Processor Controlled Off-Processor I/O

Marcel-Cătălin Roşu*
rosu@cs.cornell.edu
Computer Science Department
Upson Hall
Cornell University
Ithaca, New York 14853-7501

August 25, 1995

Abstract

The performance of modern RISC processors on operating system code is well below application code performance. The kernel code implementing communication services across the network is not an exception. Modern networking technologies are characterized by a small packet size, which further increases the communication overhead.

We took the approach of removing the kernel layer from the cross-machine communication path while still providing protection. The presence of a programmable communication processor on the network adapter made this experiment possible. The firmware running on the communication processor implements a *Virtual Communication Machine* (VCM); applications communicate with the VCM through shared memory without having to switch to kernel mode.

Data is transferred directly between application buffers and the network without any intermediate buffering in the user or kernel spaces. The VCM architecture makes this possible; in particular, the VCM can be programmed to access any location in the address space of an application. The main processor controls the communication but it is not directly involved with it; as a consequence, the overhead on the main processor is very low. The design not only provides very low latencies, but also minimizes the effect of communication on the main processor data caches.

We implemented the datagram subset of the Berkeley sockets interface on top of the VCM interface and integrated it with a user-level thread package. Multicast capabilities were added to the interface. Performance measured at both the VCM and socket layers is presented.

1 Introduction

This paper presents an experimental system for implementing user-level communication protocols. Our system runs over an ATM LAN but the same ideas can be used with other networking technologies for which the network adapter includes a programmable communication processor; all the physical memory of the host must be accessible to the communication processor. The most important characteristics of our design are:

- Data is transferred between the application buffers/data structures and the network interface registers without any intermediate buffering; we call this *0-copy* communication.

*Supported by ARPA/ONR grant N00014-92-J-1866.

- Communication is done *without* the main processor switching to kernel mode.

- Not only does the main processor execute I/O operations while in user mode but the number of instructions per send or receive can be as *low* as three instructions: a store, a load and a compare; the rest of the work is done by the communication processor on the network adapter.

- A *full protection* mechanism is included.

Because data is transferred directly from/into application buffers, only the cache sections mapped to these memory areas are affected. Our architecture allows applications to send and receive messages while in user mode eliminating the overhead of context switches. The advantage of executing as few CPU instructions as possible for an I/O operation is evident, especially when I/Os are frequent. The protection mechanism allows safe use in a multiuser environment.

A new communication architecture was designed to support these characteristics. We used the communication mechanisms in [3] (and their implementation) as the starting point of our design.

In a multitasking operating system, the network is accessed through a kernel layer that includes the protocol processing code and the device driver of the network adapter. The small packet size and the high bandwidth of some modern networking technologies, such as ATM, makes the time the main processing unit spends executing device driver code unacceptably high. Furthermore, using faster CPUs does not help significantly because processing of individual packets is dominated by memory accesses. A communication processor (CP) is often added to the network adapter card to reduce the load on the CPU. In the architecture described in [3], the network device driver is partially implemented in the firmware running on the CP and on the host CPU. The network adapter appears to the host as a (software) device handling much larger network packets. The host and the adapter communicate through shared (host or adapter) memory.

We took the approach of transferring more responsibility to the CP. We say that the new firmware running on the CP implements a *Virtual Communication Machine* (VCM). Fragmentation and reassembly is done directly from and into application buffers[1] rather than kernel buffers. The amount of processing per network packet by the CP is almost unchanged and a copy operation is eliminated. However, before any I/O operation, the memory pages holding the buffers must be pinned down and mapped into the address space accesible to the CP. These actions and their complements are implemented in an extension to the host kernel and the application must switch to kernel mode to execute them. Such switches are rare, however, because communication exhibits locality. VCM commands and error codes are passed between applications and the VCM through shared memory. To enforce protection, commands issued by an application can not access buffers of another application and applications can not use another application's connections to send or receive data. These are the basic components of the protection mechanism and are enforced by the VCM. The extra overhead is negligible.

To test the viability of our design we implemented the datagram subset of the Berkeley socket interface. This is not an implementation of the IP and UDP protocols. It is only a user-level library designed to hide the details of the VCM from the application programmer. Multicast groups are implemented as virtual hosts. The interface was integrated with the user-level thread package of the Horus system [7].

The next section mentions some of the work related to our experiment. The third section describes the architecture and functions of the VCM and kernel extension. The socket interface is

[1] In the following, an application buffer is any data structure in the application address space. However, the implementation presented in this paper has limited gather/scatter capabilities; only data structures composed of at most two arrays of words can be transferred in one I/O operation.

presented in the fourth section. The performance of our implementation is presented in the fifth section. The last section contains our conclusions and acknowledgments.

2 Related Work

The first 0-copy system we are aware of is presented in [2] and uses Ethernet adapters with scatter-gather capabilities. [4] describes the advantages of user-level protocol implementations. In [5] the effect of context switches on caches is analyzed; costs range from 10 to 400 microseconds per switch. Performance of modern RISC processors on operating system code is studied in [1]. The authors conclude that operating system performance is well below application code performance.

A communication model based on remote memory access is described in the context of more general work in [6]. The communication primitives we have implemented are based on the message passing paradigm but share a characteristic: both are designed for transferring contiguous segments to/from the virtual address spaces of applications on different machines. We implement protection in an entirely different way. The ATM adapters used are also much different: their work is based on a very simple network adapter that has no CP or DMA capabilities.

Our work extends the idea of very low overhead communication in Active Messages ([8]). We bypass the kernel as Active Messages does. However, in our approach almost all the communication overhead is handled by the CP instead of the CPU. In a way, the message handlers of the Active Messages are now implemented in the firmware on the network adapter card. But the VCM performs more tasks than handling incoming messages (as it will become evident later).

A user-level network interface called U-net has been concurrently developed at the Cornell Computer Science Department by Thorsten von Eicken, Werner Vogels, Anindya Basu, and Vineet Buch. The traditional TCP and UDP protocols, as well as an Active Messages layer, are implemented on top of the basic U-net interface. Both the basic U-net interface and the communication architecture presented in this paper provide user-level interfaces while enforcing protection. However, the VCM design allows the complete elimination of any buffering and provides the means to extend this feature to upper layers while the existing U-net interface uses one level of buffering.

In the FORE architecture (see [3]), the firmware running on the CP communicates with the host driver through a set of shared queues. The host supplies the firmware with free buffers through the free queue. The firmware demultiplexes the incoming cells according to the addressing information in the ATM cell header (connection/endpoint/...) and places them in the appropriate buffers. At the end of an incoming message, a description of the buffer holding the message is placed in the receive queue. Data is sent by placing a descriptor of the buffer holding the message in the send queue. The buffers and queues are kept in kernel space on the host. The host CPU copies data between application and kernel buffers. The firmware copies data between kernel buffers and the network interface. This copy is done with no additional buffering and with minimum latency using two techniques called "fly-by" and pipelining, respectively.

We have used the FORE architecture as the starting point of our design. Because the hardware supports the memory mapping necessary for 0-copy communication, we eliminate the extra copy and demultiplex incoming messages directly into user data structures. However, while the resulting architecture has all four characteristics listed at the beginning of the previous section, it required an entirely new implementation of the firmware to remove the alignment restrictions and to optimise buffer handling. While we implement "fly-by" DMA transfers and pipeline the outgoing data too, these were merely extensions of the FORE implementation. We handle protection in an entirely different way than [3].

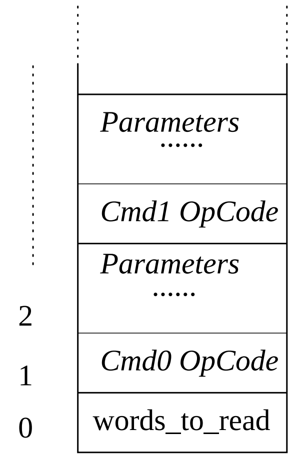

Figure 1: Command Vector Structure

3 The System Architecture

The main components of our architecture are: the VCM implemented in the firmware running on the CP, a user-level library used by applications to issue commands to the VCM, and an extension to the host kernel. The VCM is the central component and we start with a few definitions to help understand its architecture.

In the following, *host* stands for the machine running user applications. *Application* stands for any program written by a user and executing on a host. The *interpreter* is the program running on the CP. The interaction between the applications on a host and the interpreter can be best described as the interpreter implementing the *Virtual Communication Machine* and the application using the library to issue commands to the virtual machine. The address space accessible to the CP is called the *DVMA space*[2]. In the existing implementation there can be only one active interpreter/VCM servicing all the (possibly more than one) active applications on the host. The communication between applications and the VCM takes place through *shared memory.* The VCM has no memory except the application pages mapped into DVMA space. Each memory page must belong to exactly one application and the process identifier of the owner is passed to the VCM together with the page. Our kernel extension is responsible for keeping the VCM informed of any changes to this mapping.

A *command vector* (see Figure 1) is a one-dimensional array of words. VCM commands are placed in the command vector (starting with the second element), the same way that memory is used to store CPU instructions. A *command* consists of a command identifier word followed by arguments, each one word long. Commands take up to 7 arguments. The application needs two steps to issue commands: it first stores the command(s) in a command vector and then activates the vector. Commands in a vector are executed in order. After executing the last command in a vector, the VCM deactivates the vector; once deactivated, the vector can be reused to issue commands to the VCM. The command vector is activated/deactivated by manipulating its first word. The VCM selects command vectors for execution in an arbitrary order.

Figure 2 shows the interaction between the virtual machine and applications on a host. The

[2]DVMA space stands for Direct Virtual Memory Address space. In fact, this is the name of the address space accessible to the SBUS I/O devices (including the network adapter) in the SPARCstations we used for this experiment.

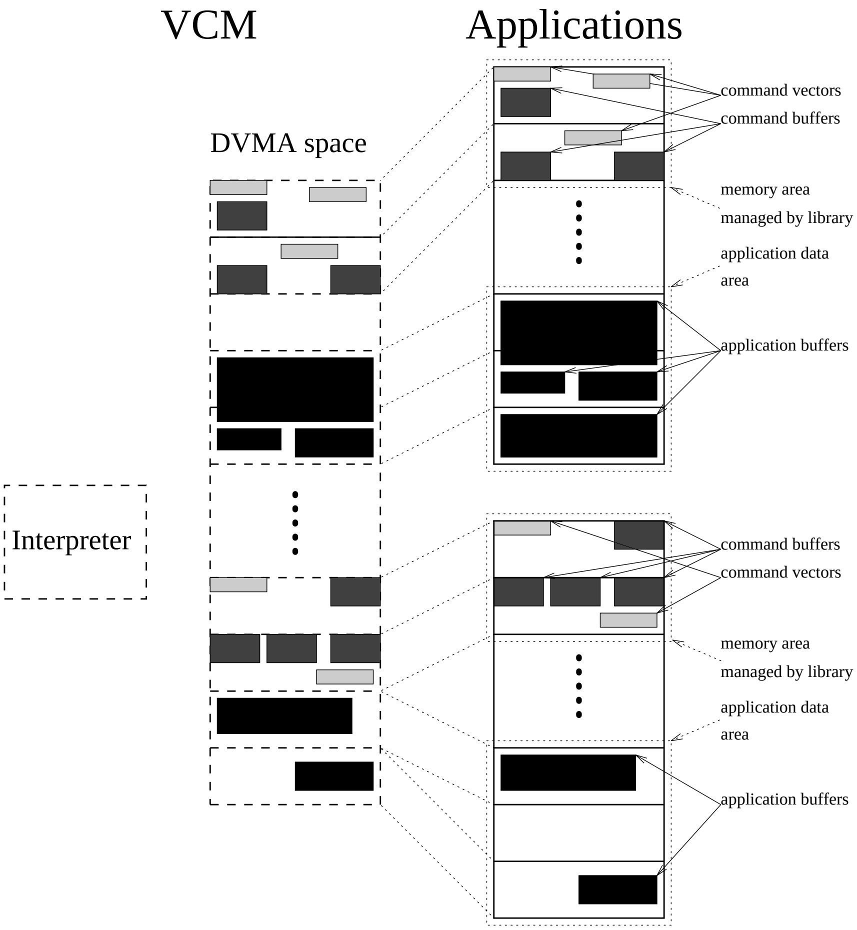

Figure 2: The Virtual Communication Machine

interpreter acts as the processing unit of the VCM and reads instructions from the command vectors. The application data segment is extended with some pages managed by the library; these pages are locked in memory and mapped into DVMA space when the application starts. Command buffers are memory areas used to store message headers, status words, and request messages[3]. Application buffers are the memory areas used by applications to communicate and can be anywhere in the original data segment of the application.

3.1 Protection

The VCM checks all the parameters of a command before executing it. The memory buffers passed as parameters must belong to the issuing application only. The process identifiers associated with the VCM memory pages are used to perform the check. The page(s) where the command vector used to issue the command is located must have the same process identifier with the pages where the buffers passed as parameters are placed.

Before any data transfer takes place a unidirectional connection between the sending and receiving applications must be established. The VCM is informed whenever a new connection is opened or closed. The connection identifier of a new connection and the process identifier of the owning application are always passed to the VCM. To disallow improper usage of connection identifiers, the VCM always checks if the issuing application owns the connection(s) passed as parameter(s).

3.2 Communication Primitives

A library implementing a message passing interface hides some of the details of the VCM from the application. The basic routines are *send message* and *accept message*. Both take a connection identifier as a parameter. This identifier is included in the header of each network packet carrying message data and it is used by the receiver to demultiplex incoming packets. Other parameters include the addresses and lengths of the buffers to be used.

The send routine instructs the VCM to copy data from the application memory to the network interface (NI) registers. Data is transferred without any intermediate buffering. The execution is considered completed after the last byte of the message is placed in the NI registers.

An incoming message is dropped if there is no accept previously posted for the corresponding connection. Otherwise, it is saved in the application memory in the buffers specified by the accept call. When dropped, the message is read from the NI registers and discarded. When saved, the message is copied from the NI registers to the final destination (in the application memory) without any additional copies in the adapter memory or in kernel or user buffers on the host. The execution of the accept primitive is considered completed after an incoming message is handled by it. At any given time there may be more than one accept posted for the same connection but only one of them handles the incomming packets of the connection (the active accept).

3.3 The Buffer Scheme

Because we intend to use the system as a tool for implementing user-level communication protocols, a general buffer scheme is provided. Messages consist of a header and a body. The header length may be zero, the body length must be positive. It is intended that header buffers be managed by the library while *any* contiguous memory area in the application space be used for the message body. Routines to allocate/deallocate small buffers in the memory managed by the library are included in the user library. The only restriction placed on user buffers is that they have to be

[3] To be explained later.

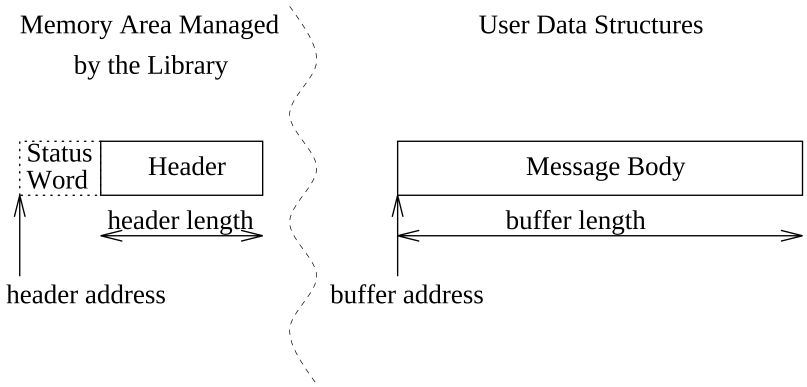

Figure 3: The Structure of Messages

locked in memory as long as they are in use by the VCM. Separate header and body buffers are provided to allow the implementation of user-level protocols that preserve the 0-copy property of the VCM layer.

Each VCM command has an associated *status word* (a word in the memory shared by the VCM and the application). The status words are placed at the begining of the header area in order to avoid using an extra parameter. Status words are written, mainly[4], by the VCM to inform an application about the result of executing the associated command. In the following, *header address* stands for the address of the shared memory variable and *header length* for the length of the actual header (see Figure 3).

The send primitive specifies the addresses and lengths of the message header and body. The address and length of the header area are among the parameters of the accept primitive; here is where the beginning of the message is stored. The rest of the message (the message body) is stored in the second buffer passed to the accept primitive; the length of the second buffer is a parameter too. If the message is too long, an error is signaled and the extra data is discarded. Before they start communicating, the sender and receiver applications must agree on the (fixed) header length; it is also useful (but not necessary) for the sender to know the maximum message size the receiver can accept.

3.4 The High-Speed and Low-Overhead Interface

To communicate, applications must first place commands in a command vector and then activate the vector. The VCM keeps polling the command vectors; as soon as one becomes active, it reads and executes the commands in the vector and deactivates the vector. The two step execution method above was considered inefficient for some applications. In particular, we wanted communication primitives with low latencies and negligible overhead on both the CP and the host processing unit. To achieve this goal, an extension of the original communication primitives was designed and implemented.

The new commands are called *loop-commands*. The name is inspired by the way we believe these commands will be used. They must first be read (using command vectors) and stored in the internal/microprogram memory of the VCM; in some sense they extend the VCM instruction set.

[4]Status words are used, in some cases, to pass information from the host to the VCM too.

After this step the loop-commands are said to be in *dormant* state. The application activates a dormant command by writing into its status word. After being executed, a loop-command returns to the dormant state and the VCM stores an error/success code in the status word. The address of the status word is one of the command parameters and it is read by the interpreter together with the command. To remove a loop-command from the VCM internal memory, the application writes a special value in the associated status word.

3.5 The API of the VCM layer

The *application programming interface (API)* consists of a library of subroutines. Most of these subroutines are meant to store the VCM command representation (2 to 8 words) in a command vector. After storing the representations of one or more commands in a command vector, the application activates the vector by setting the first word to the number of words it has placed in the vector. After reading all the commands in the vector the VCM deactivates the vector by resetting the first word to zero.

The routine to open a command vector (the first command vector is opened automatically when the application starts) is **OPEN_CMD_VEC**. **CLOSE_CMD_VEC** closes a command vector. Macros to read and write the shared variables used for synchronization between application(s) and the VCM are also provided. The API includes routines to allocate/deallocate command vectors in the memory region managed by the library.

Applications use **SEND_MESSAGE** to send a message. Data from the header area is sent first, followed by data from the buffer area. The **ACCEPT_MESSAGE** command handles incoming messages to the connection passed as parameter. Every connection has associated a FIFO queue of accept commands; the command in the first position is said to be the *active accept*. An incoming message is handled by the active accept, if any, otherwise it is discarded. After completion, the active accept is discarded from the queue. The parameters of SEND_MESSAGE and ACCEPT_MESSAGE are: connection to use, header address and length, buffer address and length.

The **ACCEPT_MULTIPLE** command is intended to be used by applications that receive a continuous stream of messages and can not always process each of them, but are interested in the most recently arrived message. This might be the case with a playback application. Whenever there is time to process the lastest message, the stream must first be redirected to another buffer and then the last message received in the original buffer can be delivered. The ACCEPT_MULTIPLE is very similar to the ACCEPT_MESSAGE command. The main difference is that once it handles a message, the second one is always discarded while the first one remains active if there is no other accept command in the queue.

The **REQUEST_MESSAGE** command uses two connections: one to send a (small) *request* message and one to receive the *requested* message. It behaves like a SEND_MESSAGE command followed by an ACCEPT_MESSAGE command. The send part of the REQUEST_MESSAGE command is different from an ordinary SEND_MESSAGE command because is sends data from only one buffer. The parameters of the REQUEST_MESSAGE command are: send and accept connections, request message address and length, header address and length, buffer address and length.

ACCEPT_STOP cancels all the pending accepts for a connection.

If a command can not be executed, the associated status word is written by the VCM right after the command is read. Otherwise, the status word is writen by the VCM after the command is executed. A SEND_MESSAGE command is considered executed after the message is sent. An ACCEPT_MESSAGE command is considered executed after it handles a message. The status word of an ACCEPT_MULTIPLE command is set after each message handled by the command. The status word of a REQUEST_MESSAGE command is set after the requested message is received. Before a command is issued, the application must set the status word to a value the VCM never writes.

LOOP_ACCEPT, **LOOP_SEND**, and **LOOP_REQUEST** are the loop-versions of the ACCEPT_MESSAGE, SEND_MESSAGE, and REQUEST_MESSAGE commands. Loop-commands are activated through the status word. The VCM interprets the upper half of the status word as the length of the message body and the lower half as the offset into the message body buffer. A non-zero value in the upper half activates the loop-command. When activated, the LOOP_ACCEPT command is placed in the same queue as an ACCEPT_MESSAGE posted for the same connection. The LOOP_SEND sends a message immediately after its activation. The LOOP_REQUEST acts like an activated LOOP_SEND followed by an activated LOOP_ACCEPT command.

After a LOOP_* command is executed, it returns to the dormant state. Setting the status word activates another execution of the command with the same buffer, header (and request message) address(es) and length(s) plus the new offset and length in the buffer area. Application may not make any assumptions about the order the VCM reads the status words of dormant loop-commands. The expected usage pattern is as follows:

- In the code before the loop, all the loop-commands needed in the loop are passed to the VCM (through one or more command vectors).
- During each iteration, the loop body activates the appropriate commands.
- In the code following the loop, the application removes from the VCM memory all the loop-commands passed to the VCM before the loop.

We implemented a subset of the Berkeley socket interface using loop-commands. The next section describes this interface.

4 The socket layer

Writing programs for the VCM is nontrivial, even when using the library. Programmers must allocate the necessary command vectors, lock application buffers in memory, place commands in the vectors and activate them, and later, test the status words for completion.

To provide a simpler interface to the VCM, we have implemented the datagram subset of the Berkeley sockets interface on top of the existing VCM interface. Implemented are: `socket, bind, recv, recvfrom, sendto, and close`. All the details listed in the previous paragraph are hidden from the programmer. The library maintains a cache of loop commands and memory buffers used for communication. Because communication exhibits locality, the overhead of this extra layer is very small; typically only a few instructions, primarily to check parameters, are needed.

Up to this stage, communication was performed over a few preset connections between hosts. In order to test our prototype in a more realistic environment, we included a *Connection Server* (CS). The server is implemented as a process running on each host and implements SPANS, the Fore Systems proprietary signaling protocol. An interesting characteristic of the CS is that it is implemented as an application running on top of the VCM layer. We considered the inclusion of the CS in the VCM unnecessary because opening and closing connections are not frequent actions.

To open/close a connection an application communicates with the local CS through the kernel extension. Signaling traffic uses a preset connection which is hardwired in the CS. When acting as

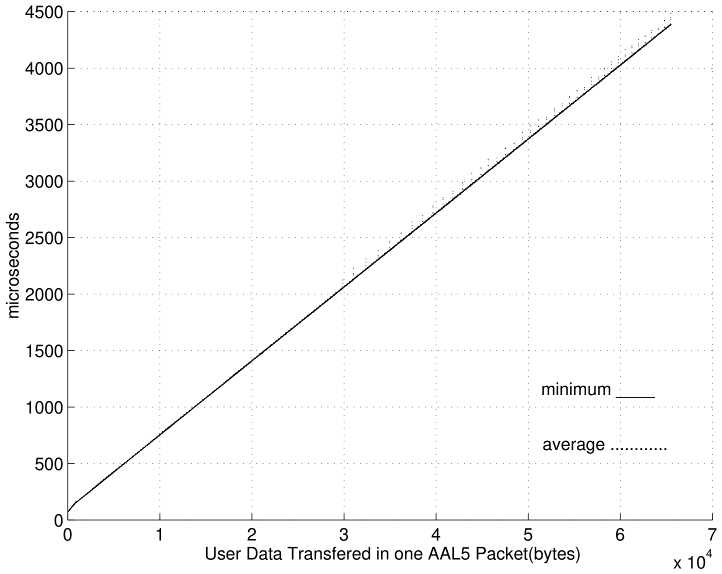

Figure 4: Minimum and Average Latency vs. User Data in the AAL5 Packet

a bridge between the CS and local applications, the kernel extension informs the VCM whenever a connection is open/closed and the process identifier of the application taking the action. The last feature enables the VCM to ensure that applications are using the appropriate connections when sending or receiving data.

Communication groups are another feature implemented by the CS. A group appears as a virtual host. Sending a message to a port on such a host translates into a multicast to all the applications that joined the group (executed `bind` with the appropriate parameters).

As the last step in the socket layer development we focused on a problem stemming from the design of the VCM: an incoming message on a connection with no accept posted is dropped. Even worse, in the single-threaded model implemented by our socket library, an application blocks in a `recv` or `recvfrom` call until a message arrives. As a consequence, there can be at most one posted accept per application. To fix this we integrated our socket library with the user-level thread package of HORUS (see [7]). In this new environment, a thread still blocks in the receive call until the expected message arrives but another ready thread (if any) is scheduled for execution. Finally, we measured the performance of the multithreaded environment and we present it in the following section together with the performance measured at the VCM layer.

5 Implementation and Performance

We have implemented our prototype using Fore Systems SBA-200 adapters to connect SPARCstations-20s (running SunOS 4.1.3) to a Fore Systems ASX-200 switch using 140 Mbits/sec TAXI fiber. The implementation consists of about 9000 lines of C code.

Latency is computed as half of the round trip time. The minimum and average are computed over a series of 100 round trips. The minimum latency is 65 microseconds for AAL5 packets that carry 4 bytes of data, and grows at a rate of about 70 microseconds/1Kbyte. Figure 4 shows the minimum and average latencies for messages 4 bytes to 64 Kbytes in length.

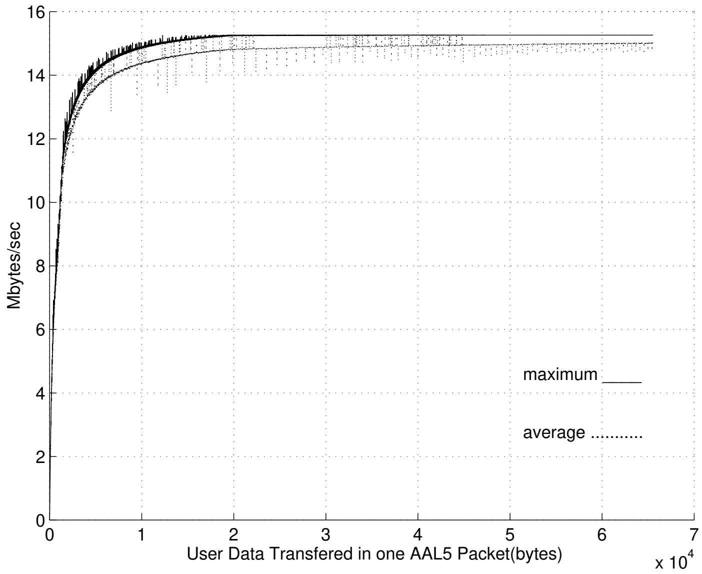

Figure 5: Maximum and Average Sender Bandwidth vs. User Data in the AAL5 Packet

The sender bandwith is computed as the data in 10 messages divided by the interval between the moment when the application activates the first send command and the moment when the VCM signals completion of the last send command (through the status word). The measurements were made with the sending host in multiuser mode. Both maximum and average bandwidths are shown in Figure 5.

The multi-threaded environment has higher startup costs. The latency graph has about the same slope but an intercept (minimum latency) of 135 microseconds. The roundrip time includes the overhead of two thread context switches; the extra 70 microseconds found in the measurement of the one-way latency are approximately the cost of one context switch.

6 Conclusion

A valid question is whether we really need a 0-copy mechanism. In order to implement reliable communication, data must be kept by the sending application until acknowledged. One solution is to block the sending thread until the message sent is acknowledged. Another solution is to save the outgoing data in some internal buffers until it is acknowledged. A more common solution is to use a sliding window protocol which usually requires buffering at both ends. However, it is easy to implement a non-0-copy mechanism on top of a 0-copy one.

Applications that need short latency communication can benefit ever more from 0-copy communication. As shown in the performance section, the slope of the one-way latency plot is determined by the wire bandwidth. This can not be true when data is buffered at the sending or/and receiving end. A simple calculation[5] gives a 30% increase in the slope of the one-way latency for a communication mechanism that uses one extra buffering at both ends and runs on workstations with a copy

[5] To compute the one-way latency use the following formula: $ct_{SEND} + \frac{x}{B_{CPU}} + \frac{x}{B_{WIRE}} + ct_{RECV} + \frac{x}{B_{CPU}}$, where x is the message size, B_{CPU} is the copy bandwidth of the workstation CPU, B_{WIRE} is the wire bandwidth and ct_{SEND} and ct_{RECV} and are two constants.

bandwith six times the wire bandwidth (a realistic assumption for the hardware we have used).

While limited, the hardware we have used (SPARCstations 10 and 20) provides enough suppport for a 0-copy mechanism. The small size of the address space accessible to the VCM is a limiting factor.

User-level communication is another advantage of the VCM architecture. Besides lowering the overhead on the main CPU and making possible user-level implementation of communication protocols, the VCM also allows applications to implement their own buffer management policies. However, removing the kernel from the communication data path leaves the VCM as the only place where protection can be implemented. Admission control must also be implemented by the VCM; otherwise a faulty/hostile application might overload the network. While our implementation does not enforce admission control we feel that the extra overhead is not significant. An informal argument can be that admission control and protection must be enforced on a per message basis, not a per packet basis.

Our design makes sense only when the network adapter includes a programable CP. We believe that this will be the rule rather than the exception for all networking technologies with small packets. How much of the main CPU's task can be transferred to the network adapter depends on the performance of the CP. However, because most vendors use an off-the-shelf design for the CP, we believe there will always be some extra processing capacity available for the extensions we describe in this paper.

Acknowledgements We would like to thank Ken Birman for the support given during the initial stage of this research. We thank Robbert van Renesse for help with the Horus threads package and for many helpful comments regarding this work. We would also like to thank Thorsten von Eicken, Werner Vogels and Anindya Basu for their help with the FORE software, firmware and hardware. Our work is heavily influenced by the work Thorsten von Eicken did on low overhead and latency communication ([8]) and by the class he taught.

We thank Ben Hao, Neel Jain, Mike Kalantar, and Peter Kopke for comments.

References

[1] T.E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, "The Interaction of Architecture and Operating System Design", In *Proceedings of the 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.*

[2] J.B. Carter and W. Zwaenepoel, "Optimistic Implementation of Bulk Data Transfer Protocols" In *Performance Evaluation Review*, Vol. 17, No. 1, May 1989, Pages 61-69.

[3] FORE Systems Inc., "Programmer's Reference Manual", August 1994.

[4] J.C. Mogul, R.F. Rashid, and M.J. Accetta, "The Packet Filter: An Efficient Mechanism for User-level Network Code", In *Proceedings of the 11th Symposium on Operating System Principles*, ACM SIGOPS, Austin, Texas, November 1987.

[5] J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance", In *Proceedings of the 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.*

[6] C. A. Thekkath, H. M. Levy, E. D. Lazowska, "Separating Data and Control Transfer in Distributed Operating Systems", In *Proceedings of the 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.*

[7] R. van Renesse, K. Birman, B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels "Horus: A Flexible Group Communications System", Cornell University's Computer Science Technical Report 95-1500.

[8] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation", In *Proceedings of the 19th ISCA, May 1992.*