Building a High Performance Communication Layer Over Virtual Interface Architecture on Linux Clusters

Jin-Soo Kim
jinsoo@computer.org

Kangho Kim
khk@etri.re.kr

Sung-In Jung
sijung@etri.re.kr

Computer System Research Department
Electronics and Telecommunications Research Institute (ETRI)
Daejeon 305-350, KOREA

ABSTRACT

The Virtual Interface Architecture (VIA) is an industry standard user-level communication architecture for cluster or system area networks. The VIA provides a protected, directly-accessible interface to a network hardware, removing the operating system from the critical communication path. Although the VIA enables low-latency high-bandwidth communication, the application programming interface defined in the VIA specification lacks many high-level features.

In this paper, we develop a high performance communication layer over VIA, named SOVIA (Sockets Over VIA). Our goal is to make the SOVIA layer as efficient as possible so that the performance of native VIA can be delivered to the application, while retaining the portable Sockets semantics. We find that the single-threaded implementation with conditional sender-side buffering is effective in reducing latency. To increase bandwidth, we implement a flow control mechanism similar to the TCP's sliding window protocol. Our flow control mechanism is enhanced further by adding delayed acknowledgments and the ability to combine small messages.

With these optimizations, SOVIA realizes comparable performance to native VIA, showing the latency of $10.5\mu sec$ for 4-byte messages and the peak bandwidth of 814Mbps on Giganet's cLAN. The functional validity of SOVIA is verified by porting FTP (File Transfer Protocol) application over SOVIA.

1. INTRODUCTION

Due to recent advances in commodity microprocessors and high-speed network interfaces, it becomes increasingly popular to use cluster systems as high performance computing platforms. Cluster systems are easy to build and have advantages as scalability and cost-effectiveness, which lead to high performance/price ratio [22, 5].

ICS '01 Sorrento, Italy

In spite of the advent of interconnection networks at gigabit speed, recent studies show that the communication subsystem is the main performance bottleneck in cluster systems [29, 1]. Especially, the use of traditional communication protocols, such as TCP/IP, is reported to fail in delivering raw hardware performance to the end users, due to (1) protocol overhead, (2) context switching overhead, and (3) data copying overhead between the user and kernel space. To address this problem, a number of user-level communication architectures have been proposed that remove the operating system from the critical communication path [28, 21, 23, 6].

The Virtual Interface Architecture (VIA) [8] is an industry effort to standardize user-level communication architectures for cluster or system area networks (SANs). Being strongly influenced by the previous academic research, the VIA specification 1.0 was defined by a group led by Compaq, Intel and Microsoft, and submitted for industry review in 1997. Since then, it has been endorsed by a number of companies.

The VIA specification describes a software interface for fully-protected, user-level access to a network hardware. Network interface cards (NICs) can be designed to support the VIA specification at the hardware level to accelerate the performance further. Examples of NICs with such VIA-aware hardwares include Giganet's cLAN [12], Fujitsu's Synfinity [15], and Compaq's ServerNet-II [7] adapters.

Although the VIA enables low-latency high-bandwidth communication over SANs, the specification only provides a minimal set of primitives mainly for user-level data transfer of a single message. Those primitives lack many high-level features such as a synchronization facility between the sender and receiver, a flow control mechanism, and so on. Thus, we believe the application programming interface (API) defined in the VIA specification is not adequate for direct use by application programmers and it is desirable to define an intermediate communication layer on top of the VIA API. In this paper, we focus on building a high performance communication layer over VIA, named SOVIA (Sockets Over VIA), which has similar semantics as Sockets API. The purpose of this paper is to support the communication model of Sockets API at user-level, without sacrificing the performance of the underlying VIA layer.

The rest of the paper is organized as follows. Section 2 overviews the VIA and its implementations on Linux. It also describes the motivation of our study and related work. Section 3 presents our evaluation methodology including hardware platform and benchmark programs. In section 4, we investigate and evaluate several optimizations to improve the latency and bandwidth of the SOVIA layer. Section 5 shows the resulting performance of FTP (File Transfer Protocol) application ported over SOVIA. Finally, we conclude in section 6.

2. BACKGROUND

2.1 Virtual Interface Architecture (VIA)

Figure 1 depicts the organization of the VIA with four basic components: Virtual Interfaces (VIs), Completion Queues (CQs), VI Providers, and VI Consumers. The VIA provides each consumer process with a protected, directly-accessible interface to a network hardware called Virtual Interface (VI). Each VI represents a communication end-point and a pair of connected VIs support bi-directional, point-to-point data transfer. 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, which represents the user of a VI.

A VI consists of a pair of Work Queues: a send queue and a receive queue. VI Consumers post requests, in the form of *descriptors*, on the Work Queues to send or receive data. A descriptor is a memory structure that contains all of the information that the VI Provider needs to process the request, such as pointers to data buffers. Each Work Queue has an associated *doorbell* that is used to notify the network adapter that a new descriptor has been posted to a Work Queue. Typically, the doorbell is directly implemented by the adapter and requires no kernel intervention to operate.

When the processing of a request completes, the network adapter marks a "*done*" bit in the status field of the corresponding descriptor. The completed descriptors should be explicitly removed from the Work Queue by the VI Consumer itself. They can be discovered either by polling the head of the Work Queue, or by using a blocking call in which the calling process is signaled upon the completion of a descriptor. Alternatively, a user-defined callback function (a *notify function*) may be executed when a descriptor completes. A Completion Queue (CQ) allows a VI Consumer to coalesce notification of descriptor completions from multiple Work Queues in a single location. When a VI is created, each Work Queue can be associated with a CQ. Once this association is established, notification of completed requests for the Work Queue is automatically directed to the CQ.

All the memory regions used for communication (including descriptors and data buffers) should be registered prior to submitting the request, so that the regions are pinned into the physical memory during data transfer. This is because the VIA allows the network adapter to read and write data directly from and to parts of the user address space, thus enabling the zero-copy protocol. When a memory region is not needed any more for communication, it should be explicitly deregistered, whereupon the pages are released and made available for swapping out. The registered memory

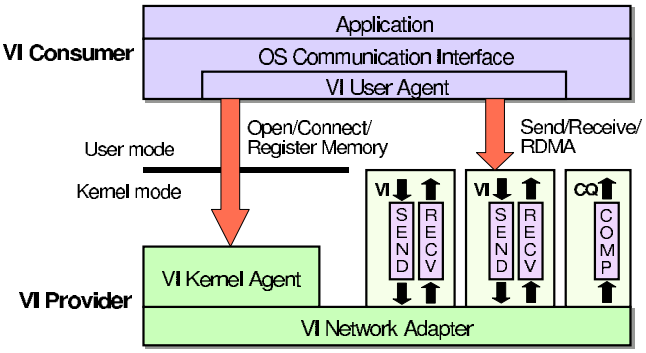


Figure 1: The organization of the Virtual Interface Architecture

regions are protected from access by other processes using unique IDs called Protection Tags that are associated both with VIs and with memory regions.

The VIA specification provides two types of data transfer models: (1) a traditional send/receive messaging model, and (2) the Remote Direct Memory Access (RDMA) model. The RDMA model differs from the send/receive messaging model in that the initiator of the data transfer specifies the locations of both source and destination buffer. Such RDMA operations provide remote memory access without receiver intervention. There are two types of RDMA operations, RDMA Write and RDMA Read, where the support for RDMA Read is optional.

The VIA supports three levels of communication reliability at the NIC level: Unreliable Delivery, Reliable Delivery and Reliable Reception. All VI NICs are required to support the Unreliable Delivery level, in which a send or RDMA Write request will cause at most one receive descriptor to be consumed. Both Reliable Delivery and Reliable Reception guarantee that data sent is received uncorrupted, only once, and in the order that it was sent. Reliable Reception is the highest level of reliability, and differs from Reliable Delivery in that a descriptor can not be marked complete until data has been transferred into the memory at the remote endpoint.

2.2 VIA Implementations on Linux

Table 1 compares three representative VIA implementations available for Linux platforms. M-VIA (Modular VIA) [3], developed by NERSC (National Energy Research Scientific Computing), aims at providing a modular implementation that can be used for various types of NICs including Fast Ethernet and Gigabit Ethernet adapters. For legacy Fast Ethernet cards, M-VIA emulates the VIA specification by software in an intermediate driver layered on top of the standard network driver. Berkeley VIA [4] implementation supports the VIA specification on Myrinet [2] by modifying its firmware. Finally, Giganet Inc. (now Emulex Corp.) has developed a proprietary, VIA-aware network adapter called cLAN [12].

Note that only the Giganet cLAN supports RDMA Write and Reliable Delivery, but it neither utilizes Protection Tags nor provides notify functions.

Table 1: A Comparison of VIA Implementations

	M-VIA	Berkeley VIA	Giganet cLAN
Supported NIC	Fast Ethernet (DEC, Intel) Gigabit Ethernet (Packet Engines, Intel, Syskonnect)	Myrinet	Giganet cLAN1000
Supported OS	Linux 2.2.x	Linux 2.2.x, Windows NT, Solaris 2.6	Linux 2.2.x, Windows NT
RDMA:			
Write	No	No	Yes
Read†	No	No	No
Reliability:			
Unreliable Delivery	Yes	Yes	Yes
Reliable Delivery†	No	No	Yes
Reliable Reception†	No	No	No
Protection Tags	Yes	No	No
Notify Functions	Yes	No	No
Completion Queues	Yes	No	Yes
MaxTransferSize	32,768 bytes	102,400 bytes	65,486 bytes

† represents the optional features of the VIA specification.

2.3 Motivation

For application programmers, the VIA specification provides a set of standardized API in the form of a user-level library called VIPL (VI Provider Library). Although the VIPL can be directly used to develop applications, it is desirable to build another communication layer on top of VIPL for the following reasons:

- In order to support the zero-copy protocol, the VIA requires that the receiver should be ready before the sender initiates its operation. This means an application at the receiving end should pre-post a descriptor to the receive queue before the sender requests a data transfer. Otherwise, the transfer can be lost and the error is not even detected by the receiving side on an unreliable VI. As the number of descriptors that can be posted in the receive queue is finite, a high-level synchronization protocol needs to be implemented between the sender and receiver to satisfy this *pre-posting constraint*.

- The VIPL provides only the basic primitives required for exchanging a single message. To increase bandwidth further, it is necessary to implement a flow control mechanism, which allows multiple messages in transit simultaneously.

- In the traditional communication architecture, incoming messages are managed by an interrupt handler inside the kernel. In the VIA, however, the user application itself should extract the completed descriptor and then post a new one for each incoming message that is delivered asynchronously.

- The programmer needs to manage a large number of descriptors and data buffers for the VIPL. As described in section 2.1, those memory regions should be locked in the physical memory through the registration while they are used for communication. In addition, as most implementations require descriptors be aligned on a certain memory boundary (i.e. 64 bytes), the programmer should pay attention to the allocation of memory spaces for descriptors.

- In the VIPL, there is a limitation on the maximum data size that can be carried on a single message (refer to MaxTransferSize in table 1). Therefore, a long message needs to be decomposed into a sequence of small messages and they should be assembled in the receiving end.

Because it is unreasonable to implement aforementioned features on each user application, programmers will typically get communication services through the other portable layers. Adding a new software layer introduces an overhead inevitably. Therefore, the problem is to make such an intermediate layer efficient so that the low-latency and high-bandwidth characteristics of native VIA can be delivered to the application.

In this paper, we develop SOVIA (Sockets Over VIA), a high performance communication layer over VIA. As the name implies, SOVIA is a subset of Sockets API and offers a simple and general communication service based on blocking `send()` and `recv()`. More specifically, this paper focuses on various optimizations for SOVIA, quantitatively analyzing the effectiveness of each optimization in terms of the latency and bandwidth seen by the application.

The SOVIA layer also can be used to accelerate the existing Sockets-based applications on the VIA with a reasonable effort. It is even possible to stack up another layer over SOVIA, allowing user applications to transparently benefit from the VIA without any source-level modifications. For example, the RPC layer can be easily extended to use SOVIA as one of the lower-level transports, in which case RPC applications need not be modified to take advantage of the VIA.

2.4 Related Work

Two possible candidates that can be used as a upper layer of VIPL are MPI (Message Passing Interface) [25] and Berkeley Sockets API [16], considering their wide spread use and acceptance in the cluster environment.

There exist several MPI implementations over VIA, such as MVICH [18], ParMa^2 [9], and MPI/Pro [10]. MVICH and ParMa^2 are the modifications of open MPI implementations, MPICH [13] and LAM/MPI [19] respectively, while MPI/Pro is a commercial product developed by MPI Software Technology Inc. Ong and Farrell [20] have compared the performance of MPICH, LAM/MPI and MVICH on Gigabit Ethernet networks. They show the overhead incurred in TCP/IP protocol stack is still high and the performance of MVICH (over M-VIA) is much superior to that of TCP/IP-based MPICH or LAM/MPI.

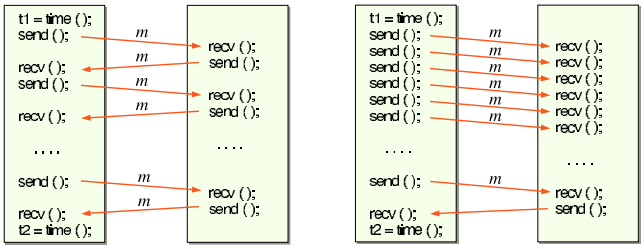
Although MPI is important for high-performance parallel computing, we consider Sockets API in this paper because it provides simpler, yet more versatile communication interface compared to MPI. Sockets-based applications mostly use blocking `send()` and `recv()`[1], while MPI has blocking and non-blocking message passing primitives with four different communication modes such as standard, synchronous, buffered, and ready. Due to the simplicity in Sockets API, it is easy to optimize the communication performance, imposing less overhead on user applications.

Fast Sockets [24] was the first attempt to support Sockets API over a lightweight user-level protocol, Active Messages (AM). However, because AM is connectionless and it has a unique message passing model in which a packet contains the name of a handler function, Fast Sockets can not be directly used for the VIA. Recently, Itoh et al. [14] have presented VIsocket, a Sockets layer implemented between the STREAM module and the (kernel) VIPL in Solaris, but the design and performance details of VIsocket have not been published yet. Giganet Inc. supplies the LANE (LAN Emulation) [11] driver for its cLAN adapters, which emulates IP layer over VIA. However, VIsocket or LANE is the in-kernel implementation and still has the overhead of context switching and data copying between the user and kernel space to exchange data.

Our approach is conceptually similar to the WSDP (Windows Sockets Direct Path) [17] technology developed by Microsoft for Windows NT platforms. The WSDP enables Windows Sockets (WS) applications that use TCP/IP to obtain the performance benefits of SANs without application modifications, by switching to the SAN WS Provider below the Winsock library. Note that in Unix-flavored systems, it is very difficult to emulate Sockets API completely at user-level, because Sockets-related data structures are kept inside the kernel and may be shared with child processes.

Speight et al. [26] examine the cost of memory registration, data copying, and polling vs. blocking receives in the context of using VIPL to implement `MPI_Send()` and `MPI_Recv()`. In addition to those basic optimizations, this paper investigates an efficient message handling strategy and a flow control mechanism to maximize the performance. Moreover, we present the performance of real application executed over the SOVIA layer.

[1]Sockets API also supports non-blocking `send()` and `recv()`, but we do not consider them in this paper.



(a) $l(m) = (t2 - t1)/2n$ (b) $B(m) = mn/(t2 - t1)$

Figure 2: The structure of microbenchmarks to measure the (a) latency and (b) bandwidth. m is the packet size and n denotes the number of packets sent and received during the measurement.

3. METHODOLOGY

3.1 Evaluation Platform

The hardware platform used for performance evaluation is two Linux servers based on Intel L440GX+ motherboards running Linux kernel 2.2.16. Each server has dual Pentium III-500MHz microprocessors with 512KB of L2 cache, 256 MB of main memory, and an on-board Intel EtherExpress 10/100 Fast Ethernet adapter. Additionally, Giganet's cLAN1000 network adapter has been attached to the 32-bit 33MHz PCI slot of each server. The cLAN1000 adapters are connected in a back-to-back topology without an intermediate switch.

The SOVIA layer has been evaluated on two different VIA implementations: M-VIA version 1.0 on Fast Ethernet adapter and Giganet's cLAN version 1.1.1. Note that M-VIA does not achieve the zero-copy protocol on Fast Ethernet due to the lack of hardware support for doorbells [3]. The TCP performance on cLAN is measured using the LANE driver supplied by Giganet. Unless otherwise stated, all the experiments were performed on a uniprocessor kernel using only one processor.

3.2 Microbenchmarks

To evaluate the impact of each optimization on the application's performance, we use microbenchmarks which measure the latency and bandwidth. The latency, $l(m)$, is the time needed to send an m-byte message from a sender to a receiver. It is measured by a half of average round-trip time in a series of ping-pong tests, as shown in figure 2(a). If the message size is not explicitly specified, we assume the time to send one word, i.e. $l(4)$, is meant by the term "latency". The (unidirectional) bandwidth, $B(m)$, denotes how many m-byte messages can be transferred in one way. To measure $B(m)$, the sender issues a long rapid sequence of `send()`'s for a given time and waits until an acknowledgment message arrives from the receiver (cf. figure 2(b)).

Similar benchmarks are implemented using the VIPL as well, to measure the latency and bandwidth of native VIA.

4. DESIGN ISSUES AND EVALUATIONS

In this section, we investigate and evaluate several design issues related to the implementation of SOVIA. Our goal is to make the SOVIA layer as efficient as possible so that the performance of SOVIA approaches to that of native VIA in terms of the latency and bandwidth, while retaining the Sockets semantics. We classify our optimizations into two categories; one for minimizing latency and the other for maximizing bandwidth.

4.1 Minimizing Latency

The SOVIA layer should be designed carefully to make the application fully exploit the native VIA's low-latency characteristics. We consider the following three optimizations for minimizing latency.

The synchronization between `send()` and `recv()`. The VIPL has its own primitives to send and receive a message such as `VipSendPost()` and `VipRecvPost()`. However, `send()` and `recv()` can not be directly replaced with the VIPL's messaging primitives, due to the Sockets semantics and the VIA's pre-posting constraint. A message sent by `send()` can be lost if `recv()` is not yet called on the destination node. Therefore, there should be a synchronization mechanism with which the sender guarantees that at least one descriptor is available on the receive queue (RQ) of the destination VI.

There are two methods to achieve this synchronization. In three-way handshaking shown in figure 3(a), the sender first sends a REQ packet to the receiver. When ready, the receiver posts two descriptors on its RQ, one for data and the other for next REQ, and then replies to the sender with an ACK packet. Upon the receipt of the ACK, the sender transmits data. Because the DATA packet is transferred after the destination node calls `recv()`, the receiver always knows the target buffer address. Therefore, it is possible for the NIC to move the incoming data directly to the user space. The three-way handshaking has, however, the overhead of exchanging REQ and ACK packets before each data transfer, and this overhead has a substantial impact on latency especially for small messages (up to almost three times of the VIA's latency).

Instead, SOVIA uses two-way handshaking illustrated in figure 3(b), where DATA packets are immediately sent to the receiver. In this case, the DATA packet may arrive before the application calls `recv()` on the destination node, hence the receiver is required to buffer the incoming data temporarily. Such an intermediate buffering at the receiver side also increases latency, but the overhead is far less than the case of three-way handshaking.

Message handling strategies. When a packet arrives at a node, the corresponding descriptor should be extracted from a queue and an appropriate action needs to be taken. Normally, the arrival of a packet is not automatically notified to the user application. However, by registering a notify function in advance, it is possible to run a specific code upon the completion of a descriptor. Under the two-way handshaking, the notify function pre-posts a descriptor, sends an ACK, and then wakes up the application thread if it has been suspended on `recv()` (cf. figure 4(a)). In M-VIA,

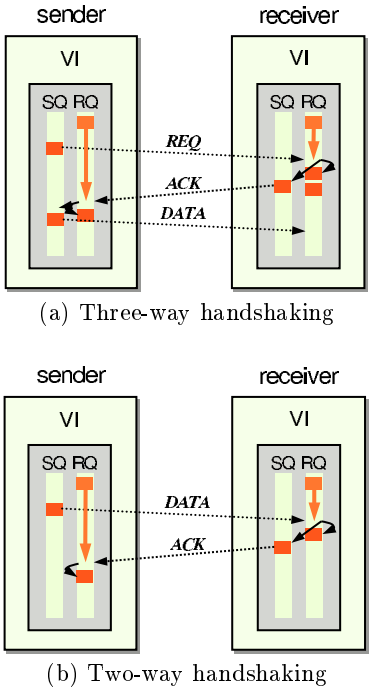


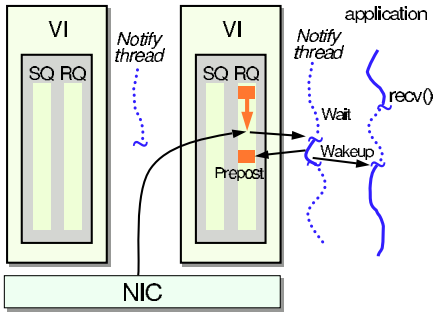
Figure 3: Synchronizing sender and receiver to satisfy the pre-posting constraint.

the notify function is implemented by a *notify thread*, which monitors the associated queue. A separate notify thread is created for each Work Queue.

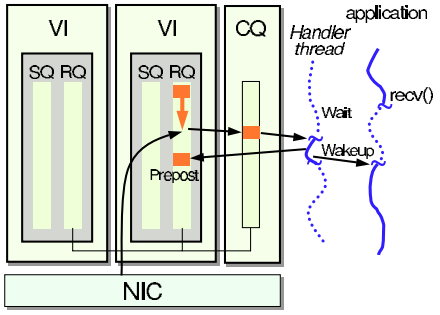
Unfortunately, Giganet's cLAN does not support the notify functions as yet. We can still emulate the notify functions by creating a dedicated handler thread manually, as shown in figure 4(b). To avoid creating multiple handler threads, we associate Receive Queues (RQs) with a Completion Queue (CQ) and let the handler thread check only the CQ.

Figure 5 compares the observed latency of each SOVIA implementation with that of native VIA and TCP. An implementation of SOVIA using the notify functions is labeled as SOVIA_NOTIFY (for M-VIA only), while one using the handler thread is labeled as SOVIA_HANDLER. First of all, we can see that native VIA outperforms TCP as expected; native VIA shows the latency of 43.3μsec (M-VIA) and 8.3μsec (cLAN) for 4-byte messages, while TCP shows 75.6μsec (M-VIA) and 54.9μsec (cLAN) for the same condition. In particular, we can notice that the TCP latency on cLAN, measured with Giganet's LANE driver, is 6.5 times higher than the native VIA's latency.

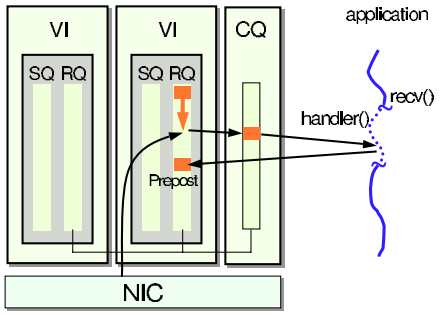
Because both SOVIA_NOTIFY and SOVIA_HANDLER operate basically in the same way, they show similar latency on M-VIA. The minimum latency of SOVIA_NOTIFY and SOVIA_HANDLER is 88.5μsec and 83.3μsec respectively, which is slightly higher than that of TCP. Although the latency of SOVIA_HANDLER has been improved on cLAN to 29.9μsec for 4-byte messages, there still remains a significant gap in latency between SOVIA_HANDLER and native VIA.



(a) SOVIA_NOTIFY

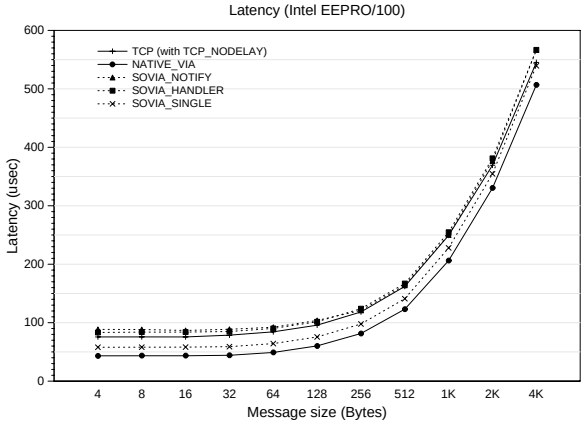


(b) SOVIA_HANDLER

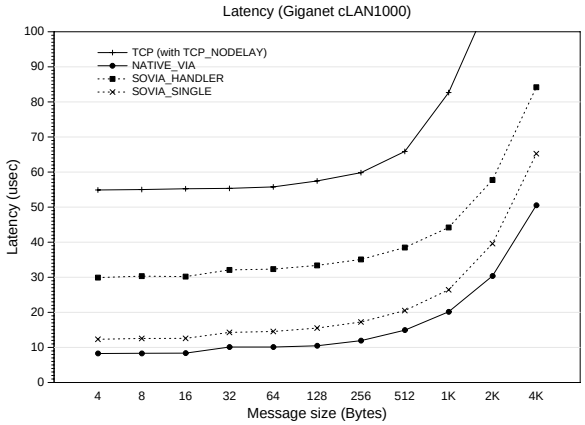


(c) SOVIA_SINGLE

Figure 4: Message handling strategies.



(a) M-VIA



(b) cLAN

Figure 5: Impact of different message handling strategies on latency

We find the reason of poor performance in SOVIA_HANDLER is due to the high thread synchronization cost in Linux kernel. When we use the separate handler thread, the application thread may need to wait for a signal from the handler thread and any data shared by theses two threads should be protected using mutexes. The cost of this synchronization is expensive, sometimes up to tens of microseconds, and more importantly, it is on the critical communication path. Considering that the latency of native VIA is less than $10\mu sec$ on cLAN, the high synchronization cost is the main source that increases latency.

To eliminate the thread synchronization cost, we have developed a single-threaded version of SOVIA (labeled as SOVIA_SINGLE), where the application thread itself is in charge of the handler thread's functionality (cf. figure 4(c)). In SOVIA_SINGLE, the incoming messages are handled by the application thread when it calls communication-involved functions, such as `send()` or `recv()`. Communication may be delayed while the application thread at the receiving node is busy for computation, but by pre-posting multiple descriptors in advance (described in section 4.2), it is possible to overlap the communication with the computation.

The performance of SOVIA_SINGLE is also plotted in figure 5. We can see the latency is improved notably both on M-VIA and on cLAN. The single-threaded implementation of SOVIA exhibits the minimum latency of $57.6\mu sec$ and $12.3\mu sec$ for M-VIA and cLAN, respectively. From these results, we can roughly calculate that the overhead due to multithreading in SOVIA_HANDLER is around $20\mu sec$.

Memory registration vs. copying. Each data transfer experiences one memory registration on the sender side, because a set of buffers used in the receiver can be pre-registered. The memory registration is the key element in the VIA which enables the zero-copy protocol. However, registering a memory region is a relatively expensive operation for small messages. The minimum cost for memory registration and deregistration is about 3 – $4\mu sec$ for M-VIA and cLAN, and the cost increases with the data size,

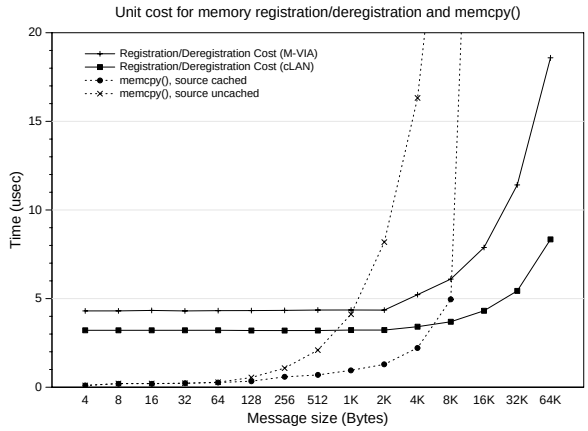


Figure 6: Memory registration vs. copying

as depicted in figure 6.

For comparison, we also plot the cost of `memcpy()` as dotted lines in figure 6. Although the actual time spent on `memcpy()` depends on whether the source is always the same (labeled as *cached*) or not (labeled as *uncached*) due to the caching effect[2], we can see that memory copying has less overhead than the registration when the data size is smaller than 1KB – 4KB. Therefore, we can consider the use of sender-side buffering, where the user data is simply copied into a pre-registered buffer before the descriptor is posted in a send queue.

Table 2 and 3 show the changes in latency on M-VIA and on cLAN respectively, by the use of sender-side buffering (labeled as SOVIA_COPY). The previous implementation, where the source data is always registered, is labeled as SOVIA_REG. As can be seen in the tables, the sender-side buffering is effective in reducing latency for small messages, but it is not adequate for large messages since the cost of memory copying increases rapidly.

Note that the difference in latency between SOVIA_REG and SOVIA_COPY on M-VIA is actually greater than the memory registration/deregistration cost shown in figure 6. This is because the cost of memory registration and deregistration on M-VIA depends not only on the data size, but also on the size of memory regions that are already registered. For example, the cost for registering and deregistering 4-byte data increases from 4.3μsec to 10.1μsec, if the operation is performed when 2MB of data remain registered in the system. SOVIA pre-registers several descriptors and data buffers for each socket connection, and as a result, SOVIA_REG on M-VIA performs worse in real situations than it is expected. The VIPL on cLAN, however, does not show this characteristics, and its memory registration cost depends only on the data size.

We take advantage of both implementations by using a hybrid approach (labeled as SOVIA_HYBRID); data is copied into the buffer if the requested size is less than or equal to 2KB, otherwise it is registered. In this way, we can reduce the latency of SOVIA as low as 45.0μsec for M-VIA and 10.5μsec for cLAN, adding only 2μsec of overhead to the native VIA's latency. The measurement results for SOVIA_HYBRID are also presented in table 2 and 3. we can see SOVIA_HYBRID effectively follows the minimum of SOVIA_REG and SOVIA_COPY in most cases within measurement error.

The number of operations for memory registration and deregistration, required for sending messages larger than 2KB, can be reduced further by the use of *memory registration caching*, as suggested in [18]. With the memory registration caching, the deregistration of a memory region is delayed expecting that the same region is used again in the near future. However, this demands a modification to the default `malloc()` and `free()` routines, because users may free a memory region without notice whose registration is still cached.

[2]The destination of `memcpy()` was fixed during this measurement.

Table 2: Changes in latency on M-VIA

message size (Bytes)	Native VIA (μsec)	SOVIA_ REG (μsec)	SOVIA_ COPY (μsec)	SOVIA_ HYBRID (μsec)
4	43.3	57.6	45.1	45.0
8	43.4	57.9	45.4	45.4
16	43.4	58.0	45.4	45.4
32	44.2	58.8	46.0	45.9
64	49.1	63.9	51.2	51.2
128	60.3	75.5	62.9	62.9
256	81.5	97.5	84.7	84.7
512	123.1	140.7	127.7	127.8
1K	206.3	227.9	213.5	213.9
2K	330.3	354.5	339.3	339.6
4K	506.8	539.3	526.3	539.4
8K	859.0	912.7	915.0	912.4
16K	1545.3	1632.5	1660.7	1632.2
32K	2910.2	3053.7	3135.6	3053.4

Table 3: Changes in latency on cLAN

message size (Bytes)	Native VIA (μsec)	SOVIA_ REG (μsec)	SOVIA_ COPY (μsec)	SOVIA_ HYBRID (μsec)
4	8.3	12.3	10.5	10.5
8	8.3	12.6	10.6	10.7
16	8.4	12.6	10.8	10.8
32	10.1	14.3	12.5	12.5
64	10.1	14.5	12.8	12.8
128	10.5	15.5	14.0	14.0
256	11.9	17.3	16.0	16.0
512	14.9	20.5	19.7	19.7
1K	20.2	26.4	26.5	26.5
2K	30.4	39.5	40.7	40.8
4K	50.5	65.3	74.2	65.0
8K	90.6	123.4	160.1	123.0
16K	169.9	240.7	307.7	240.3
32K	329.0	461.1	603.2	460.6

4.2 Maximizing Bandwidth

The optimizations discussed in the previous section primarily focuses on the efficient delivery of a single message. In this section, we examine a sophisticated flow control mechanism to maximize the bandwidth of SOVIA.

Flow control. Under the two-way handshaking shown in figure 3(b), the sender should wait for an ACK before requesting the next data transfer. The ACK packet informs that the receiver has pre-posted a descriptor to the corresponding RQ and is ready to receive another DATA. As a result, there is at most a single outstanding DATA packet per VI at any given time, under-utilizing the physical resource. TCP uses a flow control algorithm called a *sliding window protocol* [27], which allows the sender to transmit multiple packets before it stops and waits for an acknowledgment. This leads to higher bandwidth since the flow of packets can be pipelined.

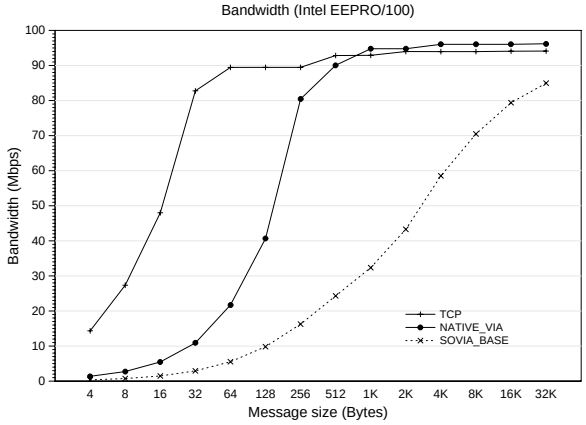
Figure 7 compares the observed bandwidth of TCP and native VIA with that of the SOVIA implementation which has no flow control mechanism. In the figure, SOVIA_BASE represents the single-threaded implementation with conditional sender-side buffering, which has minimized latency in the previous section. Comparing SOVIA_BASE to native VIA, it is apparent that there is much room to improve the bandwidth of SOVIA. When we look at the results of TCP and native VIA, we can see native VIA shows the higher bandwidth than TCP when the message size is larger than 256 bytes[3]. Once again, notice the inefficiency in Giganet's LANE driver; the bandwidth of TCP for 32KB messages is limited to about 450Mbps on cLAN, attaining only 55% of native VIA's performance (815Mbps).

To increase bandwidth, SOVIA supports a flow control mechanism similar to the TCP's sliding window protocol by extending the two-way handshaking. Our implementation of SOVIA also has the notion of *window size* w, which denotes the maximum number of messages the sender is allowed to transmit without waiting for an acknowledgment. Initially, the receiver pre-posts w descriptors to RQ. Whenever the sender transmits a DATA, it decreases w meaning that one of the pre-posted descriptors on the receiving end has been consumed. If w reaches zero, there is no available descriptors on the receiver and the further transmission is on hold until w becomes a positive number. The window size w is increased by one when an ACK is delivered to the sender acknowledging one of the previous DATA packets.

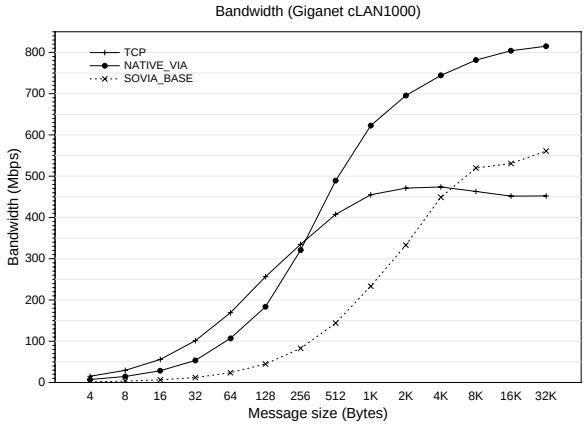
Delayed acknowledgments and piggybacking. Normally, a single ACK is generated on the receiving end for each DATA packet. The number of ACK packets can be reduced by combining several acknowledgments together directed to the same sender. In TCP, the receiver delays the ACK, typically up to $200msec$, hoping to have data going in the same direction as the ACK. If there is data to send, all the delayed ACKs are *piggybacked*, i.e. sent along with the data.

SOVIA also takes advantage of delayed acknowledgments and piggybacking, by using an adaptation of TCP's algorithms. In SOVIA, the receiver simply counts the number of ACK packets (d) that are being delayed, instead of using a timer. If d becomes greater than the predefined threshold t, where $t < w$, an ACK is delivered to the sender piggybacking d. This will increase the sender's window size w by d. On the other hand, when the receiver has a DATA packet for the same direction before d reaches t, delayed acknowledgments are piggybacked with the DATA. We make use of the 32-bit *Immediate Data* field of the descriptor to record the packet type and the number of piggybacked acknowledgments. In the worst case, w DATA packets and (w/t) ACK packets can be simultaneously delivered to a node. Therefore, under our flow control mechanism, each node should pre-post at least $w + (w/t)$ descriptors into the receive queue.

Figure 8 shows the changes in bandwidth as we apply each optimization to SOVIA. SOVIA_FLOWCNTL denotes the addition of our flow control mechanism to SOVIA_BASE, and SOVIA_DACKS adds delayed acknowledgments and piggybacking to SOVIA_FLOWCNTL. We can see that the bandwidth of SOVIA_DACKS is significantly improved compared to SOVIA_BASE for both M-VIA and cLAN. In particular, SO-



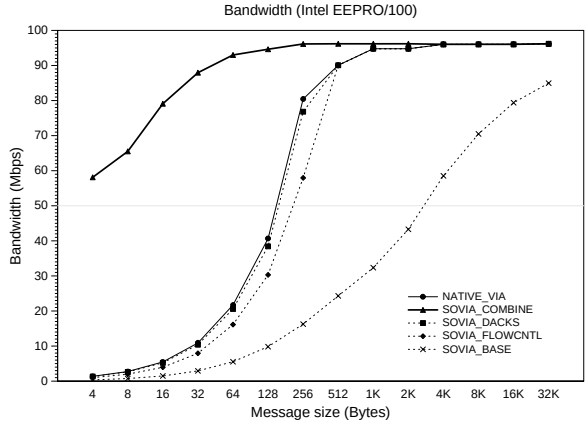
(a) M-VIA



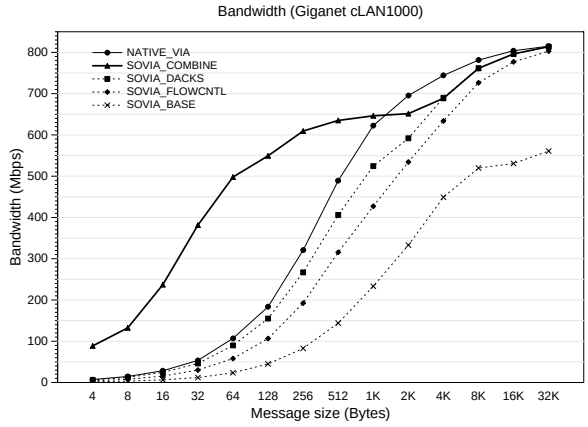
(b) cLAN

Figure 7: A comparision of bandwidth for TCP, native VIA, and SOVIA with no flow control mechanism.

[3]The socket buffer size of TCP is increased to the maximum (128KB) during the measurement of bandwidth. We have used the default MTU size for TCP, which is 1500 bytes on Fast Ethernet and 9000 bytes on cLAN.



(a) M-VIA



(b) cLAN

Figure 8: Impact of various optimizations on bandwidth

VIA_DACKS has roughly the same bandwidth as native VIA on M-VIA. On a faster network such as cLAN, the cost of the SOVIA layer is almost amortized if the message size is larger than 8KB. The peak bandwidth of SOVIA is 96Mbps on M-VIA and 814Mbps on cLAN for 32KB messages.

In this experiment, we have used the window size of 32 ($w = 32$) and the threshold is set to 16 ($t = 16$). Note that SOVIA_BASE can be emulated on SOVIA_DACKS by setting $w = 1$ and $t = 1$. Similarly, SOVIA_FLOWCNTL is equivalent to SOVIA_DACKS with $w > 1$ and $t = 1$.

Combining small messages. In figure 7, we can notice that TCP shows higher bandwidth than native VIA for small messages less than 256 – 512 bytes. This is the effect of the *Nagle algorithm* [27] enabled in TCP by default. The 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 the TCP can send a full-sized message. The Nagle algorithm is originally developed as a way to avoid congestion on wide area networks, but has a side-effect that combines small messages together. For SOVIA, it is also desirable to have a similar feature, where the consecutive data transfer requests of small-sized messages are combined into a larger one.

Table 4: SOVIA configurations evaluated in this paper.

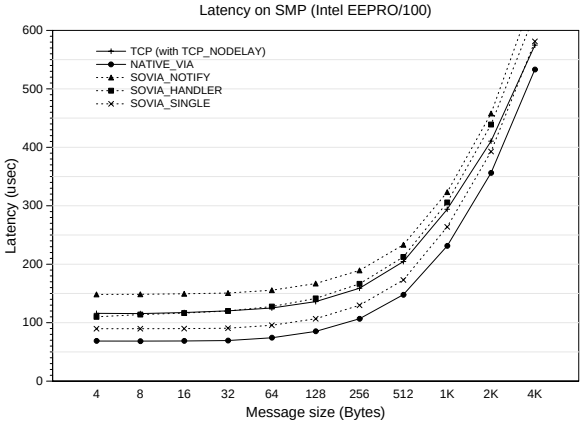
Labels	M	D	w	t	C
SOVIA_NOTIFY	notify functions	R	–	–	No
SOVIA_HANDLER	handler thread	R	–	–	No
SOVIA_SINGLE SOVIA_REG	application thread	R	–	–	No
SOVIA_COPY	"	B	–	–	No
SOVIA_HYBRID	"	H	–	–	No
SOVIA_BASE	"	"	1	1	No
SOVIA_FLOWCNTL	"	"	32	1	No
SOVIA_DACKS	"	"	32	16	No
SOVIA_COMBINE	"	"	32	16	Yes

Column M: message handling strategies.
Column D: outgoing data is registered (R), buffered (B), or conditionally buffered if it is less than 2KB (H).
Column w: window size.
Column t: threshold for delayed acknowledgments.
Column C: small messages less than 2KB are sent immediately (No) or combined together (Yes).

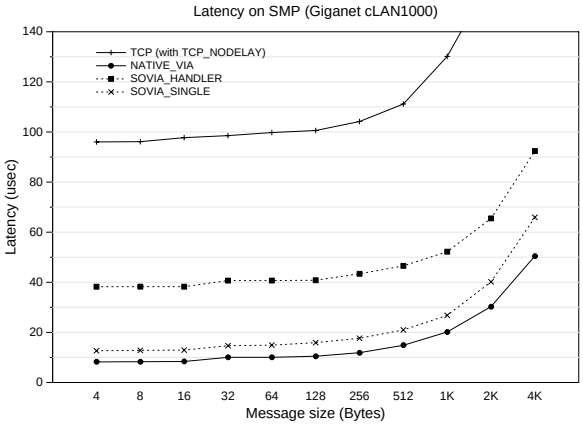
Our algorithm to combine small messages works as follows. The implementation of SOVIA already has an internal buffer which is used for sender-side buffering (cf. section 4.1). So far, a small message less than 2KB is copied into the buffer and then sent if the window size permits. However from now on, such a small message is appended into the buffer and the sender starts a timer which expires after, say, $100 msec$. Any other small messages requested within the expiration of the timer are also combined into the buffer. The data stored in the buffer is transmitted to the network either (1) when the timer expires, (2) when there is no enough room in the buffer for the new data transfer request, (3) when the requested message size is larger than 2KB, or (4) when the application calls `recv()` or `close()`. The maximum size that can be combined is 32KB, which is the message chunk size of SOVIA. For the messages larger than 2KB, it is too expensive to copy data, hence the current buffer is flushed and then the new message is transferred in a normal way.

In figure 8, the graphs labeled as SOVIA_COMBINE show the final bandwidth of SOVIA, where all optimizations, including the small-message combining, are applied. We can see that combining small messages significantly improves the bandwidth for messages less than 2KB. Combining small messages increases the latency of SOVIA by $1 - 2 \mu sec$ to manage a software timer. However, this feature may be turned off at run-time for latency-sensitive applications in the same way as TCP, where the Nagle algorithm is disabled by specifying the `TCP_NODELAY` socket option.

Table 4 summarizes SOVIA configurations and their parameters evaluated in this paper.

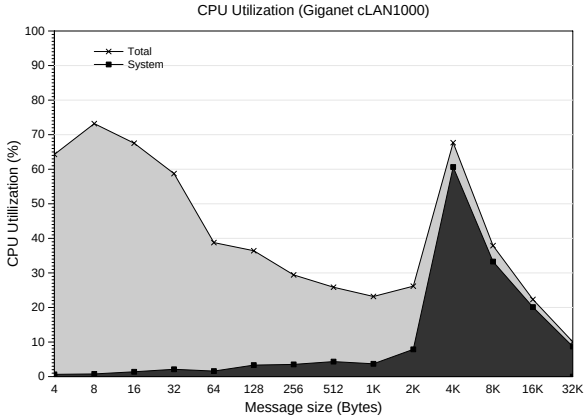


(a) M-VIA

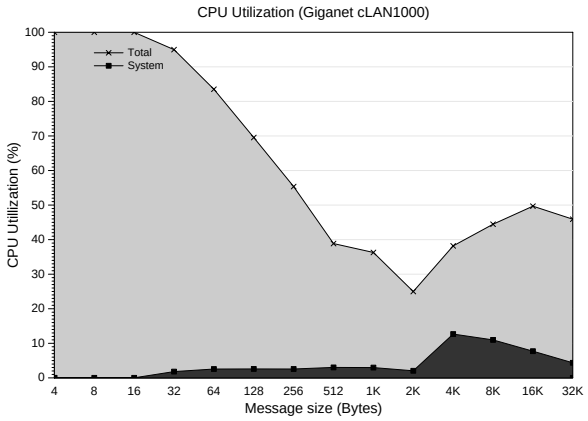


(b) cLAN

Figure 9: Latency on SMP systems



(a) Sender



(b) Receiver

Figure 10: CPU utilizations during bandwidth tests on cLAN (SOVIA_COMBINE)

4.3 Other Issues

Performance on SMP systems. In section 4.1, we have observed that the performance of SOVIA_NOTIFY and SOVIA_HANDLER is limited by the thread synchronization cost. Now we investigate how such multithreaded implementations perform under SMP (symmetric multiprocessor) systems with more than one processor.

Figure 9 plots the latency of TCP, native VIA, and SOVIA implementations with different message handling strategies. The figure is similar to figure 5, but now measured on dual-processor systems with SMP kernels. First, we can see the latency of TCP and native VIA has increased about 1.5 times on M-VIA, showing 116.0μsec and 68.6μsec respectively. Similarly, SOVIA_NOTIFY, SOVIA_HANDLER, and SOVIA_SINGLE show the increased latency of 148.3μsec, 110.0 μsec, and 89.7μsec respectively. This is because the SMP kernel has an additional overhead of acquiring and releasing a lock to access kernel data structures.

On cLAN, although the latency of TCP and SOVIA_HANDLER has increased to 96.0μsec and 38.2μsec respectively, native VIA and SOVIA_SINGLE reveal the same latency as on the uniprocessor kernel. This confirms that the operating system is not involved in data transfers on cLAN, due to the hardware support for user-level communication. On the other hand, the communication on M-VIA is not performed completely at user-level because of the lack of such hardware support in the legacy Fast Ethernet adapter, which hurts the latency of native VIA and SOVIA_SINGLE on SMP kernels.

Under SMP systems, the application thread and the notify (or handler) thread can run on a different processor, which may have a positive impact on the performance. However, our measurement results show there are no advantages of using multithreaded implementations of SOVIA on SMP systems.

Polling vs. blocking calls. Our implementation polls a send queue or a completion queue to find a completed descriptor. Optionally, it is possible to use blocking calls for the same task. When we use blocking calls both on the sender and receiver, the latency of SOVIA is increased to 19.4μsec for 4-byte messages and the peak bandwidth is reduced to 780Mbps on cLAN.

In spite of the performance degradation, using blocking calls has an advantage that they do not spend any CPU cycles

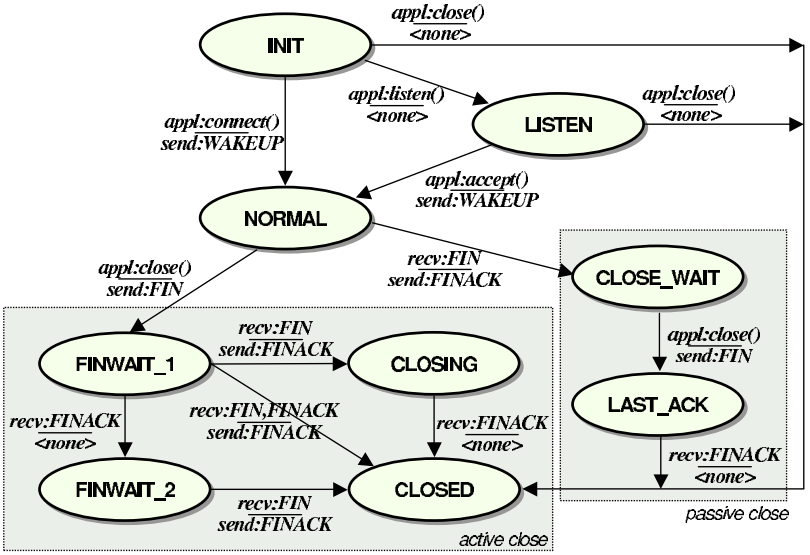


Figure 11: The state transition diagram of SOVIA

to wait for completed descriptors. Figure 10 depicts CPU utilizations of the sender and receiver during the measurement of bandwidth on cLAN using blocking calls. The CPU utilization is obtained by inserting a special kernel module which reports CPU cycles spent in kernel mode and user mode. In figure 10, a black (gray) area represents the portion of CPU utilizations spent in kernel (user) mode.

For the sender, the CPU utilization in kernel mode is suddenly increased at 4KB as we start to register outgoing data (cf. figure 10(a)). However, the relative portion of CPU cycles used in kernel mode is decreasing as the message size increases. When a 32KB message is sent, 8.7% of CPU cycles are consumed in kernel mode and only 1.4% in user mode. In other words, out of $335\mu sec$ which is the average time to send a 32KB message unidirectionally, CPU spends $29\mu sec$ in kernel mode, $5\mu sec$ in user mode, and idles for the remaining $301\mu sec$. Observing that the CPU utilization in user mode is maintained less than 10% for messages larger than 4KB, we can see most of CPU cycles in user mode are spent on copying and combining the outgoing data when the message size is less than 4KB.

Compared to the result of the sender, the kernel overhead of the receiver is very low, kept under 13% (cf. figure 10(b)). Instead, CPU spends considerable time in user mode to deliver the incoming data to the application. Because the incoming data is always buffered at the receiving side, the portion of CPU utilizations in user mode is higher than that of the sender, especially for messages larger than 4KB.

On the other hand, CPU utilizations for M-VIA are measured to be almost 100% regardless of the use of polling or blocking calls. This suggests the blocking calls are actually implemented using a busy-waiting on M-VIA.

In case the message handling is done by a separate notify thread or a handler thread as in SOVIA_NOTIFY and SOVIA_HANDLER, it is normally required to use blocking calls to receive messages. Otherwise, such a thread will slow down the computation of the application thread, even when there is no communication. Our single-threaded implementation of SOVIA, however, does not have this restriction, because the message handling is not overlapping with the computation.

Establishing and closing socket connections. Figure 11 illustrates the state transition diagram of SOVIA, which is very similar to the TCP's [27]. The notation, such as $\frac{appl:close()}{send:FIN}$, denotes a state transition when an application issues `close()`, and specifies that a FIN type of packet is sent to the peer as a result of the transition. In SOVIA, there are five types of packets; DATA, ACK, WAKEUP, FIN and FINACK. All the communications are done in the NORMAL state, and the states shown in the shaded areas are used for supporting active and passive close. To close a connection completely, both ends should exchange FIN and FINACK packets. A FIN packet is used by a node to notify the peer of its willingness to close a connection, and FINACK acknowledges the receipt of FIN.

The single-threaded implementation of SOVIA poses a problem when a connection is closed. The Sockets semantics requires that the application should return immediately from `close()`, after sending a FIN packet to the peer. Once the application executes `close()`, it does not call any Sockets API and there is no chance to handle incoming FINACK and/or FIN packets from the peer, which are necessary to receive before removing associated data structures.

To solve this problem, SOVIA creates a *close thread* when the number of open socket connections becomes zero, and asks it to handle the incoming messages. For each new connection, a WAKEUP packet is exchanged between peers. If the close thread receives the WAKEUP, it stops handling messages and is suspended until there is a notification. In this way, the presence of the close thread does not affect the ap-

Table 5: The performance of file transfers using FTP

	File 1		File 2	
File size (bytes)	13,827,767		65,257,948	
TCP/IP on Fast Ethernet	88 Mbps	(1.18 sec)	88 Mbps	(5.58 sec)
TCP/IP on cLAN	288 Mbps	(0.38 sec)	256 Mbps	(2.00 sec)
SOVIA on cLAN	736 Mbps	(0.15 sec)	552 Mbps	(0.93 sec)

plication's performance. Similarly, we create a *connection thread* whenever the application calls `listen()` on a port. The connection thread is used to wait for an incoming VI connection request.

RDMA. As presented in [10], the use of RDMA may increase the performance of long message transfers since the receiver does not have to be involved in the communication. To utilize RDMA (Write) operations, it is necessary for the receiver to inform the sender about target buffer address and memory handle using a mechanism similar to the three-way handshaking. This synchronization cost is generally amortized by transferring long messages.

Unlike MPI, however, Sockets API operates on a byte stream and the receiver may read a part of incoming data at a time. This means, even though the application requests the data transfer of a long message, enough space may not be prepared on the receiving side. Under this situation, our measurement indicates there is no significant advantage of using RDMA operations. Currently, SOVIA simply decomposes the long message into a sequence of chunks (sized 32KB) and each chunk is sent in a normal way as if it is issued by a `send()`.

5. FTP PERFORMANCE

In order to verify the functional validity of SOVIA, we have ported FTP (File Transfer Protocol) application over SOVIA. We slightly modified the FTP server (`linux-ftpd-0.16`) and client (`netkit-ftp-0.16`) contained in Linux NetKit 0.16 and measured the performance of file transfers between two nodes. Table 5 compares the performance of file transfers reported by the FTP client for two different sizes of files. To remove the effect of the disk speed, the source and destination files are stored in ramdisks.

The core loop of the file transfer operation is actually the same as our benchmark program which measures the bandwidth. Therefore, it is expected for the FTP application to achieve the peak bandwidth shown in figure 8. The measured bandwidth is, however, slightly lower than the expected one, showing 736Mbps and 552Mbps on cLAN for 13.8MB and 65.3MB files respectively. This is because there is an intermediate buffer cache layer and the performance of file reading and writing is not fast enough even though we used ramdisks. According to our measurement, a local ramdisk-to-ramdisk file copying has the bandwidth of about 580Mbps on the test platform, which is less than the SOVIA's peak bandwidth. Thus, the underlying file system becomes a bottleneck during the file transfer. On the contrary, it is the network that becomes a bottleneck in the case of TCP/IP on Fast Ethernet, hence the bandwidth does not depend on the file size.

We note that the raw performance of VIA is not delivered to the user application efficiently using Giganet's LANE driver which emulates TCP/IP inside the kernel. Although the peak bandwidth of native VIA is 815Mbps, the FTP application result in bandwidth less than 300Mbps with the TCP/IP driver, exploiting only 35% of the available bandwidth. Overall, SOVIA shows 6.3 – 8.4 times (2.2 – 2.6 times) higher bandwidth on cLAN for the FTP application, compared to the kernel-level TCP/IP driver on Fast Ethernet (on cLAN).

6. CONCLUDING REMARKS

This paper evaluates several design issues for building SOVIA, a high performance communication layer over VIA. First, we have optimized the critical communication path in terms of the latency and bandwidth, to maximize the performance delivered to user applications. And then, we have discussed other issues such as the performance on SMP kernels, the effect of using blocking calls, and the support for other Sockets semantics.

We find that the single-threaded implementation with conditional sender-side buffering is effective in reducing latency. To increase bandwidth, we have borrowed many ideas from TCP such as a sliding window protocol, delayed acknowledgments and piggybacking, and the ability to combine small messages.

With these optimizations, the performance of SOVIA closely matches that of native VIA. SOVIA shows the minimum latency of 45.0μsec and 10.5μsec on M-VIA and cLAN respectively, adding only 2μsec of overhead to the native VIA's latency. The measured peak bandwidth of SOVIA is 96Mbps on M-VIA and 814Mbps on cLAN for 32KB messages. Compared to the TCP/IP driver on Fast Ethernet, SOVIA results in 6.3 – 8.4 times higher bandwidth on cLAN for the FTP application. We believe the optimizations evaluated in this paper are also useful for other communication libraries, such as MPI.

During the development of SOVIA, we find that it is necessary to improve the thread synchronization cost of the Linux kernel if multiple threads are to be used with user-level communication architectures such as VIA. The high thread synchronization cost easily offset the low-latency benefit of the VIA and prevented us from using a separate communication handler thread. As soon as the VIA implementations are available, we are going to evaluate SOVIA on the latest 2.4.x Linux kernels again, which are known to show better performance than 2.2.x kernels.

We expect application programs written in Sockets API can seamlessly take advantage of the VIA through the SOVIA

layer. Currently, porting an existing application over the SOVIA layer requires modest user intervention. It is not sufficient to automatically replace the Sockets API with SOVIA, because sockets can be also accessed through file system interfaces or standard I/O routines. We are currently investigating a way to improve the compatibility of SOVIA by overriding the existing system calls and library routines. We also plan to port a parallel file system over SOVIA.

7. REFERENCES

[1] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Micro*, 31(11):53–60, Nov. 1998.

[2] N. J. Boden et al. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–36, Feb. 1995.

[3] P. Bozeman and B. Saphir. A Modular High Performance Inplementation of the Virtual Interface Architecture. In *Proc. Extreme Linux Conference*, 1999.

[4] P. Buonadonna, A. Geweke, and D. E. Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proc. SC '98*, 1998.

[5] R. Buyya, editor. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, Inc., 1999.

[6] B. Chun, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. *IEEE Micro*, 31(1):53–63, Jan. 1998.

[7] Compaq Computer Corp. Compaq ServerNet II SAN Interconnect for Scalable Computing Clusters. White Paper, Jun. 2000.

[8] Compaq Computer Corp., Intel Corp. and Microsoft Corp. Virtual Interface Architecture Specification Draft Revision 1.0. `http://www.viarch.org/`, Dec. 1997.

[9] G. Conte et al. ParMa^2: Porting VIA in LAM/MPI. University of Parma, Italy, `http://www.ce.unipr.it/research/parma2/`, 2000.

[10] R. Dimitrov and A. Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. In *Proc. 3rd MPI Developer's Conference*, 1999.

[11] Giganet Inc. cLAN for Linux: Software User's Guide, 2001.

[12] Giganet Inc. cLAN Hardware Installation Guide, 2001.

[13] W. Gropp and E. Lusk. User's Guide for MPICH, a Portable Implementation of MPI. Argonne National Lab / Mississippi State University, `http://www-unix.mcs.anl.gov/mpi/mpich/`, 2000.

[14] M. Itoh, T. Ishizaki, and M. Kishimoto. Accelerated Socket Communications in System Area Networks. In *Proc. Cluster 2000*, pages 357–358, 2000.

[15] J. Larson. The HAL Interconnect PCI Card. In *Proc. CANPC Workshop, Lecture Notes in Computer Science 1362*, pages 15–29, 1998.

[16] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, Inc., 1996.

[17] Microsoft Corp. Winsock Direct Specification. `http://www.microsoft.com/DDK/DDKdocs/Win2k/wsdpspec_1h66.htm`, 2001.

[18] National Energy Research Scientific Computing Center. MVICH: MPI for Virtual Interface Architecture. `http://www.nersc.gov/research/FTG/mvich/`.

[19] Ohio Supercomputer Center. MPI Primer / Developing with LAM. `http://www.lam-mpi.org/`, 1996.

[20] H. Ong and P. A. Farrell. Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network. In *Proc. of the 4th Annual Linux Showcase & Conference*, 2000.

[21] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. Supercomputing*, 1995.

[22] G. Pfister. *In Search of Clusters, Second Edition*. Prentice Hall, Inc., 1998.

[23] L. Prylli and B. Tourancheau. Protocol Design for High Performance Networking: a Myrinet Experience. Technical Report 97-22, LIP-ENS Lyons, France, Jul. 1997.

[24] S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. In *Proc. USENIX*, 1997.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[26] E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the Performance Potential of the Virtual Interface Architecture. In *Proc. ICS*, 1999.

[27] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley Publishing Co., 1994.

[28] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. Symposium on Operating System Principles*, pages 303–316, 1995.

[29] T. von Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *IEEE Micro*, 31(11):61–67, Nov. 1998.