Performance Evaluation of an Off-Host Communications Architecture

Jeffrey R. Michel, Alexander S. Waterman, and Alfred C. Weaver

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

The Computer Networks Laboratory at the University of Virginia has implemented the SAFENET lightweight protocol suite on the Navy's Desktop Tactical Computer (DTC-2). The software delivered includes the Xpress Transfer Protocol, a new transport and network layer protocol to support high-throughput, low-latency, priority-sensitive communications. One of our research questions was whether to embed XTP in the DTC-2's UNIX kernel or run XTP on an attached processor on the DTC-2's VMEbus. We implemented both strategies and the attached processor approach proved to have somewhat higher performance. In this paper we survey the hardware and software system environment, report the performance achieved at the MAC, transport, and user levels, and develop an analytic model for predicting throughput and latency as a function of vendor-provided or easily-measured system parameters. Our conclusions identify both the advantages and performance concerns inherent to off-host protocol execution.

1 Introduction

The United States Department of Defense has adopted SAFENET (Survivable Adaptable Fiber Optic Embedded Network) [6] as its communications architecture for mission-critical computer systems. SAFENET is unique in that it specifies a dual-protocol architecture. A suite of ISO protocols is specified for use in non-real-time systems (e.g., file transfer, electronic mail), and a so-called Lightweight Profile defines the protocols to be used for latency-sensitive applications. On the "lightweight" side, SAFENET's transport services are provided by the Xpress Transfer Protocol (XTP) [7]; SAFENET's physical layer is based on FDDI.

The Computer Networks Laboratory at the University of Virginia was awarded a Navy contract to provide the SAFENET lightweight protocol suite for the Desktop Tactical Computer (DTC-2), including XTP, an XTP interface to FDDI, an Ada API, directory services, and demonstration programs. The resulting DTC-2 architecture, Ada binding, and XTP implementation are described in detail in [1, 2, 3].

Our challenge was to implement the communications services in a way which preserved the throughput and latency goals of the Lightweight Services. Two approaches seemed obvious: (1) embed XTP protocol processing in the DTC-2's UNIX kernel, or (2) run XTP on an attached processor on the DTC-2's system bus. The first idea was conventional; the second was intriguing because it offered the opportunity for off-loading the host and gaining some parallelism between the user application running on the host and the protocol running on the attached processor. Whether or not that potential parallelism would result in enhanced performance was the object of our research project.

We implemented both strategies, and the attached processor architecture proved to have somewhat higher performance overall. The details of the kernelized approach may be found in [8]. This paper summarizes our results for the attached-processor approach; full details are in [5].

2 Off-host implementation

In order to achieve the potential advantages of an off-host communications architecture, great care must be taken in its design and implementation. In the following sections we discuss the possible benefits of an off-host architecture and how we sought to achieve them through our hardware and software design choices.

2.1 Potential advantages of an off-host architecture

An off-host communications architecture can benefit both the host processor and the protocol implementation. As a result, host applications can run faster and more predictably and protocol performance can be maximized. The host processor may receive the following advantages due to the off-host approach:

Reduced host load. Since all protocol processing activities are handled on the attached processor, no host processor cycles are consumed for protocol processing.

Predictable application processing. Since transport protocols perform retransmissions, react to incoming traffic, and sometimes block on network access, the execution times for protocol processing are nondeterministic. When using an off-host architecture, such nondeterministic activity occurs on the attached processor and not on the host. As a result, the CPU demands of host processes become more deterministic, thus promoting the predictable execution of such processes.

Reduced and bounded interrupt arrivals. A transport protocol typically receives interrupts from its lower-layer communications service in order to indicate packet activity. Some minimum number of packets are necessary for each message, and, in addition, an unbounded number of packets may arrive from other hosts or result from retransmissions. Each packet can result in a processor interrupt. When the protocol runs on an attached processor, the host is shielded from these interruptions. When using an off-host architecture, host interrupts need only occur on a per-message, rather than a per-packet, basis. This results in substantially fewer host interrupts per message. Furthermore, the protocol interrupts the host only for those messages which the host desires to receive. This enables application processes to place a bound on the amount of interrupt handling in which the host will engage. These characteristics have been referred to as a "network firewall" [4].

The communications protocol can receive the following benefits when run on an attached processor:

Dedicated processor cycles. When run on the host, a communications protocol must contend with application tasks for processor cycles, thereby incurring overhead from scheduling and context swaps. Such overhead reduces the amount of work the processor can perform for its application and causes poor response time to protocol events. With a dedicated processor, more cycles are available to the protocol, thereby raising throughput, and context swaps can be avoided, thereby lowering latency and reducing the likelihood of buffer overflow.

Specialized hardware. Protocols can benefit from special hardware features such as high-speed RAM, DMA copies, block moves, and fast checksumming. Such hardware may allow certain protocol operations to be accelerated or performed in parallel. Components providing these services can be utilized on the protocol processor board. Furthermore, if the network interface resides directly on the processor board, a fast communication path between it and the protocol can be achieved.

Ideal operating system environment. It has been recognized that an operating system environment can place severe constraints on a protocol's performance and proper implementation [9]. With an attached processor board, one is free to choose the ideal operating system for the protocol implementation, including none at all.

2.2 Hardware

Figure 1 depicts our hardware and software architecture. The host platform for our work is the C3 Desktop Tactical Computer (DTC-2). This machine features a 25-Mhz SPARC CPU and uses the SunOS 4.1.1 operating system, a variant of UNIX. The DTC-2 employs a VME backplane bus which we use to contain our FDDI adapter and protocol processor. The FDDI board is manufactured by Network Peripherals (NP), and the protocol processor board is a Motorola MVME-167A (167). The 167 contains a 25-MHz MC68040 microprocessor and 8 MB of on-board, 70-ns DRAM, all of which is addressable on the VMEbus. An FDDI MAC device driver resides on the 167 and communicates with the NP board through both shared memory and VME interrupts. The system is configured such that a portion of the 167's memory is mapped into the DTC-2 kernel's 32-bit virtual address space, facilitating communication through shared memory between the 167 and DTC-2. Communication also occurs through interrupts generated on the 167 board for the DTC-2. The 167 runs pSOS+, a lightweight multitasking operating system, and the off-host implementation of XTP.

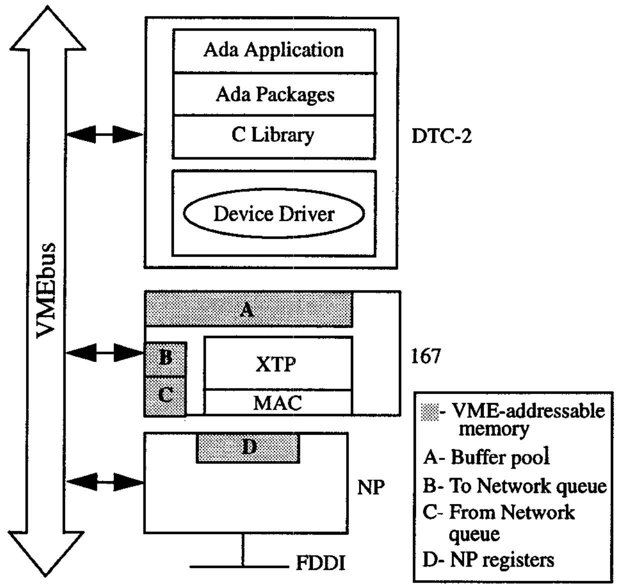


Figure 1: System Architecture

2.3 Software

Our software architecture exists to provide transport services to Ada applications running as UNIX processes. It consists of a set of Ada packages, a C library, a UNIX character device driver, an implementation of XTP, and an FDDI MAC device driver. These individual software components are partitioned onto our two processor platforms and run in several address spaces. On the host, the Ada package bodies and C library run in the address space of their Ada application's process, while the character device driver runs in the kernel with its own distinct address space. On the attached processor, the XTP implementation and MAC driver run as a set of pSOS+ tasks sharing a common address space.

An Ada application employs primitives such as `SEND_MESSAGE` and `GET_MESSAGE` from the Ada packages to perform network communication. The Ada program is linked with the library of C code which contains the user-level portion of the implementation of the transport services. Aside from the management of some user-level resources and state information, the function of this C code is to command the protocol processor and receive its acknowledgments of command completion through the use of the character device driver.

The C-library interacts with the transport layer via a standard UNIX character device interface. It utilizes system calls such as `open()`, `read()`, and `write()` to communicate with the off-host protocol processor. Commands are packaged in *control blocks* that include the command, a connection identifier with which it is associated, and pointers to any required user data buffers. The device driver maintains enough information on each connection to associate it with its user process. The control block communication to the protocol processor follows an asynchronous request-acknowledgment protocol. To issue a control block, a user process issues a `write()` to the protocol processor. Once the control block is queued for the protocol processor, the device driver returns control to the user. When the command associated with the control block completes, the protocol processor notifies the user process by queueing an *acknowledgment block* and interrupting the host. This interrupt invokes a device driver interrupt routine which in turn issues a UNIX signal to the process owning the connection identified in the acknowledgment block. A signal handling routine in the user process performs a `read()` in order to access the acknowledgment.

The device driver maintains the status of all XTP connections and manages the allocation of connection identifiers along with shared-memory data buffers associated with each connection. A 4-MB memory region of the local memory on the protocol processor is allocated as a buffer pool for incoming and outgoing messages.

The control- and acknowledgment-block communication mechanism between the host and the protocol processor employs two queues, the FN (From Network) queue and the TN (To Network) queue. The FN and TN queues reside in shared memory and allow both processors to command and acknowledge each other asynchronously. The host enqueues control blocks on the TN queue to be dequeued by the protocol processor, whereas the protocol processor enqueues acknowledgment blocks on the FN queue to be dequeued by the host.

Unfortunately, control blocks and user message data are transferred between the host and protocol processor using programmed I/O. Although both the host and the protocol processor support the use of DMA, memory architecture limitations on the host prohibited a straightforward implementation. Such limitations are due to the fact that only the highest 1 MB of kernel virtual address space is addressable on the VMEbus (DVMA space), and the host maps this region to the lowest megabyte of VME address space.

3 Performance

In order to illustrate the overall performance of our architecture, we provide throughput and latency measurements at its MAC layer, transport layer, and user level for the full range of message sizes available at each. We also include a profile of overall processing time for a `SEND_MESSAGE` operation and an assessment of the host CPU load incurred to perform it.

3.1 MAC layer

The MAC layer provides the transport protocol with a raw data link service over the 100-Mbps FDDI network. The results below were obtained using a pair of Network Peripherals FDDI boards and a pair of Motorola MVME 167A processor boards in two stand-alone VME card cages. During these tests, no operating system ran on the processors, and we performed communication between the processor board and FDDI board using block-mode DMA over the VMEbus. Figure 2 shows our end-to-end latency, and Figure 3 shows our throughput. Here, latency is half of the round-trip time of a frame, and throughput measures the rate at which the MAC driver can transmit frames with no receiver. A minimum latency of 91.5 ns occurs for a frame with no payload, and the maximum throughput of 56.6 Mbps occurs for frames carrying a payload of 4,487 bytes (4,500-byte maximum FDDI frame less LLC and SNAP headers).

3.2 Transport layer

The transport protocol (XTP) provides reliable end-to-end delivery of memory buffers from the local memory of one protocol processor board to another. Results were obtained at the top-level interface to XTP, which is the interface used by our UNIX device driver. For the throughput measurements, the data transfer operations were pipelined using the "streaming" features of our XTP implementation, and data checksums were disabled via the XTP NOCHECK option [7]. End-to-end latency and throughput are shown in Figures 2 and 3, respectively. As before, latency measures half the round-trip time of a message. In this case, throughput is for reliable, connection-oriented message transmission. The minimum latency of the protocol is 2.7 ms for a one-byte message, and a maximum throughput of 23.8 Mbps occurs for a 64-KB message.

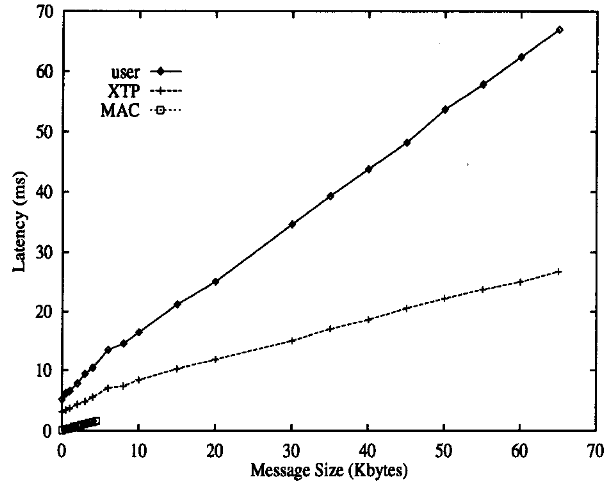


Figure 2: End-to-End Latency vs. Message Size

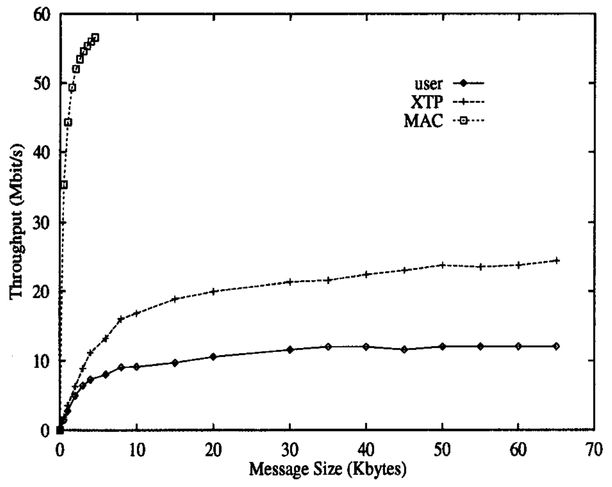


Figure 3: Throughput vs. Message Size

3.3 User level

All user-level performance measurements were obtained using two Ada application processes running on separate DTC-2 hosts and employing the connection-oriented SEND_MESSAGE and GET_MESSAGE API primitives. Latency measures half the round-trip time of a message sent from one Ada application to another. The minimum latency occurred at a message size of one byte and was 5.1 ms. For the throughput measurements, the communication primitives were pipelined using the asynchronous mode of our Ada API, and XTP's rate control features [7] and NOCHECK option were used to provide maximum performance. Such results were achieved using a RATE value of 1.5 MB/sec and a BURST value of 10 KB/burst. These settings result in our architecture's peak throughput: 12.1 Mbps at a message size of 64 KB.

3.4 Profiling

To study the user-level results, we profile the execution of the SEND_MESSAGE and GET_MESSAGE primitives used in the performance tests. First we concern ourselves with the processing of short messages. Table 1 lists where the host processor spends its time during a synchronous SEND_MESSAGE call for a one-byte message. The "other" category accounts for accumulated measurement error and operations which consume less than 10 μs. The profile for a one-byte GET_MESSAGE operation is similar.

Operation	μs/call	Calls	total μs
wait for completion	3,881	1	3,881
physio() & iodone()	301	3	903
read() system call	81	2	162
signal delivery	143	1	143
write() system call	83	1	83
disable signals	25	3	75
enable signals	22	3	66
control block copyin()	29	1	29
ack block copyout()	23	1	23
other	▨	▨	732
total	▨	▨	6,097

Table 1: Profile of a One-Byte SEND_MESSAGE

The wait for completion measures the amount of time that the host awaits an indication that the operation is complete. Although it accounts for most of the total time, the wait itself does not require host processor cycles; rather, it is a function of the transport protocol's performance. All other times in the table are incurred by various UNIX services. Even the simple operation of disabling a signal takes a surprisingly large amount of time because it requires a UNIX system call. Most significant is the time required to perform the processing of physio() and iodone() routines, kernel utilities that manage the arguments of the read() and write() system calls. It is clear that UNIX overhead dominates the host processing time when the message size is small.

To study the processing overhead of long messages, we profile a SEND_MESSAGE of 64 KB in Table 2. As before, the total time is dominated by the wait for completion. However, with the long message size, the cost of allocating and deallocating a buffer on the local memory of the protocol processor now becomes notable. Furthermore, a startling result is the time required to perform the data `copyin()` across the VMEbus. For long messages, the time to perform this backplane transfer dwarfs all UNIX overhead and even rivals the wait for completion.

Operation	µs/call	Calls	total µs
wait for completion	23,455	1	23,455
data `copyin()`	18,164	1	18,164
`physio()` & `iodone()`	301	3	903
`read()` system call	81	2	162
signal delivery	143	1	143
get and return buffer	109	1	109
`write()` system call	83	1	83
disable signals	25	3	75
enable signals	22	3	66
control block `copyin()`	29	1	29
ack block `copyout()`	23	1	23
other	▨	▨	1,636
total	▨	▨	44,848

Table 2: Profile of a 64-KB SEND_MESSAGE

The profile for a 64-KB GET_MESSAGE is similar except that the data `copyin()` in the table above is replaced with a 23,714-µs data `copyout()`, and the wait for completion takes only 17,525 µs. Note the counterintuitive result that the host spends more time copying the received message from the protocol processor than it does waiting for the protocol processor to handle the reception of the message from the network.

3.5 Host load incurred

The total time necessary for a SEND_MESSAGE call is 6.1 ms for a one-byte message and 44.8 ms for a 64-KB message. However, 3.8 ms of the one-byte operation and 23.5 ms of the 64-KB operation are spent waiting on command completion and are therefore free for other host processing. Thus the one-byte call effectively consumes only 2.2 ms, and the 64-KB call consumes only 21 ms of actual host processor time. Furthermore, only one host-processor interrupt is generated for a call with either message size.

4 Architecture-based performance model

In light of our system implementation and our results, a fundamental question is "How well did we exploit our hardware and software architecture in order to maximize user-level performance and minimize host load?" A question for developers of future off-host protocol processing systems is "How can we predict the possible performance of a given off-host architecture before its implementation has begun?" We have developed an architecture-based performance model for off-host protocol processing which attempts to provide answers to such questions.

4.1 Background and purpose

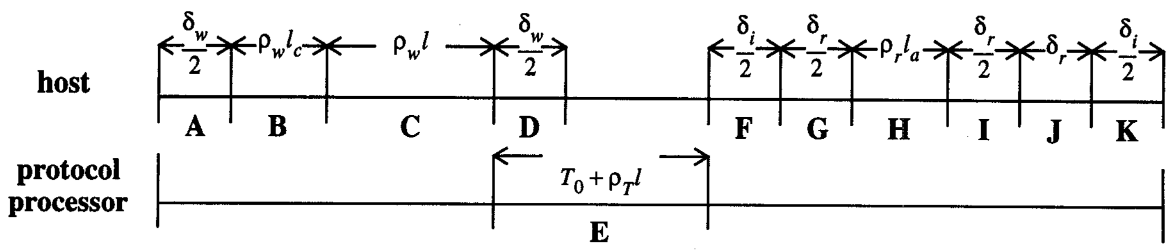
In a previous work on off-host protocol processing [4], Kanakia and Cheriton estimate the performance of their proposed Network Adapter Board. Their analysis takes into account such factors as bus transfer time, network transfer time, and memory access time in order to predict the request-response delay of their device. The approach is quite informal, and it takes into account factors above and below the level of their transport protocol (VMTP).

Our approach differs from Kanakia and Cheriton's in two main areas. First, our performance predictions result from the evaluation of simple, yet formal, mathematical expressions. Second, the transport protocol is treated as a "black box," making its performance characteristics parameters of the model. That is, the transport protocol processor is analyzed in isolation, outside the context of any host processor. Its performance parameters are then used by the model to predict the performance that will be visible at the user level in the completed system.

Our performance model has as its goals *simplicity* and *practicality*. With our model there is no need to acquire the probability distributions of any input parameters as would be required of a queueing theory model. Rather, one need only use readily-available, time-average measurements or vendor-provided specifications. With minimal effort one can predict the end-to-end latency and throughput seen at the user level. One may also obtain an estimate of the amount of host processing time required for data transfer operations. Such predictions are practical in that they are of interest to the developers of systems making use of off-host protocol processing. Furthermore, the model's complexity and accuracy are scalable in the following manner. In progressive stages of system implementation, one may use knowledge of the overhead of the software components in the implementation to gain more accurate predictions using the general methodology of the model.

4.2 Terminology and symbols

In addition to parameterizing protocol-processor performance, additional input parameters characterize the behavior of the components of the data and control path from the user level to the protocol-processor level. Our model includes terms which account for the delay introduced by each component. However, it also takes into account the processor parallelism which causes overlap in components of host and protocol-processor execution. This



A:	time to enter the device driver `write()` routine
B:	time to copy the send command to the protocol processor
C:	time to copy the user data to the protocol processor
D:	time to return from the device driver `write()` routine
E:	protocol processing time required to service the request
F:	time to initiate the user-level interrupt handler routine
G:	time to enter the device driver `read()` routine
H:	time to read the acknowledgment from the protocol processor
I:	time to return from the device driver `read()` routine
J:	time to enter and return from the device driver `read()` routine
K:	time to return from the user-level interrupt routine

Figure 4: Delay Components of the User-Level Period

parallelism causes certain terms to drop out of the analysis. All predicted quantities, as well as many terms, are a function of the message length, represented here by the symbol l.

Symbol	Meaning
δ_w	round-trip delay of the `write()` driver entry point
δ_r	round-trip delay of the `read()` driver entry point
δ_i	round-trip user-level interrupt delivery time
l_c	length in bytes of a control block
l_a	length in bytes of an acknowledgment block
ρ_w	per-byte time required for the host to write data or control information to the protocol processor
ρ_r	per-byte time required for the host to read data or acknowledgments from the protocol processor
δ_0	time required for the protocol processor to transmit a minimum-length message end-to-end
ρ_δ	per-byte time required for the protocol processor to transmit data end-to-end
T_0	period of the protocol processor in transmitting a minimum-length message
ρ_T	per-byte addition to the period of the protocol processor in transmitting a message

Table 3: Input Parameters of the Model

We must first present some terminology for two quantities used in the analysis, *delay* and *period*. In our model, *delay* measures the time between the beginning of a transmission operation at one endpoint and the completion of the corresponding receiving operation at a peer endpoint. In contrast, the *period* of an operation is the time between the beginning of the operation and the *local indication* of its completion. The three most fundamental output quantities of the model are the user-level delay, $\delta(l)$, the user-level period for a transmitting operation, $T_t(l)$, and the user-level period for a receiving operation, $T_r(l)$. The delay predicts user-level end-to-end latency, and the periods can be used to predict the user-level throughput for synchronous transfers, $\tau_s(l)$, and the user-level throughput for asynchronous transfers, $\tau_a(l)$. In addition, one may predict the host processing overhead (in units of seconds) required to transmit a message, $H_t(l)$, or to receive a message, $H_r(l)$. Table 3 lists the parameters of the model and their respective symbols.

4.3 Analysis and formulae

All derivations are based on summations of time values. Therefore, in the following analysis, the ρ-quantities (in units of seconds/byte) are multiplied by the l-quantities (in units of bytes) to produce elapsed times. To estimate the user-level period, we undergo the following analysis. Processor execution time lines for a transmitting operation are given in Figure 4. Note that δ_w, δ_r, and δ_i are defined above as round-trip times rather than one-way times because round-trip times are easier to measure in practice. The one-way time of each is then estimated as simply half the round-trip time. The δ_r term at location **J** is subtle. Because interrupts do not queue, a single interrupt may signal the arrival of more than one protocol-processor acknowledgment. The δ_r term represents a second read of the protocol processor's status to determine if another acknowledgment is pending. Given these time lines, we may sum the serial components to obtain the user-level period for a transmitting operation.

$$\begin{aligned} T_t(l) &= \frac{\delta_w}{2} + \rho_w l_c + \rho_w l + \max\left(\frac{\delta_w}{2}, T_0 + \rho_T l\right) + \\ &\qquad \frac{\delta_r}{2} + \rho_r l_a + \frac{\delta_r}{2} + \delta_r + \frac{\delta_i}{2} \\ &= (\rho_w + \rho_T)\, l + \frac{\delta_w}{2} + T_0 + \delta_i + 2\delta_r + \rho_w l_c + \rho_r l_a \end{aligned}$$

We base the second step of the derivation on the assumption that $\delta_w/2 \le T_0 + \rho_T l$, which should be quite acceptable in most cases. This is an example of where processor parallelism causes a serial delay term to drop out.

The analysis of the other output quantities proceeds along these same lines and is omitted for brevity. For more information, see [5]. The period for a receiving operation is

$$T_r(l) = (\rho_r + \rho_T) l + \frac{\delta_w}{2} + T_0 + \delta_i + 2\delta_r + \rho_w l_c + \rho_r l_a.$$

From $T_t(l)$ and $T_r(l)$ we obtain the synchronous user-level throughput:

$$\tau_s(l) = \frac{l}{\max(T_t(l), T_r(l))}.$$

The derivation of the asynchronous user-level throughput, $\tau_a(l)$, is more complex due to the fact that we must take into account both the parallelism of the host and protocol processors and the shared resource of the backplane bus. The derivation is based upon the following assumptions about the steady-state operation of the asynchronous mode:

- The host keeps the command queue to the protocol processor full.
- The protocol processor is always either busy or blocking on bus access.
- An insignificant portion of T_0 is spent on bus access.
- The time represented by $\rho_T l$ is dominated by that of bus transactions.

This implies that the $\delta_w/2$, T_0, δ_i, and $2\delta_r$ terms of $T_t(l)$ and $T_r(l)$ drop out of the analysis so we have:

$$\tau_a(l) = \frac{l}{\max(T_t(l), T_r(l)) - (\frac{\delta_w}{2} + T_0 + \delta_i + 2\delta_r)}.$$

The expression for $\delta(l)$ is

$$\delta(l) = (\rho_w + \rho_\delta + \rho_r) l + \frac{\delta_w}{2} + \delta_0 + \delta_i + 2\delta_r + \rho_w l_c + \rho_r l_a.$$

The minimum host processing time required to transmit a message is

$$H_t(l) = (l_c + l)\rho_w + \delta_w + \delta_i + 2\delta_r + l_a\rho_r,$$

and the minimum host processing time to receive a message is

$$H_r(l) = (l_a + l)\rho_r + \delta_w + l_c\rho_w + \delta_i + 2\delta_r.$$

4.4 Predictions

For our architecture, the inputs to our model have the values listed in Table 4. Given this input, the model predicts the user-level end-to-end latency shown in Figure 5 and the user-level synchronous throughput shown in Figure 6, both as a function of message length. Furthermore, the model predicts the minimum host processing time to transmit a message to be 1.4 ms for a one-byte message and 19.5 ms for a 64-KB message.

Parameter	Value
δ_w	392 µs
δ_r	382 µs
δ_i	143 µs
l_c	104 bytes
l_a	64 bytes
ρ_w	279.45 ns/byte
ρ_r	364.8 ns/byte
δ_0	3.112 ms
ρ_δ	363.97 ns/byte
T_0	2.062 ms
ρ_T	295.82 ns/byte

Table 4: Input Parameter Values for our Architecture

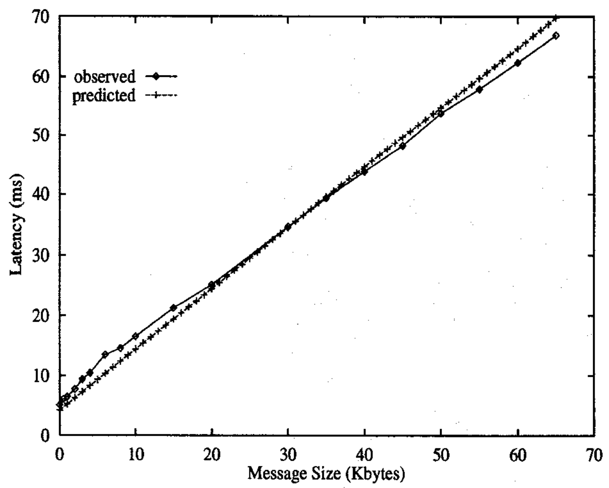


Figure 5: Latency vs. Message Size

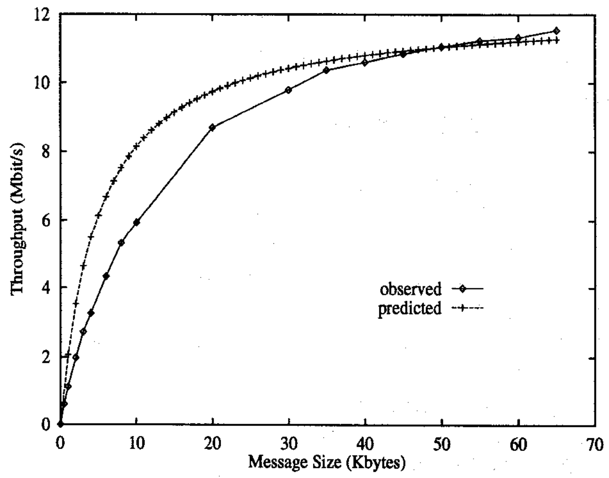


Figure 6: Throughput vs. Message Size

4.5 Comparison

By examination of Figure 5 we can see that the end-to-end latency prediction is a good estimate of the observed user-level end-to-end latency. Figure 6 shows that the user-level throughput predictions of the model are relatively accurate for large messages; however, they overestimate the throughput for short messages. This discrepancy indicates that there is more delay overhead in the data path than the terms of $T_t(l)$ and $T_r(l)$ take into account. A comparison of the observed minimum host processing time to perform a transmission with the predictions above indicate a 36% difference for a one-byte message and a 7.1% difference for a 64-KB message.

5 Conclusions

The performance results reported and the platform chosen for our SAFENET research enable us to assess the proposed benefits of an off-host implementation.

Reduced host load. Due to addressing limitations, the architecture we chose did not reveal the reduction of host load we anticipated. A large percentage of time is spent in both the SEND_MESSAGE and RECEIVE_MESSAGE operations performing programmed I/O, where programmed I/O entails data copies that involve the host processor. Some amount of programmed I/O will be necessary in any off-host or in-host implementation to command the attached network device. Address space restrictions forced us to use host-driven programmed I/O to transfer data between the host and the attached processor. We believe that this is a problem inherent to our specific hardware architecture, and not a problem inherent to off-host protocol processing. The introduction of user buffer DMA capabilities would reduce the VME data copy overhead to a constant DMA setup time, not dependent on data length, thus host load would be constant and low.

Predictable application processing. Predictability is achieved through the asynchronous control block strategy employed for command submission. The control block is queued for processing in constant time. Immediately afterward, control returns to the user process such that it may continue processing; the system need not wait on the completion of the data transfer to regain control.

Reduced and bounded interrupt arrivals. Preliminary measurements show that the transmission of one 64-KB message produces 26 host interrupts for a UNIX in-host implementation of XTP, compared to 1 host interrupt for our off-host implementation. This is a reduction in host-processor interrupts of 96%. This percentage would be even higher if malicious or malfunctioning network nodes were generating spurious packets. The ability of the attached processor to field network interrupts and drive the MAC device also decreases work the host must perform for communication processing.

Dedicated processor cycles. The attached processor has the sole responsibility of running the XTP implementation, and all of its cycles are dedicated to the communication system, since user processes are located on the host. As a result, there is no contention with other processes for processor cycles; thus, the overhead associated with context swaps and scheduling is avoided. Preliminary results of a UNIX kernel implementation of the same protocol show transport layer throughput to be about 36% lower than that of the off-host approach.

Specialized hardware. The architecture adopted for our SAFENET system was not able to fully utilize the DMA hardware offered on the attached processor. The architecture could benefit from both the support of DMA and specialized hardware for protocol functions such as the checksum. Additionally, the maturation of technology such that the attached processor and network adapter reside on the same board would very likely enhance performance. This is because such an architecture would reduce both contention for the backplane and the overhead of transfers across it.

Ideal operating system environment. The lightweight operating system environment of the attached processor is a better match to the performance demands of a communications system. Much of the lost performance of the in-kernel approach can be attributed to data copies mandated by the UNIX communication architecture [8]. In contrast, the attached processor operates as a dedicated protocol engine, and its operating system is better suited for such a purpose.

The application of the simple performance model to our architecture and comparison with observed performance indicates that our implementation allows us to make conclusions about the exploitation of our chosen architecture. The latency comparison indicates that our implementation incurs more constant overhead than was predicted given our design and its architectural components; thus, further optimization of our implementation may result in reduced latency. We do, however, seem to exploit all of our architecture's throughput potential. The model indicates that our implementation does indeed reach the throughput limits imposed by its surrounding architecture; hence, any significant improvement in throughput must arise from a modification of our fundamental architectural design rather than by optimizations within its architectural framework.

In short, we have shown that it is possible to achieve many of the promising benefits of off-host architectures. However, we have also demonstrated that certain design choices (e.g., the use of host-driven programmed I/O) can erode some of the potential benefits of the off-host approach.

Acknowledgments

The authors gratefully acknowledge the extensive support provided by Mr. John C. Fenton who developed our protocol processor software and documented its performance characteristics.

References

[1] B. J. Dempsey, J. C. Fenton, J. R. Michel, A. S. Waterman, and A. C. Weaver, "Tutorial on UVA SAFENET Lightweight Communications Architecture," Computer Science Technical Report Number TR-93-01, University of Virginia, January 1993.

[2] B. J. Dempsey, J. C. Fenton, J. R. Michel, A. S. Waterman, and A. C. Weaver, "Ada Binding Reference Manual—SAFENET Lightweight Application Services," Computer Science Technical Report TR-93-02, University of Virginia, January 1993.

[3] B. J. Dempsey, J. C. Fenton, J. R. Michel, A. S. Waterman, and A. C. Weaver, "SAFENET Internals," Department of Computer Science Technical Report Number TR-93-05, University of Virginia, January 1993.

[4] H. Kanakia and D. R. Cheriton, "The VMP Network Adapter Board (NAB): High-Performance Network Communication for Multiprocessors," *Proceedings of the SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pp. 175-187.

[5] J. R. Michel, "Performance Evaluation of an Off-Host Communications Architecture," M.S. Thesis, Department of Computer Science, University of Virginia, August 1993.

[6] MIL-STD-2204: *Survivable Adaptable Fiber Optic Embedded Network (SAFENET)*, United States Department of Defense, September, 1992.

[7] W. T. Strayer, B. J. Dempsey, and A. C. Weaver, *XTP: The Xpress Transfer Protocol*, Addison-Wesley, Reading, Massachusetts, 1992.

[8] A. S. Waterman, "A Comparison of Off-Host vs. In-Kernel Communications Architecture," M.S. Thesis, Department of Computer Science, University of Virginia (in preparation).

[9] R. W. Watson and S. A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," *ACM Transactions on Computer Systems*, Vol. 5, No. 2, May 1987.