

Chapter 1

Introduction

In December of 1997, Intel, Compaq, and Microsoft, with the help of over 100 contributing industry organizations, published the *Virtual Interface Architecture Specification* to provide an industry standard interface for high performance cluster communications.

Clusters of standard servers are becoming a cost-effective alternative to large mainframe computers, and high performance communication between servers is required for clusters to scale in capacity and performance. The Virtual Interface Architecture (VIA) provides the standard for efficient cluster communications required to achieve a critical mass of cluster middleware and application development.

How does the Virtual Interface Architecture meet these needs? And what are its fundamental underlying concepts? How can applications benefit from the capabilities of VIA? This book is intended as the definitive guide to answer these questions.

What's the Problem?

Clusters are an integral part of the computing resources that power the Internet. Clusters can provide the large scale distributed computing resources necessary for sophisticated e-commerce applications. But scalable cluster computing cannot realize its full potential without high performance intra-cluster communications. Although standard networking hardware promises to deliver ever-faster data rates, existing application interfaces limit its effectiveness. VIA remedies this problem by drastically improving the efficiency of the interface between the application and the network.

Prior to the development of VIA, multiple high speed System Area Networks (SANs) had been developed to solve the problem of high performance communication for scalable clusters. But the interface and operation of these networks were unique and proprietary. This lack of standardization limited the number of applications that were developed for them. These SANs were also relatively expensive due to the fragmentation of the market, and thus lacked the economies of scale needed for cost-effective development and deployment.

Note

When VIA was being defined, the acronym 'SAN' was commonly used to designate a System Area Network. Since then, the storage industry has adopted the acronym for Storage Area Networks.

System designers alternatively employed existing Local Area Network (LAN) technologies for communication within commercial clusters, largely due to their standardization, wide availability, and relative low cost. Unfortunately, these technologies fail to realize the performance potential of the underlying LAN network hardware. This is especially true for scalable clusters, where the highest speed networks are generally deployed, and where the communication requirements are often the result of aggregating the needs of many clients, potentially thousands of them.

This situation leads system designers and software developers into a dilemma. Proprietary interconnects provided the performance required in order to deploy scalable clusters, but the cost and market fragmentation limited the return on investment. Standard LAN technologies offered standardization and lower cost, but the lack of performance limited cluster scalability. VIA fills this gap between standardization and performance.

Clusters fail to exploit the performance of the underlying network due to the magnitude of software overhead in the legacy communication stack. Software overhead limits communication performance by increasing latency and reducing bandwidth. Software overhead also consumes valuable CPU cycles that could otherwise be used by applications.

Physical networks are increasing in speed at a much greater rate than processors. This situation exacerbates the problem of communication overhead. Network speeds are increasing by orders of magnitude, illustrated by the fact that Ethernet speeds have increased from 10, to 100, to 1000 megabits per second, with 10000 megabit Ethernet on the horizon. Meanwhile, CPU speeds are following "Moore's Law," which states that computer processing power doubles every eighteen months.

Figure 1.1 illustrates the relationship between Moore's Law and network hardware speeds, in this case Ethernet, over the last ten years. In 1991, proc-

processor speeds were about 33 Megahertz, while the speed of Ethernet hardware was 10 Megabits per second. Tracking Moore's Law, there is a notable disparity starting in 1995 with the advent of 100 Megabit Ethernet. When considering full-duplex, switched networks, the problem is even more pronounced. (Full duplex networks allow data to flow in both directions simultaneously.) As seen in Figure 1.1, the disparity widens over time, and the last point on the graph reflects CPU speeds and Ethernet rates that are just now emerging.

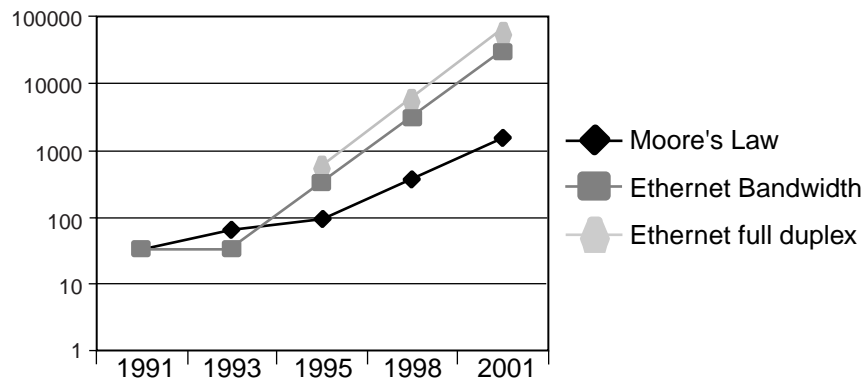


Figure 1.1 Network Speed vs. Processor Speed Over Time

This relationship of network and processor speeds can also be viewed in terms of the effect of software overhead on communication performance. Figure 1.2 compares the processing time for legacy network software stacks (Processing Time) with the average time required for data to move through the physical network media (Ethernet Wire Time). The data takes into account both the increase in processing power due to Moore's Law and the increase in raw Ethernet bandwidth over time. Figure 1.2 shows that increasing the speed of the network media can't significantly increase the communication performance of a given system unless the software overhead is reduced.

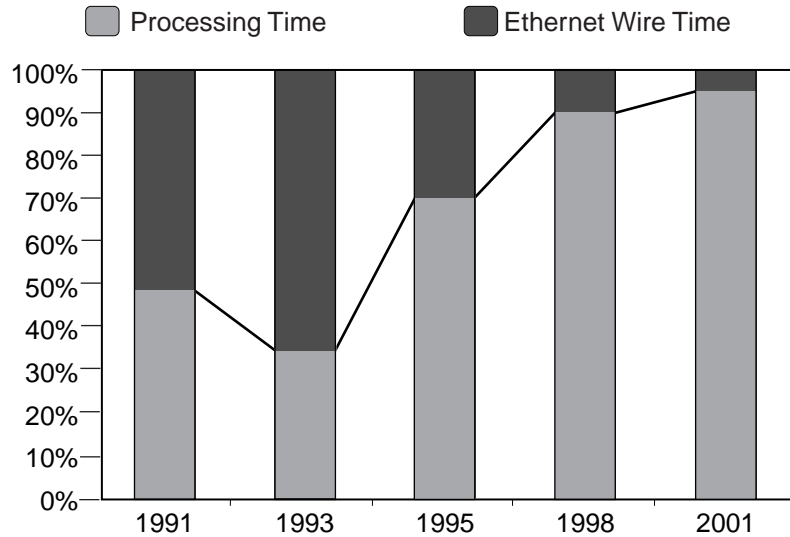


Figure 1.2 Software Time vs. Hardware Time

The achievable performance of a network can be measured in two ways. First, achievable performance may be gauged by measuring the effect of software overhead on *bandwidth* or, the rate at which data can effectively be transferred across a network. Second, the effect of software overhead on *latency* (i.e. the amount of time required to transfer a given message) can be measured. Both of these metrics greatly influence performance in the message-passing programming model.

To determine the amount of time required for the transfer of a single message, the following simple latency equation can be used:

$$t = a + Bn$$

In this equation, (t) is the total time (latency) required to transfer a message; alpha (a) is the fixed, non-overlapped transfer time per message, which is largely dominated by software overhead; beta (B) is the amount of time required for the physical wire to move one byte, which reflects the speed of the network media; and n is the length of the message in bytes.

Figure 1.3 shows the effects of software overhead (alpha) on message latency for a network link that operates at a maximum rate (beta) of one gigabit per second. Latency is an important factor in determining how well parallel and distributed applications scale. Increased latency reduces the number of transactions that can take place between communicating processes in a given amount of time. In Figure 1.3, the effect on latency is shown for varying val-

ues of alpha. (The value of 62 microseconds was chosen based on a recent measured result.¹) The data in Figure 1.3 shows that for small messages, software overhead (alpha) dominates the amount of total latency incurred. It also shows that software overhead has significant impact on total latency throughout the range of message sizes.

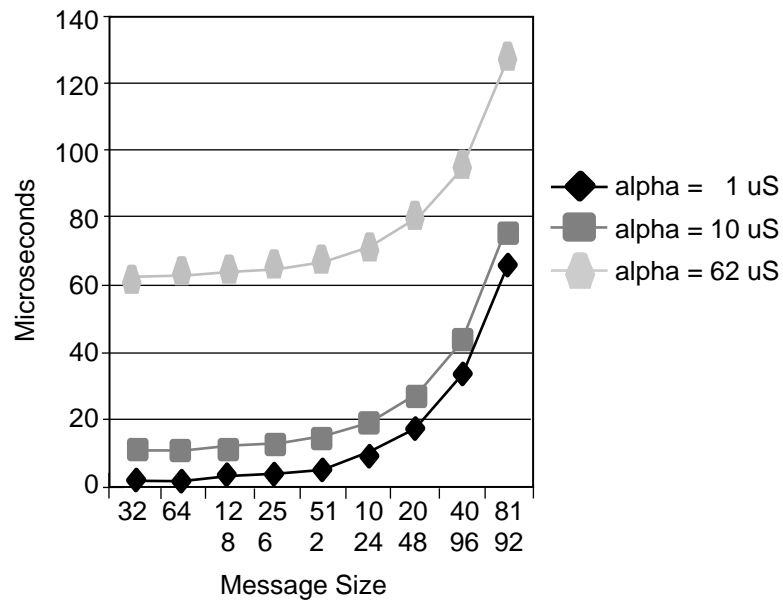


Figure 1.3 Effect of Software Overhead on Latency

Figure 1.4 shows the effects of software overhead (alpha) on the achievable bandwidth for the same one-gigabit network link used in Figure 1.3. Again using the latency equation, it graphs the best possible bandwidth that can be achieved for alpha with values of 1, 10 and 62 microseconds. Figure 1.4 shows that the magnitude of software overhead has a severe impact on the maximum bandwidth that can be achieved. It also shows that even small amounts of overhead can have a major impact on achievable bandwidth for very small messages. For example, only 10 microseconds of software overhead effectively yields less than 50% of the maximum 1 gigabit per second link rate, even for messages of 1024 bytes. In the near future, with network

¹ The measured result showed that a server with a clock rate of 400 Mhz expends 62 microseconds on each packet when running the SPECWeb96 benchmark.

link rates expected to be 10 gigabits per second and beyond, the effect of this software overhead becomes more and more pronounced.

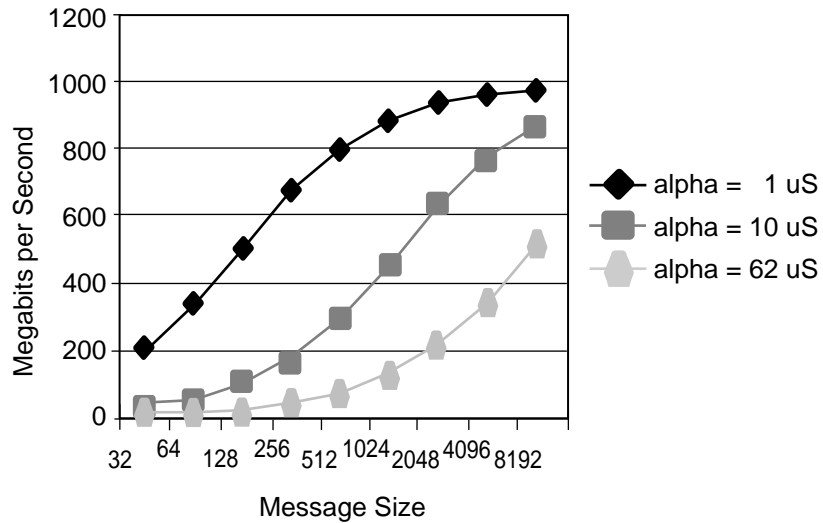


Figure 1.4 Effect of Software Overhead on Bandwidth

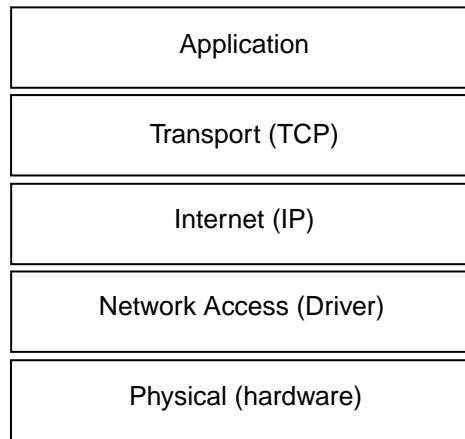
To make matters worse, there is a well-known rule-of-thumb known as the *80/80 rule* of network distribution. This rule states that about 80% of the messages that traverse a typical network are less than or equal to 256 bytes, while about 80% of the data is found in messages greater than or equal to 8192 bytes.

A supporting study by Jonathan Kay and Joseph Pasquale at the University of California, San Diego showed that a network running NFS (Network File System) traffic had a bimodal distribution where the great majority of messages were small, along with a number of very large messages. In their study, 86% of the NFS messages were less than 200 bytes, while 9% of the messages were at least 8192 bytes.

Most small messages relate to control and synchronization information, while the long messages are bulk data exchanges. This 80/80 rule of message distribution shows that small messages occur with the greatest frequency. Since software overhead has the greatest negative performance impact for small messages, their predominance amplifies the importance of minimizing overhead in order to create scalable, high performance clusters and distributed systems.

So what are the components of software overhead? To answer this, we need to understand existing network protocols, and how they are integrated into the operating systems that implement them.

Current operating systems implement network communications as a set of layered drivers within the kernel. This software is commonly referred to as the “network stack.” Figure 1.5 represents a simplified network stack for the TCP/IP protocol. In this figure, the layers between the application and the physical hardware are implemented as kernel software modules, with additional layers possible depending upon the specific operating system.



OM13078

Figure 1.5 The TCP/IP Network Stack

Applications access the network stack by making a system call, often called a trap, into the kernel of the operating system. The kernel networking stack then takes care of multiple issues, such as the protected sharing of the network controller, the multiplexing of incoming messages to multiple processes and logical channels, interrupt handling for synchronization and device control, memory and buffer management, virtual to physical memory address translation, data reliability and integrity, and so on. The result is that thousands of instructions are executed in order to support communications that add to latency and increase the load on the CPU.

Another major contributor to overhead is data copying. Legacy networking stacks require data copies in order to multiplex incoming messages. The stack must store an incoming message in an intermediate buffer, inspect the message to determine the destination buffer, then copy the data to the memory

buffer that was originally specified by the application. Well-tuned network stacks require only one copy on the receiving side to accomplish this multiplexing, but even a single copy uses memory bandwidth, CPU cycles, and perturbs the cache of the executing processor.

Even the act of transmitting a message, where a data copy is not strictly required (but often done anyway), requires substantial processing. Processing is needed to insert protocol headers, build the gather list, lock the virtual memory buffers into physical memory, and translate the buffer addresses from virtual to physical memory addresses in preparation for DMA operations. All of this must be done in the performance critical path, before the message can actually be sent.

The Intel® Enterprise Architecture Lab recently measured the number of processor cycles spent for each packet on a popular Internet benchmark, SPECweb96. Using a popular network OS running on a standard server, the experiment determined that an average of 25,000 processor cycles were executed for each Ethernet packet that was transferred by the networking stack. A processor speed of 400 Megahertz results in about 62 microseconds of overhead per packet. Figure 1.4 shows that with this magnitude of software overhead, the server cannot achieve half the full rate of a one-gigabit network link.

Note

A Packet is a unit of transfer for a network. For Ethernet, a single packet can carry from 64 to 1500 bytes of data.

Other hardware-related factors affect latency and bandwidth. These factors include memory speeds and latencies, and the I/O structure of the computer or device that is attached to the network. These hardware factors have also been considered in the definition of VIA.

How VIA Solves the Problem

VIA reduces the magnitude of software overhead incurred during communication transfers. To do so, VIA bypasses the operating system kernel in the performance critical data path. This approach leads to a number of challenges that must be faced in order to provide the protection and functionality normally furnished by the OS kernel.

VIA provides the illusion of a dedicated network interface to multiple applications simultaneously, thus “virtualizing” the interface. This virtualization of the interface enables user-level message-passing, providing applications with direct network access without OS kernel intervention. The VIA “OS bypass” technique is illustrated in Figure 1.6.

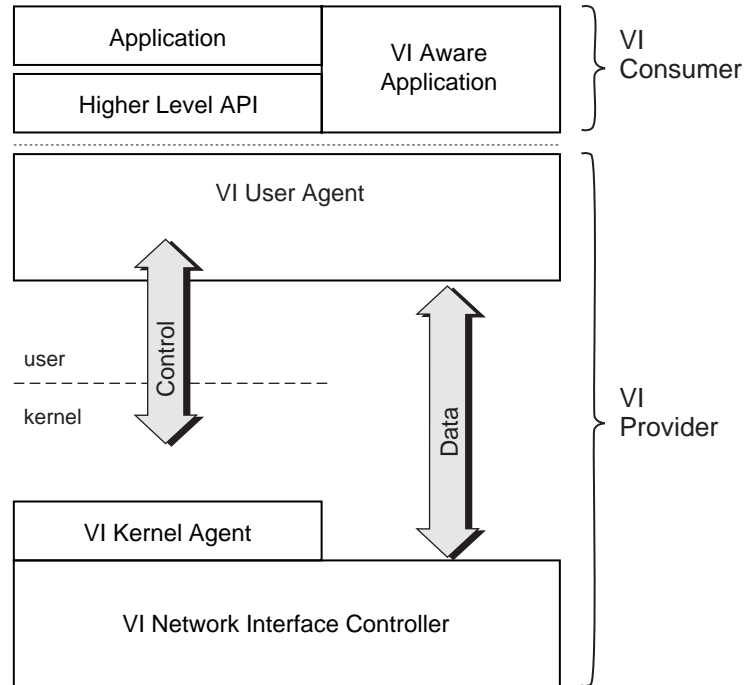


Figure 1.6 VIA Model for OS Bypass

An implementation of the Virtual Interface Architecture, including the hardware and software components, is called a *VI provider*. The users of a VI provider are *VI consumers*².

Two major design principles enable OS bypass. First, VIA partitions the functionality that is normally provided by the kernel between the VI consumer and the VI provider. Second, VIA separates the control path from the performance critical data path.

Functional Partitioning

With VIA, the functionality that is normally supplied by the operating system is partitioned between the VI consumer and the VI provider. For example, the VI consumer allocates and manages its own buffers for use in message-

² In practice, the operating system kernel can be a VI consumer as well as a user-level application, but the VIA specifications did not specify a standardized interface. Some implementations of VIA provided a non-standardized kernel interface for use by in-kernel services and applications.

passing. This capability allows the consumer to optimize its message buffering strategy in a way that best fits each application's particular requirements. In stark contrast, kernel-based network stacks manage message buffers in a generalized way, with memory resources that are shared among all of the users of the network.

The VI consumer is also able to directly notify a Network Interface Controller (NIC) that new messages are to be transmitted, and that new message buffers are available for receiving incoming messages. These functions are normally handled by the operating system in order to control the sharing of the NIC.

The VI provider is directly responsible for a number of functions normally supplied by the operating system. The VI provider manages the protected sharing of the network controller, virtual to physical translation of buffer addresses, and the synchronization of completed work via interrupts. The VI provider also provides a reliable transport service, with the level of reliability depending upon the capabilities of the underlying network.

Separation of Control and Data

In addition to functional partitioning, VIA separates the control and data paths to enable OS bypass. The control path includes the functions that allocate or change the state of a given resource for its use in moving data. For instance, the control path includes operations for the setup and teardown of connections, as well as memory buffer preparation. The data path, which is the performance critical path, includes the actual message-passing operations, such as Send, Receive, and Remote DMA.

Memory registration is an important control path operation defined by VIA. Memory registration prepares a virtual memory region of the VI consumer for subsequent use as a buffer for transmitting and receiving messages. Memory registration includes locking the memory pages into physical memory, and resolving the virtual to physical address translation. These operations are required by all network interfaces that perform direct memory access (DMA) operations to move data between the network and host memory. In contrast, legacy networking stacks normally prepare transmit buffers in-line with the transmit operation. For Receive operations, legacy network stacks use intermediate receive buffers that are subsequently copied to the destination buffer of the application. Thus, memory registration not only moves the expensive locking and translation operations out of the critical path, but also allows for zero-copy Receive operations.

VIA is optimized for the data path. The VIA data path includes message-passing operations for the exchange of information. VIA utilizes an asynchro-

nous queuing mechanism to post work onto the network interface. VIA supports four major message-passing operations; Send, Receive, RDMA-Write, and (optionally) RDMA-Read.

The Send and Receive operations are well-known operations used by most message-passing systems. In order for a message to be exchanged, each *Send* operation must have an associated *Receive* operation at the remote end. The RDMA operations are single-ended, in that only one end of a given connection must issue the operation in order for it to successfully complete. The RDMA operations also contribute to a true zero-copy message-passing paradigm, as will become clear when describing the VIA principles of operation in Chapter 3.

Fundamental Concepts of VIA

In modern computer systems, Virtual Memory (VM) allows application processes to use host memory as if it were its exclusive resource. VM protects the memory of a given process by disallowing access by other processes.

Similarly, the Virtual Interface Architecture (VIA) provides applications with a protected resource, specifically an interface to the network. Both VM and VIA utilize hardware mechanisms together with system software to virtualize a shared physical resource.

VIA “virtualizes” the network interface by allowing processes to create and directly manipulate communication endpoints that are protected across process boundaries. These communication endpoints, simply called Virtual Interfaces (VIs), allow the application to access the network directly without help from the operating system in the performance critical paths.

VIA supports a *message-passing* model of communication, as opposed to load and store memory semantics employed by shared memory systems. VIA is connection oriented; point-to-point connections are constructed by associating pairs of VI endpoints. Communicating processes use connections to exchange information. Each VI employs a simple, asynchronous queuing structure to directly move data in and out of the virtual memory space of an application.

Since each VI is a protected resource, a process can access it directly without intervention from the underlying operating system. This user-level message-passing model results in high performance networking and I/O. By avoiding the overhead normally incurred by the operating system kernel, VI exceeds the performance of conventional network stacks implemented in the kernel.

Many years of private sector and academic research efforts culminated in the creation of VIA. VIA is unique because it defines an industry standard for high performance I/O and networking that is independent of the underlying network media, processor architecture, and operating system. VIA provides portability and compatibility for distributed computing applications, cluster communications, and I/O.

The *Virtual Interface Architecture Specification* describes the components and operations that comprise VIA. The specification also includes an example programming interface and an example hardware design.

In addition, the *Virtual Interface Developer's Guide* defines the Virtual Interface Provider Library (VIPL), an application programming interface for VIA. VIPL provides software developers with a consistent and portable programming interface.

Performance Case Study

Using some simple micro-benchmarks to measure bandwidth and latency, it is possible to compare the base performance of an existing VIA implementation with a TCP/IP software implementation.

The networks that we compare are the GigaNet cLAN[†] versus Gigabit Ethernet. These networks have a comparable peak bandwidth of 1 gigabit per second. The VIA version of the test uses a cLAN network interface that supports the VIA specification and the test application is written using the VIPL programming interface. The TCP/IP version of the test uses standard, off-the-shelf Gigabit Ethernet network interfaces and the test application is written using the standard sockets programming interface. The Gigabit Ethernet configuration runs the TCP/IP protocol stack supplied by the operating system.

The test configuration included two servers directly connected by a network link using no intermediate switch. Each server contained a single 800 Megahertz Intel[®] Pentium[®] III processor. The network interface cards plugged into a 64-bit wide, 33 Megahertz PCI bus.

Figure 1.7 shows a bandwidth comparison of VIA and TCP/IP. This simple one-way bandwidth test used a single connection to pass data between two servers in one direction. The data shows that the reduced overhead of VIA enables the servers to achieve a much higher percentage of the potential bandwidth of the underlying networking hardware. The graph shows that over half of the network bandwidth ($N/2$) is achieved for relatively small messages with VIA, while the TCP/IP networking stack fails to achieve half of the network bandwidth, even for very large messages.

Note $N/2$ is the message size at which 1/2 of the maximum bandwidth for the network medium is achieved. At $N/2$, the transmission time of the message is equal to the sum of the overheads.

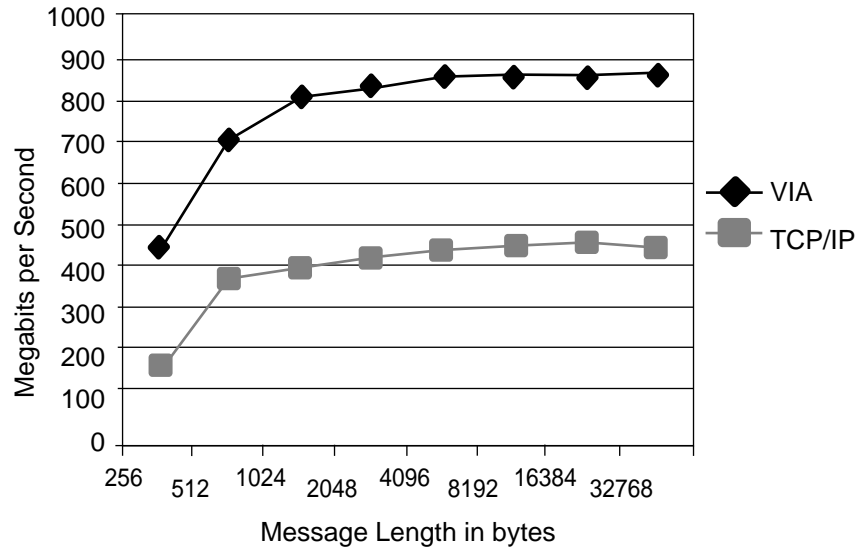


Figure 1.7 Measured One-way Bandwidth

Using the same test configuration, Figure 1.8 compares the message latencies of VIA and TCP/IP. This simple round-trip latency test measures the time it takes for a single message to be transferred from one server to another and back again. The graph shows that the latencies for VIA are significantly lower for VIA than for the TCP/IP stack, largely due to the reduced software overhead.

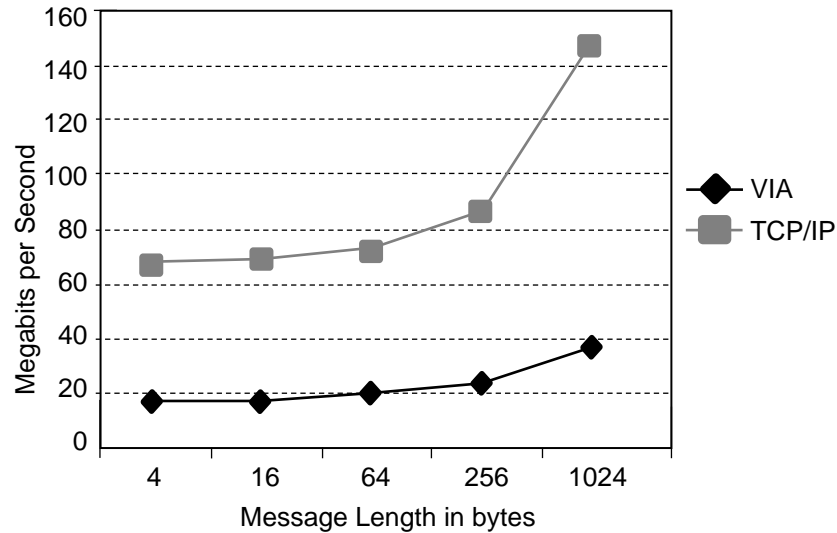


Figure 1.8 Measured Round-trip Latency

The latency determines how many transactions can potentially take place between computers within a certain amount of time. Another simple performance test was conducted to see what the effect of latency is on the number of transactions that can be performed. This simple transaction test sends a request message of 256 bytes from one computer (the client) to another computer (the server), and the server side then replies with a message of increasing lengths. Figure 1.9 compares native VIA with TCP/IP using this simple transaction performance test. It shows that given lower latency, many more transactions can be achieved between two computers, an important factor in determining the scalability of clustered servers.

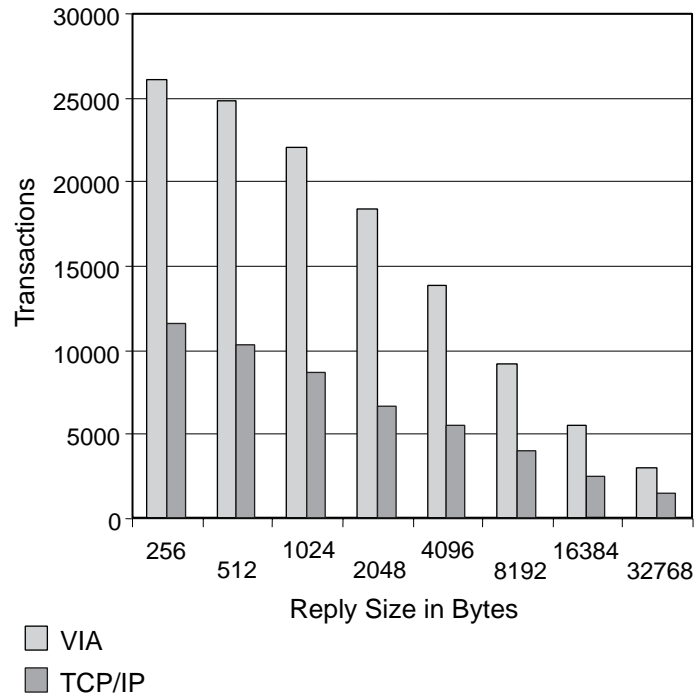


Figure 1.9 Transaction Performance Test

In the near future, the demand for efficient, low overhead communication will increase significantly. With network media moving from 1 to 10 gigabits per second and beyond, the efficiency of the network interface is key to realizing efficient scalable clusters and distributed data centers. VIA provides the efficient interface that enables clusters that scale with advances in interconnect and network data rates.

