acceptable to the VI Consumer. A VI Consumer should be able to accept a connection from any remote endpoint or a specific remote endpoint, based on the discriminator supplied. Sometime after the server VI Consumer begins waiting for a connection, the client VI Consumer issues a VipConnectRequest() request to its VI Provider specifying the remote VI to which to connect, and a timeout value. Then server's VipConnectWait() request completes, and information about the client VI is returned to the server VI Consumer. The server side either accepts the request (prepares a VI for connection and issues a VipConnectAccept() request) or rejects it (by calling VipConnectReject()) based on the attributes associated with the remote VI (e.g. reliability). On the client side, VipConnectRequest() request returns successfully if the connec-

	Unreliable	Reliable Delivery	Reliable Reception
Detecting corrupt data	Yes	Yes	Yes
Data delivered at most once	Yes	Yes	Yes
Data delivered exactly once	No	Yes	Yes
Data order guaranteed	No	Yes	Yes
Data loss detected	No	Yes	Yes
Connection broken on error	No	Yes	Yes
RDMA-Read supported	No	Optional	Optional
RDMA-Write supported	Yes	Yes	Yes
Send/RDMA-Write descriptor marked done	When data leaves initiator	When data leaves initiator	When data arrives at target

Figure 10: VIA reliability comparison

tion request was accepted. If the request is not accepted within a time-out, it'll fail.

Peer-to-peer model This model is symmetric, both sides executing similar methods to connect. Each peer first creates a VI and issues a connect peer request by calling VipConnectPeerRequest() passing its VI. Then they block or poll on the status of the request by calling VipConnectPeerDone()/VipConnectPeerWait(). If the request is not accepted within a time-out or VI reliability levels of the two peers doesn't match, VipConnectPeerDone()/VipConnectPeerWait() will fail.

3.3.3 VIA reliability

VIA assumes that the underlying network is highly reliable, if not perfectly so. VIA has several reliability modes into which an endpoint can be switched. In the least-reliable mode (unreliable service), VIA simply attempts to transmit packets. Only if something completely prevents it from placing the packet onto the wire is an error signaled. In reliable delivery mode, VIA signals an error if the packet did not make it onto the wire successfully. Finally, for reliable reception, an error is signaled if the packet is not received into the destination host's memory successfully.

In all of these schemes, the failure is catastrophic – all following transactions on that endpoint are discarded and the endpoint is placed into an error state. Figure 10 shows the difference between these reliability levels.

4 Design alternatives

Since VIA specifications is flexible, designers can end up with different design choices based on their underlying hardware/software. Here we briefly overview two case studies to show some of the available alternatives.

4.1 VIA on IBM Netfinity NT Cluster [6]

FirmVIA is an experimental implementation of VIA for IBM SP Switched-Connected NT cluster which has surpassed other software implementations of VIA. The peak measured bandwidth was $101.4 MBytes/s$ and a one-way latency of $18.2 \mu s$ was achieved for short messages. Two of the design decisions taken in *FirmVIA* are as follows:

4.1.1 Caching descriptors

We know a VI initiates a data movement operation by posting a descriptor. As descriptors are kept in the host memory, NIC needs to retrieve the content of the descriptor before initiating the transfer. For most NICs this requires a DMA as this is the only way to access the host memory. The other approach is to make transfer of descriptor host-initiated by using PIO at time of descriptor posting. But there is a trade-off between DMA and PIO depending on the amount of transferred data; PIO being more efficient for transfers of up to 5 words (in this platform). For the receive operations, NIC needs to find the proper descriptor as soon as it sees the VI identifier of the receiving process in an incoming packet. Having the descriptor already in the NIC memory will decrease the experienced latency of the receive operation. On the other hand, NIC memory is limited and not all posted descriptors can be kept in it. The solution is to cache portion of descriptors in the NIC memory and keep address and size of the data buffer. As they learned, descriptors have high locality of reference and the achieved performance was good.

4.1.2 Software Doorbell

A software doorbell can be implemented as a bit in NIC memory mapped to a system address. But since each Send/Receive work queue of every VI needs its own doorbell, NIC could end up a good portion of its time polling all

doorbells especially when number of VIs increase (an overhead of $0.6\mu s$ per VI). The other approach is to use a centralized queue of send descriptors maintained by the NIC where after each descriptor post, the descriptor is added to the queue, so only head of the queue needs to be polled by the NIC. Since all VIs need to share this queue, appending to the queue must be controlled in an operating system safe fashion; and kernel intervention is required to do this. The added cost of a kernel call ($2.27\mu s$) was found to be justifiable relative to overhead of the first approach and this method was employed.

4.2 SOVIA [8]

SOVIA (Sockets Over VIA) is the implementation of a user-level Sockets layer over VIA. The objective was to use the SOVIA layer to accelerate the existing Sockets-based applications with a reasonable effort and to provide a portable and high performance communication library based on VIA to application developers. SOVIA achieves comparable performance to native VIA, a minimum latency of $10.5\mu s$ and a peak bandwidth of $814Mbps$ on Giganets cLAN. Their ported FTP (File Transfer Protocol) and RPC (Remote Procedure Call) applications over the SOVIA layer performed very well; easily doubling the file transfer bandwidth in FTP and reducing the latency of calling an empty remote procedure by 77% in the RPC applications.

There can be several different approaches to support Sockets API:

1. Traditional implementation is all in kernel with the Sockets layer located on top of TCP and UDP protocol stacks in kernel.
2. A simple way to support Sockets API on VIA is to insert an adaptation layer between IP and VIA's Kernel Agent. However, it is not straightforward to emulate connectionless IP services on the connection-oriented VIA, and applications still suffer from the overhead of TCP/IP protocols.
3. The TCP/IP protocol stack is not required to transfer data between two end-points on the same cluster if the physical interconnect is reliable and provides transport-level functionality. The overhead of TCP/IP protocols can be eliminated on VIA by collapsing kernel software layers and emulating Sockets API directly over the VI Kernel Agent.
4. And finally the best approach is to implement the Sockets to VIA translation layer in user-mode to eliminate the context switching and data copying experienced between user and kernel space experienced in approach 3. So SOVIA layer decided to entirely be implemented at user-level on top of VIPL.

Here are a couple of decisions the authors made in their implementation:

4.2.1 Memory registration vs. copying

The memory registration requirement for sender is more expensive than the receiver because it is not easy to pre-register all send buffers and some must happen in-line with the send operation. But on the receiver side, there is usually time between issuing a receive and when data really arrives because of *the Receive Assumption*.

Instead, SOVIA considered using sender-side buffering, where the outgoing data is copied into the pre-registered buffer before the corresponding descriptor is posted to a send queue. This breaks the zero-copy advantage so it was implemented in hybrid fashion; using copy to pre-registered buffers for messages of size less than 2KB and resorting to the normal memory registration for any message larger.

4.2.2 Combining small messages

TCP's Nagle algorithm requires that when a TCP connection has outstanding data that has not yet been acknowledged, small messages cannot be sent until the outstanding data is acknowledged or until TCP can send a full-sized message. SOVIA employs similar approach by combining small messages. The pre-registered buffer already described in previous section is not used for a single small message anymore. From now on, such small messages are appended into the buffer and the sender starts a timer. Any other small messages requested within the expiration of the timer are also combined into the buffer (up to $32KB$) and then transmitted to the network. Before the buffer gets filled or the timer expires, any arriving message of more than $2KB$ will trigger the previously combined pre-registered buffer. The large message will be send later without combining.

5 Case Studies

Here we briefly overview two open source implementation of VIA.

5.1 M-VIA [9]

M-VIA (Modular-VIA) is a complete high-performance implementation of the Virtual Interface Architecture for Linux (kernels 2.2.x and 2.4.x), written at the NERSC center at Lawrence Berkeley National Laboratory. Some of its features are:

- Modular design that makes it easy to port M-VIA to new hardware. If an appropriate device class already exists for a network card, only a few changes to the device driver will be needed.
- Hardware accelerated or full software implementation, depending on hardware support.
- High performance, through the use of fast traps and other techniques.

- Coexistence with traditional networking protocols (e.g. TC1P/IP) running on the same network.
- M-VIA is a robust full-featured implementation of VIA. It passes the strict Intel conformance tests, ensuring that codes written using M-VIA will be portable.

Table 1 compares the performance of VIA and TCP on two different NICs. The VIA implementation shows significant improvement over TCP on the same network.

Network	Protocol	Latency (μs)	Bandwidth (MB/s)
Packet Engines GNIC II	TCP	59	31
Packet Engines GNIC II	M-VIA	19	60
Tulip Fast Ethernet	TCP	65	11.4
Tulip Fast Ethernet	M-VIA	23	11.9

Table 1: M-VIA Performance against TCP

M-VIA consists of different modules:

- Kernel Agent: Performs privileged operations on behalf of VIPL and assists M-VIA Device Drivers with operations requiring operating system support. It consists of:
  - Connection Manager: Establishes logical point-to-point connections between VIs.
  - Protection Tag Manager: Allocates, deallocates, and validates memory protection tags (Ptags).
  - Registered Memory Manager: Handles the registration of user communication buffers and descriptor buffers.
  - Error Queue Manager: Provides a mechanism for posting asynchronous errors by VIA devices and blocking.
  - Kernel Extensions: Provides functionality required for efficient implementation.
- M-VIA Device Driver: Provides abstraction of VI NIC and the ability to override all of the default functionality of the Kernel Agent layer should allow any natively supported or software emulated VIA device to be supported by M-VIA.
- Device Classes: For logically grouping commodity network interfaces into common categories such as Ethernet, ATM, FDDI, etc. and meant for rapid development through code reuse.
- VI Provider Library: M-VIA contains a single VI Provider Library, which is interoperable with native hardware and software VIA devices developed within the Modular VIA framework.

Is M-VIA a true zero-copy? Lacking hardware support, M-VIA is only zero-copy (no memory-to-memory copy in host) on sends, but requires one copy on receives. The copy is unavoidable for NICs without special hardware support. On x86 platform, M-VIA uses fast trap on the sender to avoid the overhead of a system call. This fast trap achieves transition to the kernel's privilege level without most of the overhead of a system call (e.g. it does not yield the CPU). On the receiver, data is received directly with a low-level hardware interrupt that does all the necessary processing.

5.2 Berkeley VIA [7]

Berkeley VIA implemented a subset of VIA over Myricom's Myrinet M2F (with the LANai 4.x-based NIC) network by emulating it as a VIA-capable hardware through programming its firmware. Doorbell is implemented through firmware polling doorbell locations in NIC memory.

Here is what happens for a send operation (see Figure 11):

Build: To send a message, the application builds its message in a buffer within a registered-memory region and creates a descriptor, also within that region, for the message.

T1: To notify the NIC of the descriptor's existence, the user-level library posts a doorbell to the NIC. At this point, the application may proceed with computation; all further work is done by the NIC.

T2: The NIC moves the received doorbell into a queue of unprocessed doorbells.

T3: When the doorbell reaches the front of the queue, the NIC uses the address contained in the doorbell to locate and transfer the descriptor across the I/O bus to a staging area.

T4: Using the information in the descriptor, it transfers the data itself across the I/O bus to a data staging area.

T5: When the data is available, the NIC places it onto the network using the network address contained in the descriptor.

T6: Once the transfer onto the wire is complete, the NIC updates the status field of the descriptor to signal completion .

Cmpl: At this point, the application can detect completion of the message transmission (Cmpl) by reading the descriptor's status field.

For receive operations and more info. refer to [7]. The achieved performance was comparable to other user-level networks but not better (latency of $26\mu s$ and bandwidth of $360Mbits/s$). Authors suggested methods like "descriptorless message" (using doorbell token to specify the data buffer) and "Merged Descriptors" (combination of a standard VIA descriptor and its data) to improve the performance. They managed to decrease the latency to $17\mu s$ after implementing "descriptorless message".

The website of the project is not available at the time of this writing to find more information about the implementation and the source codes (http://www.millennium.berkeley.edu/via.php3).

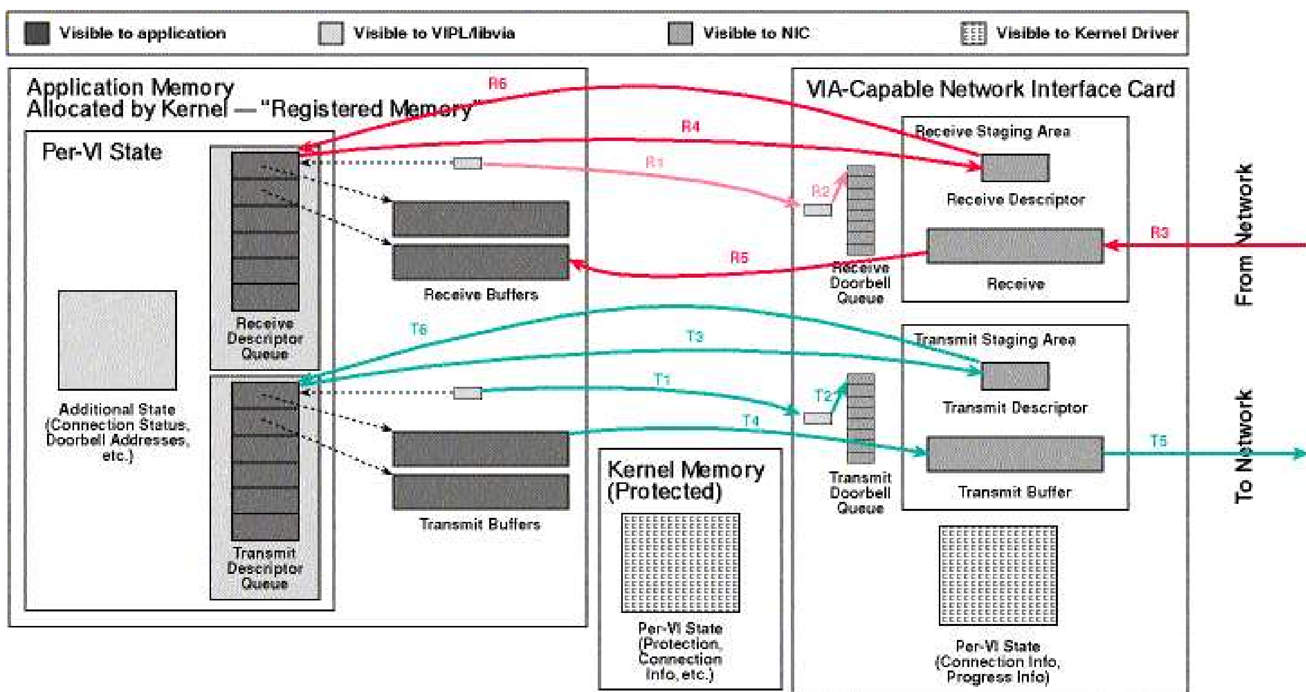

Figure 11: Berkeley VIA Transmit and Receive data flow (from [7])

6 VIA and InfiniBand Architecture

InfiniBand Trade Association (IBTA) was initiated in 1999 and version 1.0 of the InfiniBand Architecture Specification was released in October 2000. By creating a unified fabric, InfiniBand Architecture provides a mechanism to share I/O interconnects among many servers [11]. The architecture is grounded in the fundamental principles of channel-based I/O, the I/O model favored by mainframe computers. InfiniBand channels are created by attaching host channel adapters and target channel adapters through InfiniBand switches. Host channel adapters are I/O engines located within a server. Target channel adapters enable remote storage and network connectivity into the InfiniBand fabric. This interconnect infrastructure is called a "fabric," based on the way input and output connections are constructed between host and targets. All InfiniBand connections are created with InfiniBand links, starting at data rates of 2.5 Gbps and utilizing both copper wire and fiber optics for transmission.

This is a comprehensive specifications including electrical and mechanical configuration of IBA physical media, format of packets, sets of operation, semantics and management interface. It includes many VIA concepts; like RDMA, user-level access to hardware, work queue model, etc.

From the hardware perspective IBA is much more complete while its software spec. does not go as far as VIA. It is not including an API; and defines a set of verbs instead providing abstract description of an IBA NIC, not specifying calling sequence and data types. IBA provides more operations (atomic *Compare & Swap* and *Fetch & Add*) including VIA's four operations (Send/Receive, RDMA-W/R). IBA extends memory protection model of VIA by allowing granularity of memory registration at byte level rather than page level. Its Memory Windows mechanism provides the user with more control over the region of a registered memory exposed for RDMA operations. A work request is IBAs equivalent of VIAs work queue. IBA also provides new transport services (Reliable Datagram, Unreliable Datagram and Raw Datagram services) extra to VIA's three Unreliable, Reliable Delivery and Reliable Reception services.

7 Conclusions and future of VIA

The VI Architecture has largely achieved its goals. Hardware vendors have developed several VI-compliant products that feature high bandwidth and low overhead. These products have clearly demonstrated the portability, OS and network independence of VIA. Also, significant commercial applications developed or adapted to use VIA indirectly (e.g. over Sockets) or through accessing native VIA directly. A few are DB2 EEE, Oracle Database and MS SQL. [2] has shown that native VIA support in MS SQL for application server to database-server com-

munication or among database servers themselves has resulted in over 33% performance improvement comparing to TCP/IP. Also, legacy message passing interfaces like MPI (MVICH) and Sockets are implemented on top of VI.

Users and developers of VIA have found some areas for improvements [1]:

- Increasing limited number of VIs (currently limited to 64000).
- Its connection-oriented based model requires managing more VIs.
- Memory registration: It is hard to implement a legacy message-passing interface on top of VIA because buffer used for Send/Receive must be registered up front.
- Large number of locked pages can lead to performance penalty.
- *Receive Assumption* can be relaxed by using credit-based flow control to hold messages at sending node till receive is posted on the other side (like SOVIA).
- People have been complaining about insufficient standardization. VIPL should have been defined as part of the standard rather than only an example. More detailed spec. for HW interfaces (like doorbell) would have made writing portable VIPL library easier across HW implementations but on the other hand would have limited custom HW/SW support and design approaches (like [6]).

Future VI Developers Forum (VIDF) took on the task of evolving VI architecture and standardizing VIPL API. Version 1.1 of VIPL is defined, including extensions requested by database vendors. It had to be published as VIDF Extensions to VIPL-1.0 because of disagreement between 3 original promoters of VIA. The website of the forum (http://www.vidf.org) is not accessible and we were not able to find more information about its current status.

A new organization called Direct Access Transport Collaborative (DATC) defined a new API that is not derived from VIA specifications. Their first goal was to add kernel API to allow device drivers and OS access VI capabilities (like zero-copy and RDMA).

As the final word, VIA has opened up new opportunities for cluster computing and this trend continues with further releases of VIA or in form of new standards.

References

[1] Greg Regnier, Don Cameron. *The Virtual Interface Architecture*. Intel Press, 2002.

[2] Boris C. Bialek. Virtual Interface Architecture and Microsoft SQL Server 2000. *Dell Corp. White Paper*, January 2001.

[3] *Virtual Interface Architecture Specifications*. Revision 1.0, December 4, 1997.

[4] *Intel Virtual Interface (VI) Architecture Developers Guide*. Revision 1.0, September 9, 1998.

[5] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2): 66-76, March-April 1998.

[6] M. Banikazemi, V. Moorthy, L. Herger, D.K. Panda, B. Abali. Efficient Virtual Interface Architecture (VIA) support for the IBM SP switch-connected NT clusters. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium, IPDPS 2000*, pages 33-42, 1-5 May 2000.

[7] P. Buonadonnaa, A. Geweke and D. Culler. An implementation and analysis of the Virtual Interface Architecture. In *In Proceedings of the Supercomputing (SC 1998)*, pages 7-13, Nov. 1998.

[8] Jin-Soo Kim, Kangho Kim, Sung-In Jung. SOVIA: A user-level Sockets layer over Virtual Interface Architecture. In *Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER.01)*, 2001.

[9] NERSC. *M-VIA version 1.2 documentations*, Sep. 2002. http://www.nersc.gov/research/FTG/via/.

[10] Thorsten von Eicken, Werner Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, 31(11): 61-68, November 1998.

[11] InfiniBand Trade Asociation. *An InfiniBand Technology Overview.* http://www.infinibandta.org/ibta/.