# The *END*: A Network Adapter Design Tool

Atri Indiresan, Ashish Mehra, and Kang G. Shin

Real-time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
313-763-0391 (voice); 313-763-4617 (fax)
{*atri,ashish,kgshin*}@*eecs.umich.edu*

## ABSTRACT

End-to-end communication performance of distributed applications is determined largely by the underlying network technology and the operating system (including the protocols) running on end hosts. As network speeds increase, the communication performance bottleneck tends to shift to the hosts (clients and servers). This places critical requirements on the host communication subsystem, including the design of the network adapter and its interaction with the host communication software. In this paper, we explore network adapter design issues via an *Emulated Network Device* (*END*). *END* is a tool that interfaces to a real host and permits designers to evaluate adapter design by coupling a representative model of an adapter with the host communication software. This facilitates highly accurate performance evaluation of different adapter designs in a realistic setting by capturing host-adapter concurrency and the overheads associated with host-adapter interaction (such as interface handshake, data-transfer operations, and interrupts). Such a tool could be employed by device designers to compare and evaluate design alternatives before actually building a prototype.

We first present the architectural framework of *END* and how it interfaces to an actual host. Next, we demonstrate the utility of *END* as a network adapter design tool by conducting a case study of the Ancor VME CIM 250, an adapter for the Fiber Channel networking technology. This is achieved by first building a representative model of the device, called the *base model*. The base model is then modified to incorporate design changes that improve its performance. With these changes, network throughput increases by up to 50%, suggesting that similar improvements can be expected on the real device as well. We also discuss issues related to extensibility, portability, and limitations of *END* and suggest ways to overcome these limitations.

*Index terms* — Network adapter design, device emulation, communication subsystems, performance evaluation.

The work reported in this paper was supported in part by the Defense Advanced Research Projects Agency, monitored by the US Air Force Rome Laboratory under Grant F30602-95-1-0044, the National Science Foundation under Grant MIP-9203895 and the Office of Naval Research under Grant N00014-94-1-0229. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

# 1 Introduction

Recent advances in high-speed networking have made it possible to realize distributed applications with various requirements on end-to-end communication, including minimum bandwidth, bounded delays, delay jitter, and packet-loss rates. These applications often require high data-transfer throughput and low, bounded, end-to-end delays. A high-speed network by itself cannot guarantee high application-level throughput and/or bounded data-transfer delays. End-to-end communication performance depends not only on the underlying networking technology, but also on the end-host operating system as well as the interface between the host and the network. As network speeds increase, the performance bottleneck tends to shift to the end host, in particular to the hardware and software components of the host communication subsystem.

While communication software primarily comprises the protocols, the communication hardware at a host primarily comprises the network adapter and the interface between the host and the adapter. The design of the network adapter, and the division of functionality between the adapter and the host communication software, can have a significant impact on the performance delivered to applications. In order to design network adapters that integrate well with the host communication software and deliver good performance, one must study the impact of various design parameters in a *realistic* setting, i.e., when the network adapter is controlled and accessed by the communication software on the target host platform. The performance evaluation methodology employed must consider the hardware components and overheads involved (such as the system I/O bus, caches, device interrupts), and capture the hardware-software concurrency and dynamic host-adapter interaction without excessive intrusion.

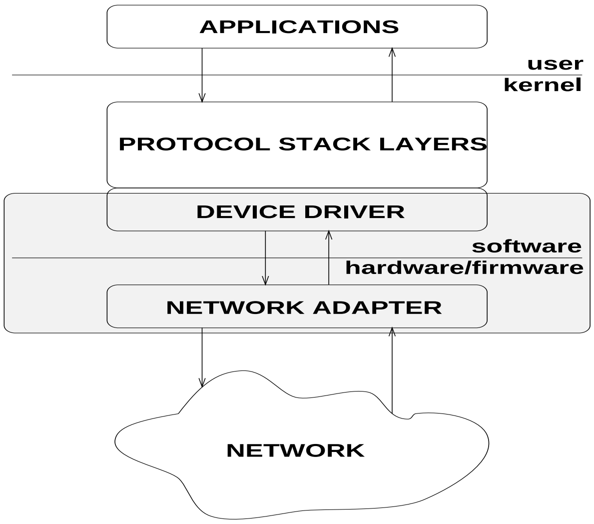
In this paper, we propose *network device emulation* as a mechanism to study the hardware-software interface for communication subsystems, and to help design network adapters that integrate well with the host operating system and applications. In particular, we present the *E*mulated *N*etwork *D*evice (*END*), a network adapter design tool that interfaces to a real communication protocol stack on the host via the system I/O bus. Most network interfaces are on the system's I/O bus instead of the private memory bus [1], so this configuration allows *END* to generate the same overheads as a real adapter. Further, *END* can emulate all the operations of a network adapter without interfacing to a real network. Instead, it uses a synthetic network model as a sink and source of traffic.

Designers can use *END* to experiment with network adapter design, including the partitioning of functionality between the host communication software and the hardware/firmware on the network adapter, before actually developing a prototype implementation of the adapter. The experimentation can be performed directly on the target host platform, with *END* running concurrently on its own processor, thus accounting for overheads and host architectural features that influence communication performance. *END* thus permits hardware-software *codesign*, where adapter design tradeoffs can be explored early in the design cycle.

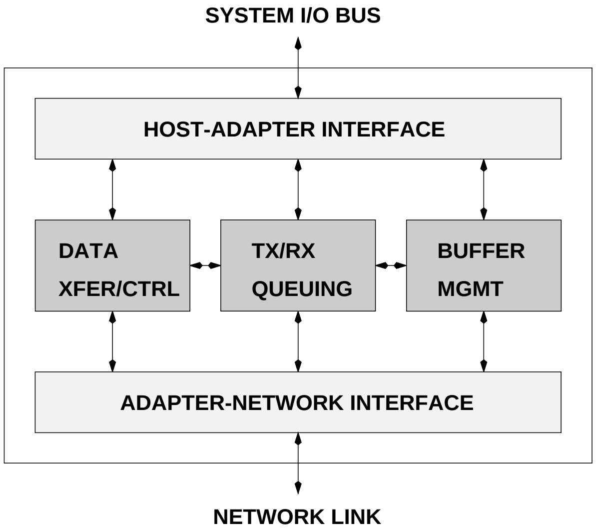
Despite these strengths, device emulation, as we envision it, has its limitations. To construct a detailed emulation environment, the designer or performance analyst must be intimately familiar with the hardware and software components involved in communication. Some of this burden can be alleviated by providing libraries of models that the designer can utilize to build the device emulator. Further, the host must be equipped with at least two processors so that the device emulator can execute concurrently with the host CPU. We do not believe this to be a problem because desktops equipped with at least two processors are now readily available. More importantly, the overheads associated with detailed device emulation may limit the performance of *END*, and hence the adapters it can emulate. However, this performance degradation can be reduced by using a processor faster than the host CPU. Overall, the ability to accurately capture both the network and the host behavior, and run experiments in real time, outweigh the disadvantages.

Device emulation has been used before to model source/sink behavior of I/O devices (disks, networks, sensors). Our contribution is in applying device emulation to capture the details of network adapter design. *END* can be used in two ways: (i) to design a new network adapter with special functionality (such as adding support for quality of service (QoS) in communication), and (ii) to explore design alternatives that improve the performance of an existing prototype. In a companion paper, we have used *END* to study QoS issues in adapter design [2] (option (i)). In this paper we demonstrate the utility of *END* to explore design alternatives (option (ii)) for an existing device by conducting a case study of the Ancor VME CIM 250, an adapter for the Fiber Channel networking technology. This is achieved by first building a representative model of the device, called the *base model*, and implementing it using *END*. The base model is then modified to incorporate design changes that improve communication performance. We show that with these changes, communication throughput increases by up to 50%, suggesting that similar improvements can be expected on the real device as well. We argue that the design of *END* is general and not biased towards particular types of adapters; as such, it can emulate any adapter within constraints of the emulation overheads incurred. We also discuss issues related to extensibility, portability and limitations of *END* and suggest ways to overcome these limitations.

The rest of the paper is organized as follows. Section 2 highlights some issues in network adapter design and compares the strengths and weaknesses of different evaluation techniques relative to device emulation. The architectural framework and our implementation of *END* is presented in Section 3. Section 4 presents the case study of the Ancor CIM 250 network adapter and its emulation using *END*. The utility of *END* as a network adapter design tool is illustrated in Section 5 by evaluating the performance of the CIM, identifying performance bottlenecks, and evaluating design improvements using *END*. Section 6 discusses the limitations of our approach, and how they may be addressed. We also discuss the portability and extensibility of *END* to other host architectures, network adapters, and I/O devices. Section 7 discusses related work, and Section 8 concludes with a summary of our results and future work.




(a) Host communication subsystem




(b) Generic network adapter architecture

Figure 1: Focus of this paper.

## 2 Network Adapter Design

In this section we discuss various issues involved in network adapter design. Our goal is to identify the architectural components that determine the mechanics and performance of data transfer between the host and the network via the network adapter. The architectural framework of *END* is based on these architectural components. We also compare and contrast device emulation with other performance evaluation techniques.

### 2.1 Factors Affecting Communication Performance

End-to-end communication performance depends on a variety of factors, some attributed to performance bottlenecks in the network and others attributed to performance bottlenecks in the end hosts. Factors affecting performance in the network (such as underlying network technology, network congestion, etc.) are beyond the scope of this paper. Within the end hosts, communication performance is largely determined by the capacity of the software and hardware components to move data between applications and the network (see Figure 1(a)). Data transfer involves traversing a protocol stack, moving data between the host memory and the network adapter, and between the network adapter and the network itself. The software components that affect data transfer performance include the protocol stack, scheduling and synchronization mechanisms in the host operating system, synchronization and scheduling mechanisms on the adapter, and the adapter firmware. The hardware components that influence data transfer performance include the host CPU speed, the host-adapter interface, the host-adapter data transfer bandwidth, and the network bandwidth. For a given network and end host, the network adapter should be designed such that its hardware and software components do not limit communication performance.

Our focus in this paper is on the design of the network adapter and how it interacts with the host communication software through a device driver. Figure 1(b) illustrates the generic architecture of a typical network adapter. There are five basic components that comprise this architecture: the host-adapter interface, the data transfer control module, the transmission/reception queuing module, the buffer management module, and the adapter-network interface. Transmission and reception to/from host memory is accomplished via interaction between all of these components, as discussed below. Note that most network adapters are accessed by the host via the system I/O bus [1]. Network adapters can vary significantly in complexity depending on their desired performance goals and the underlying network technology. To manage this complexity in an efficient and flexible manner, adapters typically employ one or more general-purpose microprocessors (say one for transmission and one for reception) and some custom hardware under control of the adapter firmware.

**Host-Adapter Interface:** The host-adapter interface is typically in the form of a device driver running on the host, and a companion "host driver" on the adapter. The two drivers exchange information via command-response mailboxes or queues across the system I/O bus, and may synchronize their operations either via interrupts, polling, or some combination of the two (`reference?`). Interrupts allow immediate synchronization, at the cost of increased host overhead of handling an interrupt and the resultant context switches and cache misses. Polling eliminates this overhead, but, depending on the polling frequency, may reduce the responsiveness of the interface significantly. To minimize overheads, the interface must be as simple as possible. Example operations exported by the host-adapter interface include initiation of packet transmission, initiation of packet reception, and examining the status of pending transmissions.

**Adapter Internals:** Referring to Figure 1(b), three modules provide the internal functionality of the adapter: data transfer control, transmission/reception queuing, and buffer management.

*Data Transfer Control:* Once packet transmission or reception is initiated via the host-adapter interface, data may be transferred between host memory and adapter buffers via DMA or programmed I/O (PIO). PIO has no startup latency, but requires CPU cycles for the entire duration of transfer, potentially limiting concurrency. With DMA, the host CPU is free to perform useful computation once it sets up the DMA correctly. The DMA startup latency could be significant, and data transfer may have to be suspended several times due to simultaneous memory accesses by the host CPU. The choice between DMA and PIO is a function of factors such as the bandwidth of the system I/O bus and the size of the data transfer. A hybrid approach would be to use PIO for small packets and DMA for larger ones.

*Tx/Rx Queuing:* Once packet transmission is initiated, or a packet arrives from the network, the adapter must queue the packet until it can either be injected into the network (transmission) or received by the host (reception). The actual queuing policy and mechanisms employed depend on the expected traffic mix. For example, for best-effort adapters it may suffice to provide simple first-in-first-out (FIFO) queuing of packets with deep pipelining of operations on the adapter. This delivers high throughput by exploiting the overlap between, say, DMA of one packet and transmission of another, and keeping the overheads of queuing low. More sophisticated queuing and scheduling algorithms may be required when providing per-connection QoS guarantees.

Pipelining allows the host and adapter to operate independently, rather than in lockstep. While this increases data transfer throughput, it reduces host control over the order of transmission of packets. While this does not affect transmission of best-effort traffic, it poses difficulties when transporting traffic with QoS guarantees. Note that the host exercises complete control over packet transmission order when the pipeline depth (number of packets outstanding on the adapter) is one [3].

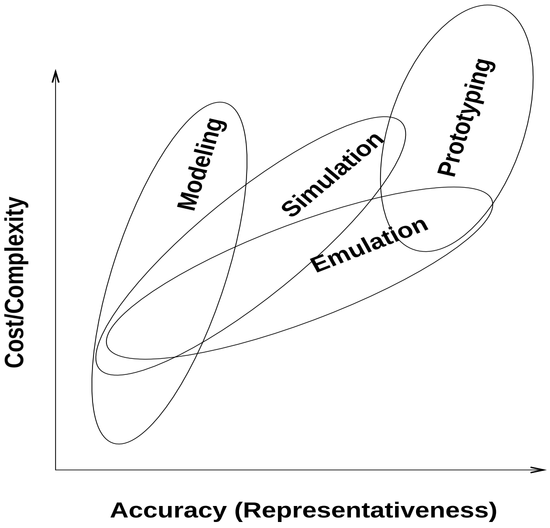
*Buffer Management:* The adapter may need to manage buffers as a staging area for packet transmission and reception. The buffers themselves may reside either in adapter memory or in host memory. In the former case, the adapter must provide buffer management policies, such as reserving buffers for connections with QoS guarantees, as well as handle buffer overload conditions correctly. The adapter may also exercise partial control over packet buffers on the host, and, if buffer space on the adapter is low, it could potentially delay transfers between the host and adapter, until just before packet transmission. Buffer management plays an important role on packet reception as well. Expensive copying overheads can be avoided if the adapter manages buffers such that packet headers are stored separate from packet data [4].

**Adapter-Network Interface:** At the level closest to the network, the adapter transmits (receives) data to (from) the network medium by copying data from (to) its buffers to (from) the network under control of the medium access protocol of the attached network. As before, it can use either PIO or custom DMA hardware for this operation. If packet buffers reside in host memory, the adapter needs to inform the host on completion of packet transmission so that the host buffer can be reused. This notification is not required if packet buffers reside on the adapter and a similar notification was issued after copying the packet to the adapter. Similarly, on packet reception from the network, it must stage this data into on-board buffers and subsequently transfer the data to the host.

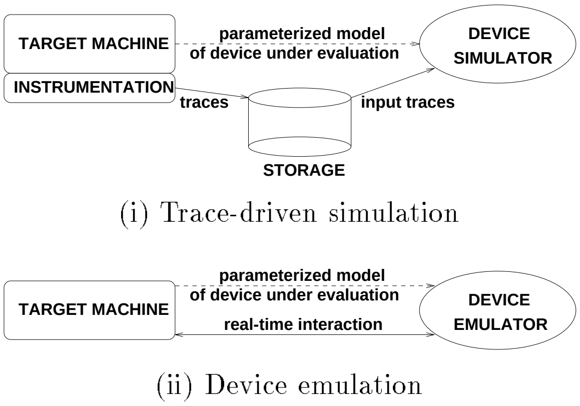
## 2.2 Performance Evaluation Techniques

In order to design a network adapter that meets desired performance requirements, one must study design alternatives in a realistic setting, i.e., when the network adapter interacts with the communication software on the target host platform. Several techniques may be used to design and evaluate network devices (I/O devices in general): mathematical modeling, simulation, emulation and prototyping. Figure 2(a) shows the cost-accuracy tradeoffs of these techniques, as explained below. The $x$-axis represents the accuracy of the technique employed, and the $y$-axis represents the cost/complexity of employing the technique.

*Mathematical modeling* is typically employed to study the queuing behavior of network traffic. Though mathematical models are relatively inexpensive to develop for overly-simplified systems, they rarely account for system overheads encountered in practice (such as interrupt handling and context switches). More detailed models capturing concurrency, contention, and dynamic component interac-




(a) Cost-accuracy tradeoff




(b) Trace-driven simulation vs device emulation

Figure 2: Performance evaluation techniques and tradeoffs.

tion in real systems rapidly become intractable.

Another technique is (trace-driven) *simulation*, which has several advantages [5]. Since a simulator is built in software, it can be readily modified and augmented to test new features and interfaces. Simulators are usually easier and cheaper to build than real systems. They can model "ideal" systems that are impossible to build, e.g., an infinitely fast network. However, a simulator is typically an artificial device, i.e., no real system components are involved, which has been accurately parameterized via performance measurements. Exceptions to this do exist in approaches that execute actual software under control of the simulator, as in [6]. However, while sufficient to study the network performance of communication protocols, such approaches are not applicable when hardware components (such as the system I/O bus, caches, device interrupts) must also be considered and hardware–software concurrency and dynamic interaction captured in the evaluation. Further, simulation is typically much slower than the execution of real systems. Increasing the accuracy of the simulation model increases its cost while slowing it down even further.

*Prototyping* a device and interfacing to higher software layers in the operating system is time-consuming and expensive. While prototypes can be highly accurate (Figure 2(a)), they are not easily modifiable. The on-board firmware may be modified to study different design options [4], or one may employ programmable adapters [7], but the internal hardware architecture is typically impossible to modify without developing a new prototype. More importantly, the hardware engineers designing network adapters are often far removed from the concerns of those writing the communication software, and vice versa, resulting in a design poorly integrated with the host operating system.

### 2.3 The Case for Device Emulation

As Figure 2(a) illustrates, device emulation provides reasonable accuracy at an acceptable cost/complexity. Device emulation shares the advantages of simulation in that it is a flexible technique that allows rapid evaluation of various design alternatives. In contrast to a simulator, an emulator is a module that interfaces with a real host (Figure 2(b):(ii)), giving the latter the impression that it is interacting with the actual subsystem being emulated in real-time. On the other hand, Figure 2(b):(i) illustrates the evaluation procedure for trace-driven simulation. The target machine (for the device being designed/evaluated) is instrumented to generate run-time traces that are later fed as input to the device simulator. Not only must the host software be instrumented to generate sufficiently detailed traces, the instrumentation code may be intrusive enough to disturb the timing (and hence the sequence) of important events. Further, the traces thus generated must reside on stable storage before being fed to the device simulator. Device emulation does not require that the target machine be instrumented, as long as it has necessary driver software to communicate with the device emulator. Note that both simulation and emulation require construction of accurate parameterized device models that are typically derived from existing devices [8].

In our case, *END* emulates a network adapter and interfaces with the target host, giving the impression that the host communication software is communicating with a real network. This has several significant advantages. Interfacing a device emulator to the target host allows adapter design tradeoffs to be evaluated in the presence of applications, operating system overheads, interrupts, etc. This helps identify design limitations and bottlenecks early in the design cycle. Further, since device emulation is carried out on the target platform, it allows development and testing of host operating system software that interfaces to the device, and rapid integration of the actual hardware device when it becomes available.

## 3 *END* Emulation Architecture

In this section, we present the emulation framework for *END* in the light of the architectural components described in Section 2. We first define the functional interface that *END* must export to the host, followed by description of the components constituting *END*. We then describe our implementation of *END* and how it operates in response to host commands.

### 3.1 Host View of Network

A host interacts with the network via the network adapter, viewing it as a sink for data transmission, and a source for data reception. Interaction with the adapter involves exchange of commands and data, and the timing of these events is determined by the characteristics of the network, as well as prevailing traffic conditions. In order to accurately emulate a network adapter, *END* must export to the host the same "view" as, and have performance characteristics similar to, that of a real adapter.

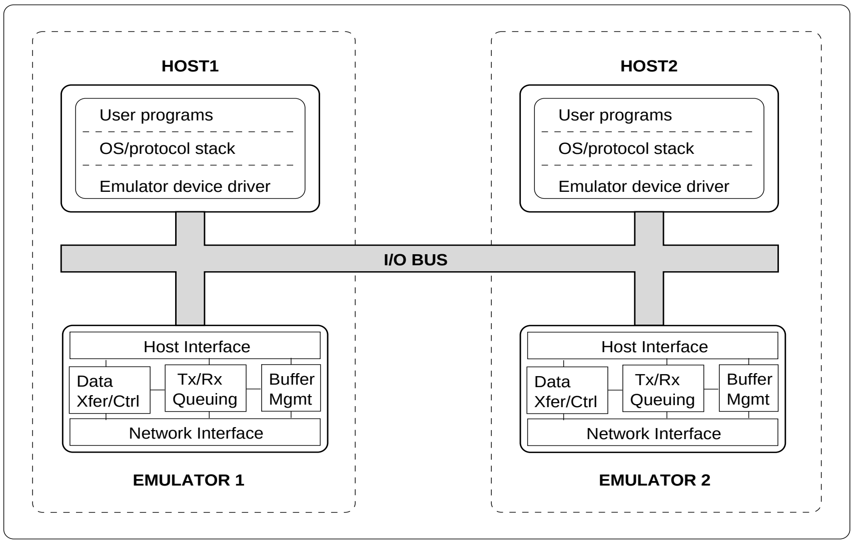



Figure 3: *END*-based device emulation architecture.

That is, it must be integrated with the host system using the same functional interface.

*END* interacts with a device driver on the host the same way as a real device; handling commands, issuing responses, and synchronizing with the host via interrupts or polling. Typically, when an application transmits data, the communication subsystem acknowledges successful transmission after a delay determined by the size of data, and the system and network load, and allows the application to reuse the data buffer. As long as the application does not expect an acknowledgment from its peer receiving application, all it sees is data being transmitted at a certain rate.

This scenario is applicable to many applications, like servers, that generate a largely unidirectional flow of data. A transmitting application will not be able to distinguish between *END* and a real network as long as *END* captures this timing behavior. Note that this suggests the use of one packet transmission time as the basic measure of network performance. For data reception, *END* can generate incoming traffic based either on stochastic models or traces of real network traffic. In case of protocols that require two-way traffic (e.g. TCP/IP), *END* may have a module that responds to outgoing traffic in the same way as a peer application on another node. Otherwise, genuine two-way traffic may be generated (as described in Section 3.3).

### 3.2 Emulator Components

Figure 3 shows our emulator-based architecture for studying issues in adapter design. Each network "node" corresponds to a host processor board and an emulator processor board (running *END*) on the same (system) I/O bus. As discussed in Section 3.1, certain experiments may be performed using a single node. Two-way communication is accomplished via two or more such nodes on the same I/O bus, with the I/O bus serving as the communication medium.

**Host Node:** The host node is precisely the target host. Communicating applications send and receive data via the protocol stack in the operating system. At the bottom of the protocol stack is the emulator device driver for the target network adapter. Though this driver communicates with *END*, it handles the full functionality of a real device driver; i.e., it implements the actual host-adapter interface. This allows a developer to implement and test a complete driver even while the network adapter is being designed or implemented, and also ensures that observed performance of the driver is comparable for the emulator and the real adapter.

**Network Adapter:** The network adapter node is a general-purpose processor board, with a CPU and memory. Since its main function is to handle data transmission and reception efficiently, it needs at most a minimal executive to provide process management and handle interrupts.

Pure packet transmission can be modeled as a transmission-complete notification after a suitable delay. This is quite accurate, except that since actual data is not transferred, the host CPU does not suffer the overhead of cycle stealing during DMA data transfers. However, this feature is an advantage in that the system may be configured for arbitrary network speeds, allowing it to evaluate the efficacy of different host and adapter algorithms at different network speeds.

There are two aspects to packet reception: the model used to generate packet arrivals, and the mechanism used to generate packet headers and data. Packet arrivals can be generated either using a stochastic model, or using two-way communication as discussed in Section 3.3 below. For the purposes of studying adapter performance, the actual data content is unimportant. However since received data must traverse the protocol stack, packet headers must be meaningful.

**Time Services:** *END* needs time services to manage the time abstraction. This requires an event manager to register desired delays, and notify the requesting node when the time expires, either with an interrupt or simply setting a completion flag that the adapter can poll.

### 3.3 Two-Way Communication Across The "Network"

For two-way communication, we utilize the system I/O bus as the "network", with data being transferred from one adapter node to another (see Figure 3). The I/O bus limits the speeds and the number of nodes that can be emulated, since bus contention increases with the number of emulated nodes. However, this approach is useful since it can be used to verify the correctness of protocols and perform relative performance comparisons rather than evaluate absolute performance. If the actual data transmitted is not important, it suffices to copy only the packet headers as long as one accounts for the time to transfer the packet data. If header lengths are small compared to the data, I/O bus contention can be reduced substantially; this permits evaluation of adapter performance for a large range of network speeds. We note that at least one other study has evaluated protocol performance using the I/O bus of a multiprocessor host as a high-speed network [9].

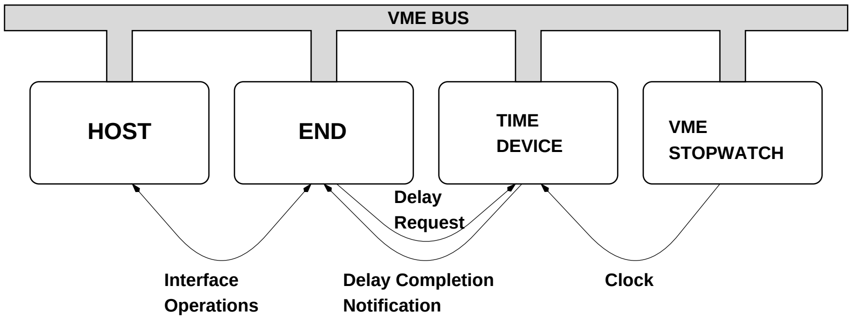



Figure 4: Emulator system configuration.

### 3.4 Implementation

*END*'s environment is determined by the platform on which it is implemented. In particular, the host processor(s), the I/O bus and the operating system are typically fixed. While *END* can accurately represent network performance on this platform, it can also serve as an indication of performance on a different platform as long as the relative speeds of the target host and network are maintained. However, to make *END* generally applicable, portability is an important consideration. While the host-adapter interface would probably change from one platform to another, as would the availability and mechanisms for DMA, interrupts, clocks etc., most of the adapter services themselves will be fully portable – both to other emulator platforms, as well as to real target adapters.

Figure 4 shows the configuration of our implementation of one emulator node. The processor boards of each emulated node communicate with each other via a VMEbus. We have completed the implementation of the emulation of data transmission. Data reception will be similar, with some differences highlighted in the description of the implementation which follows.

**Device Emulator Core:** The device emulator is a minimal executive with support for processes, interrupt handling, and semaphores. It is essentially a cyclic executive that polls the host for commands, performs the operations, and notifies the host of completion. It has several sub-components that may be configured and parameterized based on the kind of adapter behavior to be emulated. Table 1 summarizes the features that are described in detail below.

**Host–emulator interface:** The host–emulator interface uses *command/response queues* to exchange information. The queues are implemented as circular FIFOs. The exact interface is determined by the actual commands and responses, and whether synchronization uses interrupts, polling, or some combination of them.

**Data transfer control:** Data transfer may be via either DMA or Programmed I/O (PIO). In our prototype, there is no actual data transfer[1], we supply generic time models for each case. The DMA has a fixed, non-zero, startup time for each block of memory, and an incremental cost per byte transferred, while the PIO has little or no startup cost, but, typically, a higher incremental cost. In addition, for PIO, the emulator CPU will have to be idled for the duration of the PIO, and not allowed to perform

[1]There will be real data transfer (of headers) for two-way traffic, or for a pure reception model.

| Component | Configuration | Options/Parameters |
|---|---|---|
| Host/Adapter Interface | number of interfaces | relative priority of interfaces |
| | types of interfaces | mailbox, FIFO |
| | synchronization | poll, interrupt |
| Data Transfer and Control | DMA | fixed delay per block, delay per byte |
| | program I/O | delay per byte |
| Tx/Rx Queuing | number of stages | |
| | number of queues | FIFO/EDD, relative queue priority |
| Buffer Management | allocation/freeing | from a shared pool |
| | overload behavior | drop, block connection, priority based |
| Network Interface | delay model | setup cost, delay per byte |
| | synchronization | poll, interrupt |

Table 1: Configurable components of *END*.

any other functions, since it is supposed to be "busy" copying data.

**Transmission and reception queuing:** Packets can be queued for various operations during transmission and reception. These queues are determined by the data type and the operation, and queuing policies are selected independently for each queue. Two queuing policies have been implemented, FIFO and Earliest Due Date (EDD), and the selection of queues may be based on their relative priorities, or round robin.

**Buffer management:** Buffers are allocated from a shared pool on the host and/or adapter. They may also be reserved on a per-connection basis [3].

**Network interface:** The network interface for data transmission is a pure delay model. Data transmission may be synchronous (the emulator waits until the transmission delay elapses), or asynchronous (the emulator performs other operations during the "data transmission" delay). The delay services are provided by the *time device*, described below.

**Time Device and VME StopWatch:** The time device is a processor board without any OS support. It is a cyclic executive that simply reads delay requests from the emulator node and inserts them into a priority queue ordered by their completion times. Delay requests and responses are exchanged via a mailbox. Each delay request has a unique identifier, a completion time, resources used and priority, and a flag indicating whether the emulator needs an interrupt in addition to the completion notification in its response mailbox. If two timed activities contend for a shared resource, the lower priority activity waits for the completion of the higher priority activity, or is preempted by it and the completion time is adjusted accordingly. A *completion function* associated with each request is executed when the emulator is notified of the expiry of the time interval.

Time measurements are provided by the VME StopWatch [10][2]. It has a high resolution ($25ns$), 24-bit timer, that wraps around about every $0.4s$. Since the time device continuously polls the timer to determine when an interval has elapsed, it also detects timer wraparound, and ensures that elapsed

[2] Many modern processors have high resolution clocks and will not need such external timing support

time is measured correctly.

### 3.5 Emulator Operation

An adapter may be thought of as a multi-stage device. Each stage corresponds to an activity like data movement (e.g. copying, transmission) or transformation (e.g., encoding, segmentation), and requires certain resources which are typically shared (e.g., CPU, buffers, busses, network links). Messages on the device pass through one or more stages, with a certain delay at each stage. Activities in different stages may proceed concurrently as long as they do not contend for the same resources. If a stage has more than one of its required resources, multiple messages may pass through that stage at once.

Operation of the emulator is illustrated with the example of the transmission request procedure shown in Figure 5 (numbers in the following description correspond to line numbers in the figure). Each stage $i$ has associated with it an initial function ($\texttt{stage}_i\texttt{\_do}$ (8)) and a completion function ($\texttt{stage}_i\texttt{\_done}$ (18)). It has one or more queues (with different queuing disciplines) to enqueue messages or operations (5, 12, 21), and selection of a particular queue typically depends on the message type. Delay at each stage (15) depends on the message, and it may be synchronous or asynchronous (e.g. for short messages, an adapter may choose to use PIO to copy data, thus occupying the CPU, and preventing other operations from proceeding; for larger messages, it may initiate a DMA and concurrently proceed with other activities). When the activity of a stage has completed, $\texttt{stage}_i\texttt{\_done}$ is invoked. This passes on the results of the current stage to the next stage (21–22), and then processes the next object waiting for service at that stage. If it is the last stage (26), it sends a transmit acknowledge indication to the host.

This structure is very general, and easily accommodates a variety of resources and services including CPU, busses, buffers and data structures. It also captures contention between different activities that share resources. For instance, transmission and reception use the same resources, though in the reverse order. This flexibility is needed since different network adapters might have different services and components, and also because *END* may be extended to model other I/O devices with significantly different architectures (disk controllers, frame grabbers). To define the operation of an adapter, one needs to determine the stages, the resources, and instantiate the queuing and delay semantics for these stages, resources and data types.

## 4 A Case Study: Emulating the Ancor CIM 250 using *END*

In this section, we conduct a case study of the Ancor CIM 250 and emulate it using *END*. Recall that emulating an existing adapter represents one way of using *END*. We have also used *END* to evaluate the impact of various adapter design configurations on provision of QoS guarantees [2]. In earlier work on implementing real-time communication services [11], we evaluated the Ancor CIM 250 [12] (or CIM) extensively. Our experiments revealed several performance bottlenecks and potential for

```
TransmitRequestProcedure(message)
begin
     // Each stage could have multiple queues.  Selection of the
     // queue and queuing policy is determined by the message type.
     q_insert(queue_1, message.deadline, message.data, message.type)
     stage_1_do()
end

stage_i_do()
begin
     if (stage_i_free) // there is a free resource for this stage
          stage_i_free--
          dequeue highest priority stage_i message
          event.delay = stage_i_delay(message) // compute delay
          event.end_function = stage_i_done
          delay(event, stage_i_sync(message)) // synchronous delay or not?
     endif
end

stage_i_done()
begin
     // Message has completed stage_i.  Insert it into queue_{i+1}
     q_insert(queue_{i+1}, message.deadline, message.data, message.type)
     stage_{i+1}_do() // Initiate stage_{i+1} function
     stage_i_free++ // Free stage_i resource
     stage_i_do() // process next message queued at stage_i
end

stage_n_done()
begin
     // Last stage -- send acknowledgment, and process next message
     send XMIT_ACK to host
     stage_n_free++
     stage_n_do()
end
```

Figure 5: Transmission emulation for $n$-stage adapter operation.

| Phase | Device driver | | VME CIM 250 |
|---|---|---|---|
| Initialization | send `write` | ⟶ | start transmit sequence |
| | | ⟵ | send `write ack` |
| | | ⟵ | Interrupt Request |
| DMA | send `address list` | ⟶ | DMA list of `address` commands into local memory |
| | | | DMA data for list into local memory |
| | | ⟵ | send `address ack` |
| | | ⟵ | Interrupt Request |
| Fiber Channel | | | Transfer data to Fiber Channel network interface |
| | | ⟵ | send `transmit ack` |
| | | ⟵ | Interrupt Request |

Figure 6: Host–adapter interaction for message transmission.

improvement, making the CIM a suitable candidate for this study. The performance characteristics of the CIM have also been studied in detail in [13]; we have used some of these results in this study.

Our emulation methodology is to first construct a *base model* that is representative of the device (in this case the CIM). A model is said to be representative if it performs the same functions, and has performance equivalent to that of the target device. Once the base model is constructed and implemented using *END*, design changes can be directly incorporated in *END*. Performance changes observed in the modified model will then be similar to those due to similar design modifications in the real adapter. As illustrated below, constructing a representative model of an existing adapter requires systematic "black box" performance analysis of the device. Since *END* has been designed independently of the CIM, its design is not biased towards that of the CIM; this study therefore serves as a fair test of *END*'s ability to model real adapters. We now describe the CIM, its data transmit path, and how we emulate its transmission behavior using *END*.

## 4.1 Ancor CIM 250

The Ancor VME CIM 250 [12] Communications Interface Module (CIM) is a network adapter for ANSI Fibre Channel 3.0 [14] networks. Besides communication interface hardware (an IBM OLC266 Fiber Optic link card), the CIM has an NEC 32 MHz 70236 I/O processor, 8MB DRAM, independent DMA controllers for data movement, and VMEbus interface logic. The CIM communicates with the host using command/response FIFOs; it supports commands to initiate read/write operations, set up DMA between the host and adapter, and signal the completion of DMA and network transmission. The CIM polls the command FIFO for commands from the host driver, writes responses to the host via the response FIFO, and then issues an interrupt to notify the host of the response.

A simplified sequence of the events involved in CIM transmission is shown in Figure 6. During the *initialization* phase, the host issues a `write` command, and the adapter responds with a `write ack`. In the *DMA* phase, the host provides the addresses and lengths of buffers to be transmitted via the `address`, `address end` and/or `address list` commands, and receives an `address ack` on completion of the DMA. Next, the *Fiber Channel (FC)* phase starts transmitting the data, and sends a `transmit`

| $\mathbf{Stage}_i$ | Trigger | $\mathbf{Stage}_i$**_do** | $\mathbf{Stage}_i$**_done** |
|---|---|---|---|
| Initialization | `write` | Init data structures<br>send `address ack`<br>interrupt host | |
| DMA | `address list` | read address list<br>enqueue segments for DMA<br>`StartDmaXfer()`<br>{ dequeue segment;<br>async delay for DMA} | if (last segment copied)<br>enqueue message for xmit<br>`Xmit_do()`<br>send `address ack`<br>interrupt host<br>`StartDmaXfer()` |
| Xmit<br>(Fiber Channel) | `DMA_done` | dequeue message<br>async delay for transmit | send `transmit ack`<br>`Xmit_do()` |

Figure 7: Outline of *END* model for transmission with CIM .

`ack` to the host on completion. The CIM interrupts the host each time it sends a response. Reception is triggered by the arrival of data on the CIM, and has an initialization and DMA phase (see [12, 13] for details). Note that the host may issue additional `write` commands before a previous one completes. This increases concurrency since different CIM and host operations may now occur simultaneously. We call the number of incomplete `write` operations the *pipeline depth* on the CIM.

## 4.2 CIM Functional Model

The *END* model and the CIM appear functionally identical to the host if both have the same commands and responses. The main loop of *END* polls for commands from the host, executes them and sends appropriate responses to the host. The host–*END* interface is almost identical to the host–CIM interface; it issues exactly the same commands, gets the same responses, and interrupts the host at the same priority as the CIM. The only difference in the interface is that the CIM has hardware support for command/response FIFOs, and so the host writes commands to (and reads responses from) a fixed address in its I/O space. In contrast, *END* writes commands to (and read responses from) circular FIFOs in *END*'s memory. The host device driver is almost identical in both cases – less than 30 lines of code changed in the device driver (out of over 2000 lines of C code).

Figure 7 shows an outline of the *END* model for the transmit operations of the CIM using the structure presented in Figure 5. Functions $\texttt{stage}_i\texttt{\_do()}$ of *END* are triggered by events such as a command from the host or the completion of the previous stage. $\texttt{stage}_i\texttt{\_done()}$ represents the action taken after the completion of the time delay (if any) representing the action of that stage. These functions are adequate to completely define the host–*END* interface, i.e., this model will interact correctly with the host device driver. To complete the model, it is necessary to ensure that it executes with the same delays as the CIM.

## 4.3 CIM Performance Emulation

To accurately emulate the CIM using *END*, it is necessary to compute the delay functions for the DMA and FC phases. We gathered information for these functions from three sources: existing literature [13],

| Symbol | Description | Value |
|---|---|---|
| $\mathcal{M}_{overhead}$ | fixed cost per message | 1280 $\mu s$ |
| $\mathcal{D}_{segsize}$ | DMA segment size | 256 bytes |
| $\mathcal{D}_{overhead}$ | DMA overhead per segment | 1.76 $\mu s$ |
| $\mathcal{S}_{small}$ | size of small messages | 4096 bytes |
| $\mathcal{D}_{byte}^{small}$ | DMA time per byte (small messages) | 0.42 $\mu s$ |
| $\mathcal{D}_{byte}^{large}$ | DMA time per byte (large messages) | 0.31 $\mu s$ |
| $\mathcal{F}_{framesize}$ | FC frame size | 2048 bytes |
| $\mathcal{F}_{overhead}$ | FC message overhead | 1350 $\mu s$ |
| $\mathcal{F}_{framedelay}$ | Transmit time per FC frame | 79.75 $\mu s$ |

Table 2: Important system parameters

our own measurements [11], and conversations with staff at Ancor Communications, Inc. Lin *et al.* [13] performed extensive experiments on the CIM (using VME probes) and characterized the delays of various components. As a first approximation for our model, these delays deliver a performance significantly different from our observations. This is attributed to the fact that the measurements in [13] were made on a completely different platform with a different CPU, memory, protocol stack and operating system. Accordingly, we have utilized only the measurements corresponding to operations that are strictly internal to the CIM, and hence, not affected by the change in platform. These include the delays for setting up the DMA and the FC phase. The delays during DMA, and other overhead, are computed from measurements made on our platform, as described in the next section.

### 4.3.1 Arriving at an Accurate Base Model

We construct an accurate base model for the CIM by computing $\mathcal{T}_{DMA}$, the time to DMA a message (the DMA phase), and $\mathcal{T}_{FC}$, the time for the FC phase. While $\mathcal{T}_{FC}$ is inferred from the results reported in [13], $\mathcal{T}_{DMA}$ must be computed via measurements since it is platform-dependent. $\mathcal{T}_{FC}$ is given by

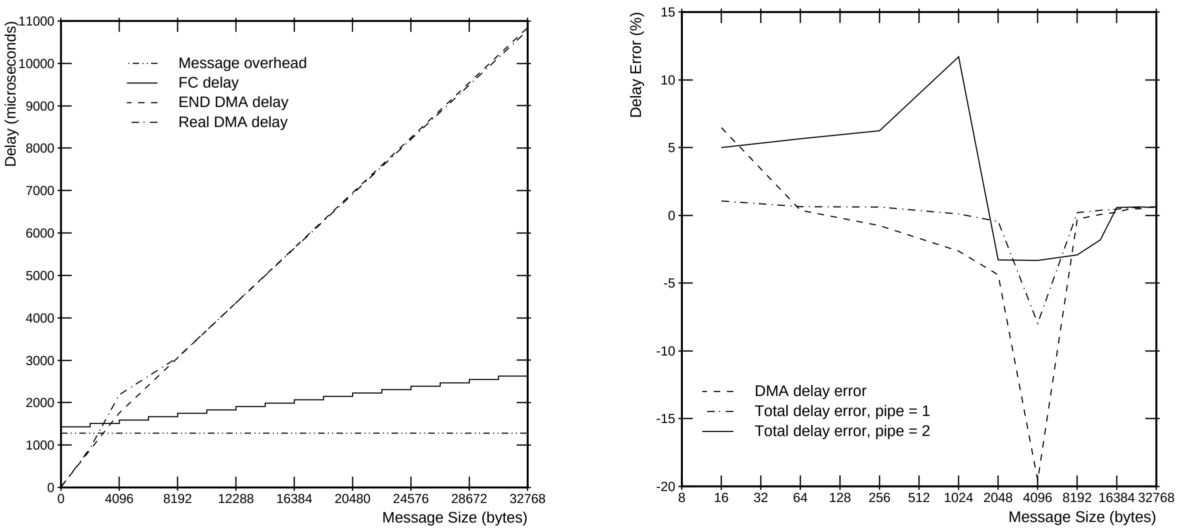
$$\mathcal{T}_{FC}(\mathcal{S}) \quad = \quad \mathcal{F}_{overhead} + \lceil \frac{\mathcal{S}}{\mathcal{F}_{framesize}} \rceil \times \mathcal{F}_{framedelay}, \tag{4.1}$$

where $\mathcal{S}$ is the message size, $\mathcal{F}_{overhead}$ is the per-message overhead for the FC phase and $\mathcal{F}_{framedelay}$ is the delay for each FC frame of size $\mathcal{F}_{framesize}$. Given the values in Table 2, $\mathcal{T}_{FC}$ can be computed for a given message $\mathcal{S}$.

We compute $\mathcal{T}_{DMA}$ as a function of $\mathcal{S}$ by experimentally measuring $\mathcal{T}_{m(pipe=1)}$, the message transmission time on the CIM with a pipeline depth of 1, as follows:

$$\mathcal{T}_{m(pipe=1)}(\mathcal{S}) \quad = \quad \mathcal{M}_{overhead} + \mathcal{T}_{DMA}(\mathcal{S}) + \mathcal{T}_{FC}(\mathcal{S}), \tag{4.2}$$

where $\mathcal{M}_{overhead}$ is the message overhead, which corresponds to the cost of generating the message, traversing a minimal protocol stack, and crossing the host–CIM interface for the entire message (but excluding the costs of the DMA and FC phases). Since the DMA delay is emulated by *END*, we must measure $\mathcal{M}_{overhead}$ on *END*. Using the same protocol stack, we ran experiments transmitting




(a) Breakup of message costs
($x$-axis has linear scale)

(b) Error in model
($x$-axis has log scale)

Figure 8: Sources of delay in the model and their accuracy.

12K messages of various sizes on *END* and measured the average message transmission time for each message size, with the DMA and FC delay functions set to zero. Table 2 lists the measured value for $\mathcal{M}_{overhead}$ as $1280\mu s$. To verify that this was reasonable, we repeated the same experiments on the CIM using small (44 bytes including headers) messages and measured the average time till the `address ack` to be $1240\mu s$, which yields an $\mathcal{M}_{overhead}$ value of $\approx 1220\mu s$ after accounting for the DMA delay.

$\mathcal{T}_{DMA}$ can now be computed as the only unknown in Eq. 4.2. One would expect $\mathcal{T}_{DMA}$ to be a linear function of $\mathcal{S}$. However, our computed value for $\mathcal{T}_{DMA}$ revealed that this is not the case for the CIM; the incremental delay is a non-linear function of $\mathcal{S}$, being much higher for messages smaller than $\mathcal{S}_{small}$ (listed in Table 2), and lower for larger messages (see Figure 8(a), also expanded in Figure 10(a)). As an approximation, we use a delay function that is the combination of two linear functions (Figure 8(a)):

$$\mathcal{T}_{DMA}(\mathcal{S}) = \begin{cases} \lceil \frac{\mathcal{S}}{\mathcal{D}_{segsize}} \rceil \times \mathcal{D}_{overhead} + \mathcal{S} \times \mathcal{D}_{byte}^{small} & \text{if } \mathcal{S} < \mathcal{S}_{small} \\ \lceil \frac{\mathcal{S}}{\mathcal{D}_{segsize}} \rceil \times \mathcal{D}_{overhead} + \mathcal{S}_{small} \times \mathcal{D}_{byte}^{small} & \\ \qquad + (\mathcal{S} - \mathcal{S}_{small}) \times \mathcal{D}_{byte}^{large} & \text{otherwise,} \end{cases} \tag{4.3}$$

where $\mathcal{D}_{overhead}$ is the fixed overhead per DMA segment, $\mathcal{D}_{segsize}$ is the DMA segment size used by the CIM, and $\mathcal{D}_{byte}^{small}$ and $\mathcal{D}_{byte}^{large}$ are the incremental costs per byte obtained empirically. The values corresponding to these parameters are listed in Table 2. As can be seen from Figure 8(b), the above approximation works well except for a small region around $\mathcal{S} = 4K$ bytes, at which point the DMA performance of the CIM diverges significantly.

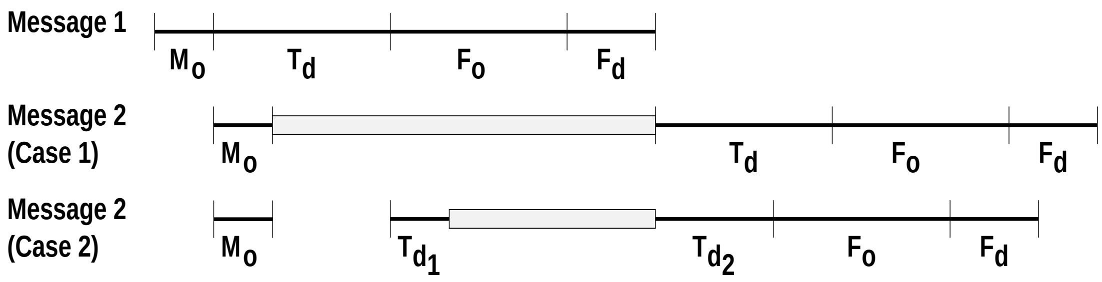



Figure 9: Inter-message overlap during CIM transmissions.

### 4.3.2 Accounting for Concurrency and Contention

The model described above is adequate when there is no contention for resources on the CIM. This is a reasonable assumption as long as the CIM handles only one transmission or reception at a time. In general, this is not true since the host may send several `write` requests to the CIM before completion of earlier requests, or have incoming data. In this case, for example, $\mathcal{T}_{DMA}$ of one message may (partially) overlap $\mathcal{M}_{overhead}$ of another, permitting some concurrency. However, not all operations can occur concurrently. Both the DMA and FC phases use the CIM's local memory bus. Hence, for pipeline depth of 2, the DMA and FC phases occur successively, and may overlap with $\mathcal{M}_{overhead}$. Since the DMA and FC phases take longer than $\mathcal{M}_{overhead}$, the average message delay with pipeline depth of 2 may be estimated as:

$$\mathcal{T}_{m(pipe=2)}(\mathcal{S}) \quad = \quad \mathcal{T}_{DMA}(\mathcal{S}) + \mathcal{T}_{FC}(\mathcal{S}) \qquad (4.4)$$

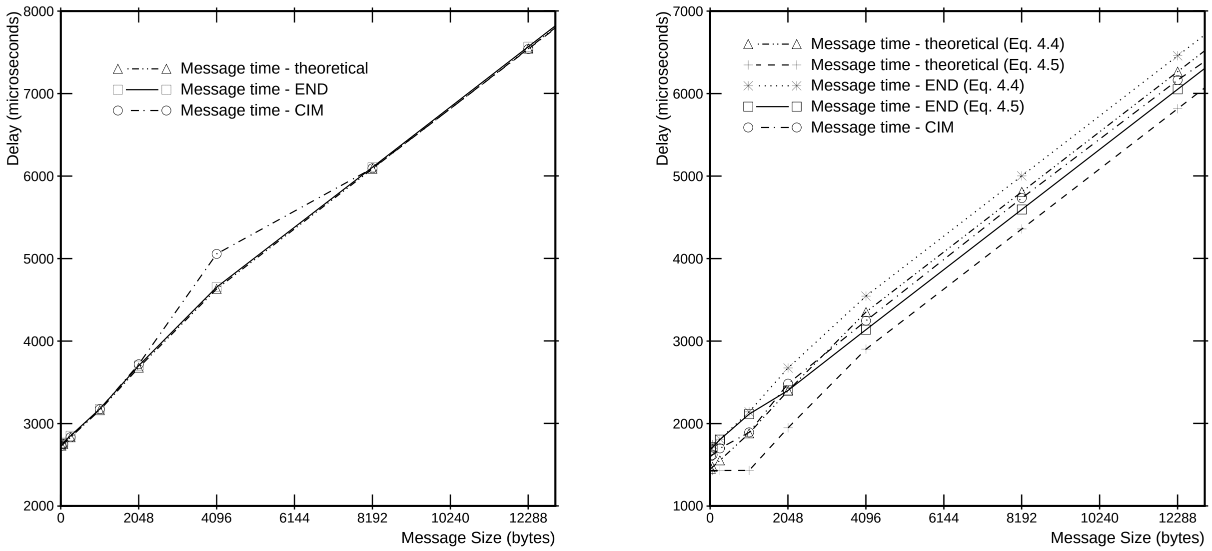
This is illustrated in illustrated in Figure 9, where $M_o$, $T_d$, $F_o$ and $F_d$ correspond to $\mathcal{M}_{overhead}$, $\mathcal{T}_{DMA}$, $\mathcal{F}_{overhead}$ and $\lceil\frac{\mathcal{S}}{\mathcal{F}_{framesize}}\rceil \times \mathcal{F}_{framedelay}$, respectively. Eq. 4.2 above corresponds to the timeline for `Message 1`, while Eq. 4.4 corresponds to `Message 2 (case 1)`.

Note that the above analysis will always be optimistic with respect to the corresponding *END* model (Figure 10(b)). This is because, in practice, perfect overlap between $\mathcal{M}_{overhead}$ and other operations is unlikely and one must account for the cost of other operations on *END*. The measured performance on the CIM, on the other hand, is better than that predicted by the above analysis, indicating that it does not account for some other source of concurrency in CIM operations.

The above analysis assumes that no part of the DMA and FC phases can overlap; however, only the data transfer part of these two phases cannot overlap due to contention for the CIM memory bus. Some or all of $\mathcal{F}_{overhead}$ may overlap with other operations. From empirical observations, allowing $\frac{1}{3}\mathcal{F}_{overhead}$ to execute concurrently with other operations yields good agreement between the CIM and *END* performance. The revised equation is:

$$\mathcal{T}_{m(pipe=2)}(\mathcal{S}) \quad = \quad \max(\mathcal{T}_{DMA}(\mathcal{S}), \frac{1}{3}\mathcal{F}_{overhead}) + \frac{2}{3}\mathcal{F}_{overhead} + \lceil\frac{\mathcal{S}}{\mathcal{F}_{framesize}}\rceil \times \mathcal{F}_{framedelay} \qquad (4.5)$$

This corresponds to `Message 2 (case 2)` in Figure 9, where $T_d$ is split into two parts, $T_{d_1}$ and $T_{d_2}$. $T_{d_1}$ overlaps with $F_o$, after which the FC operations seize the bus and preempt any further DMA




(a) Pipeline depth = 1 (b) Pipeline depth = 2

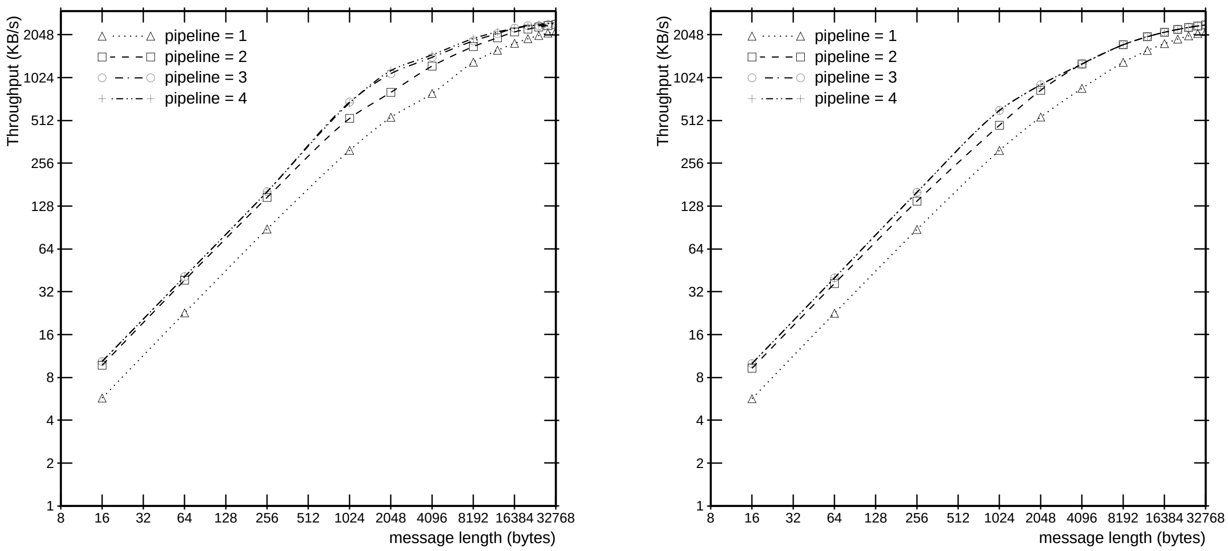
Figure 10: Theoretical and measured mean message times.

transfer. This is because FC operations have higher priority than DMA. The remaining part, $T_{d_2}$, resumes when Message 1 completes transmission.

**Discussion:** Figure 10(a) shows good agreement between the predicted and emulated performance when pipeline depth is 1, and both are quite close to the real measurements on the CIM. In Figure 10(b), for a pipeline depth of 2, we see that the theoretical delay is always less than that measured using the same delay model in *END*, for reasons stated earlier. The *END* model (with Eq. 4.5) shows good agreement with the real CIM, though not as good as with pipeline depth of 1. Note that we are able to construct reasonably accurate theoretical models because the system has homogeneous traffic and is otherwise idle. Under more complex traffic conditions, or greater pipeline depth, or other applications running on the host, it will be very difficult to construct accurate theoretical models. However, since *END* captures low level details, and interacts dynamically with a real host, its behavior will track that of a real system. We should point out that FC and DMA data transfers cannot occur simultaneously, and this knowledge suffices to construct models capturing average behavior. However, the actual sequence of events is important to measure delay jitter.

### 4.3.3 Equivalence of the CIM and the *END* model

To show the equivalence of the CIM and its *END* model, throughput and delay were measured for various message sizes and various values of pipeline depth. Figure 11 shows the mean throughput for the CIM and *END*. The curves for pipeline depth of 1 are almost identical in both graphs (mean error = 1.1%). The error increases to 3.3%, 6.5% and 7.9% for pipeline depths of 2, 3 and 4, respectively. The reason for this increase in error is that this model does not capture all the details of the hardware interactions in the CIM, and these interactions increase as the number of messages increases. Figure 10




(a) CIM throughput (b) *END* throughput

Figure 11: Comparison of CIM and *END*'s transmission throughput. Note that the graph uses a log-log scale, and throughput increases by as much as 50% when maximum pipeline depth is increased from 1 to 2.

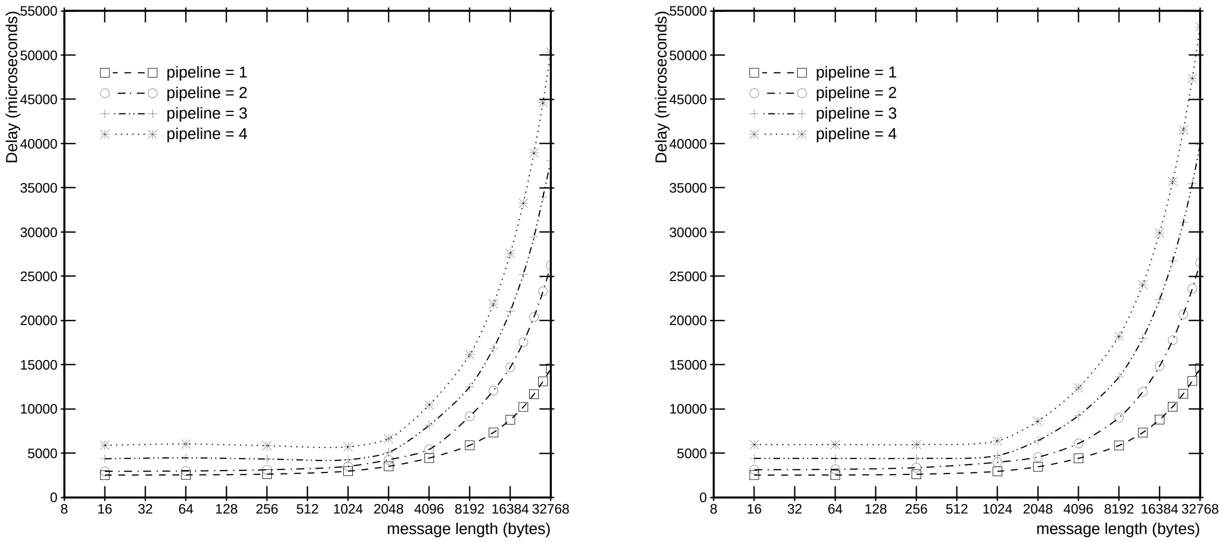
shows the mean message delays for pipeline depths of 1 and 2; since the real and emulated curves are on the same graphs, the accuracy of the emulation is apparent visually.

Similar results are observed for message delays. Figure 12 shows the mean message delays. Note that while the mean delays are almost identical, the standard deviations are quite different. The standard deviations are comparable for pipeline depth of 1, but increase rapidly on the CIM, but remain virtually unchanged on *END*. As before, this is due to the fact that *END* does not capture all the interactions of the messages on the CIM which cause increased delay jitter.

However, it should be noted that all performance information regarding the CIM was inferred purely from external measurements and without knowledge of the exact firmware. When designing a real network adaptor, knowledge of the exact firmware and hardware/software interactions would help build even more accurate models. In such cases, it would be possible not only to determine potential interactions, but, if such interactions were considered undesirable, to redesign the architecture or firmware to minimize their impact.

## 5 *END* as a Design Tool

In this section we demonstrate the use of *END* as a design tool to explore and evaluate design alternatives. We use *END* to identify and correct performance bottlenecks in the CIM design. Since *END* emulates a representative model of the CIM, we may reasonably conclude that performance enhancements observed due to any changes in the *END* model of the CIM would be representative of performance enhancements due to similar changes in the real CIM. We consider two main design




(a) CIM message delay (b) *END* message delay

Figure 12: Comparison of CIM and *END*'s transmission delay.

modifications to improve performance:

(i) simplification of the host-adapter interface to reduce overhead (Section 5.1), and

(ii) architectural changes to increase concurrency in CIM operations (Section 5.2).

The performance improvements due to these changes are described in Section 5.3.

## 5.1 Reducing the Host-Adapter Interface Overhead

Figure 6 shows the host-adapter interface for CIM `write` operations. The host sends two commands to the CIM, and gets three responses, each accompanied by an interrupt. In the initialization phase, the host device driver and the CIM exchange identifiers to keep track of the rest of the transaction. However, this phase really serves no useful purpose. The initiator of a transaction (the device driver for `write` and the CIM for `read`) can create a unique transaction ID, which suffices to keep track of the progress of that transaction. Further, the CIM sends a response and an interrupt to signal the completion of the DMA and the FC phases. Again, this is not really necessary, and a user should be able to configure the interface to decide whether the responses need be sent after either or both these phases, as determined by the higher software layers. Note that the equations for delay in this case will be almost identical to Eq. 4.2 and Eq. 4.5; only the value of $\mathcal{M}_{overhead}$ will reduce.

## 5.2 Exploiting Increased Concurrency

Though the CIM has dedicated hardware to move data between the host and the CIM's memory, and between the CIM's memory and the network, these operations cannot proceed in parallel since both use the same memory bus. Dual memory banks with independent memory busses have been suggested as a cost effective technique to increase the memory system's bandwidth, as opposed to more expensive memory organizations like dual-ported memory [15]. For general purpose applications, data placement is a problem in such an architecture, since it is hard to guarantee that simultaneous memory accesses will be from different memory banks. The pattern of memory access is much simpler in network adapters and, if alternate transfers are placed in alternate memory banks (similar to double-buffering), most of the time it will be possible for the DMA and FC phases to proceed in parallel. With pipeline depth of 1, there is no benefit to having two busses. When the pipeline depth is 2, however, the DMA and FC phases proceed concurrently. As in Eq. 4.5, the message overhead and copy time for the address list get completely overlapped with the data transfer time, and the message delay is determined by the slower of the DMA or the FC phases. This delay may be estimated as:

$$\mathcal{T}_{m(twobus,pipe=2)}(N) \quad = \quad \max(\mathcal{T}_{DMA}, \mathcal{T}_{FC}) \tag{5.1}$$

### 5.3 Performance of the Improved CIM

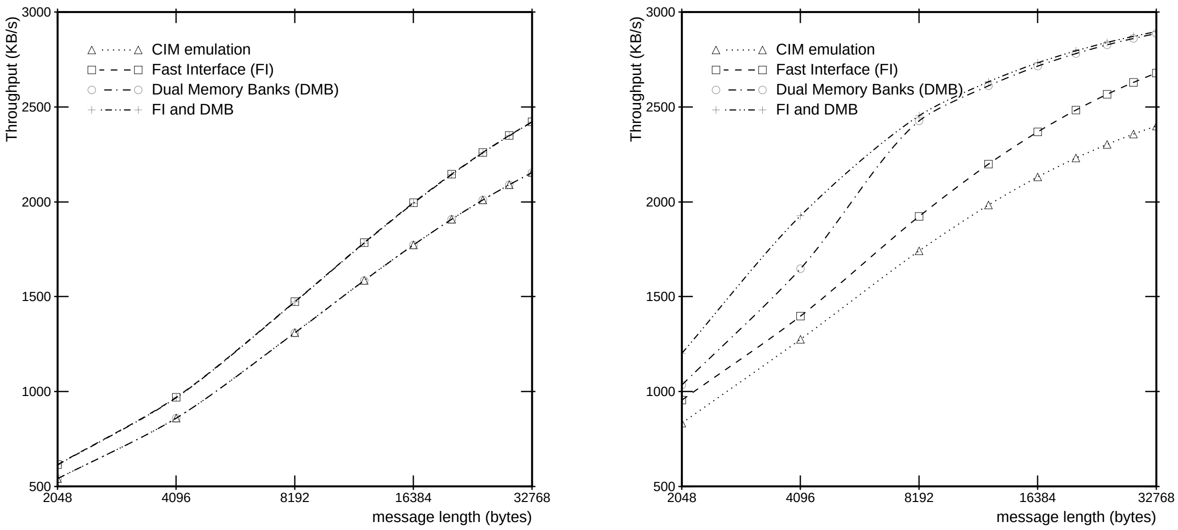
We modified the *END* model of the CIM to reflect the changes described above. For the changes in the interface, modifications were required both in the device driver and *END*. To capture the performance with dual ported memory banks, it suffices to modify the model so that DMA and FC phases proceed in parallel, i.e., the FC phase no longer preempts the DMA phase.

Figure 13(a) shows the performance improvements when the pipeline depth is 1. In this case, since at most one message is being processed on the adapter at a time, there is no memory contention and hence there is no benefit with dual memory banks; all performance improvements arise from the faster interface. Throughput increases by 12–15%, with a greater increase for smaller packets since, in these cases, the cost of the interface was more significant compared to the cost of transmission.[3]

Figure 13(b) shows the performance improvements when the pipeline depth is 2. In this case, throughput rises significantly due to the dual memory banks. For small messages, there is not much additional concurrency, and the throughput increases by about 8%. For larger messages, the throughput increases by as much as 28%. For very large messages, since the FC phase is much faster than the DMA phase, the cost of the DMA phase begins to dominate again, and the improvements decline to about 17%. With both features implemented together, throughput increases by 15–50%.

As expected, similar performance improvements were observed for measurements of individual message delays. The fast interface has an additional benefit in that delay jitter is reduced, especially for small messages. This is because the fast interface reduces the number of interactions with the host, and the accompanying unpredictability. This effect is more pronounced for small messages since the delay at the interface is a significant part of the total transmission delay.

[3] The FC phase has a large setup cost, and this dominates the transmission time for very small messages. This bounds the improvement in throughput.




(a) Pipeline depth = 1

(b) Pipeline depth = 2

Figure 13: Transmission throughput for improved CIM models.

# 6 Discussion

In Section 4 we established that *END* could accurately emulate the behavior of a real network adapter and in Section 5 we demonstrated how the model of the network adapter could be modified to evaluate design alternatives. While this shows how *END* was used in a particular instance, we now address issues of the general applicability of device emulation, in particular, of *END*.

**Portability of *END*:** *END* is written almost entirely in C (about 2500 lines of code; only the interrupt service routine wrappers are written in assembly) and is easily portable. It requires a minimal executive for services such as process management and interrupt handling, which are widely available on other platforms. The manner in which interrupts are generated and handled depends on the processor and bus controllers, and would need to be rewritten for different platforms. However, this is a very small part of the code and is easily modifiable. For time services, a high resolution clock is essential; this too is available on many modern microprocessors.

***END* as a Prototype:** As seen in Section 4, the device drivers for the real and emulated adapters were almost identical. This high degree of code reusability allows software development and testing to proceed in parallel with, or even ahead of, hardware design and implementation. In addition, the *END* model itself can serve as a prototype for the adapter firmware. All the algorithms and data structures used in *END* should be the same as those used in the real firmware. Each call to the `delay()` function corresponds to a hardware activity, and can be replaced by code to program the particular hardware device. In *END*, the `stage`$_i$`done()` calls correspond to completion of a timed delay, and in real adapters, they will be invoked on completion of a real hardware activity.

***END* as a Programmable Traffic Source:** *END* can be configured as a special network interface in the host operating system to act as a stochastic source of network traffic. Applications or higher software layers can open this interface and "program" it to generate network traffic with characteristics such as a given arrival distribution, source and destination addresses/ports, etc. This could be used by designers to debug or evaluate protocol implementations, for example.

**Limitations of *END*:** One of the main advantages of *END* is that it operates in real time and interacts with a real system, and hence, all overheads of the host operating system and applications are real. However, the *END* model itself is essentially a real-time simulation, and any study is only as accurate as the underlying simulation model. Since *END* runs on a general purpose processor board, it may not have the same hardware components as the target adapter. Consequently, it may not always be possible to represent all hardware interactions on an adapter. Further, the accuracy of *END* is limited by emulation overheads and several factors such as relative CPU speed, interrupt latency, and memory bandwidth on *END* and the target device, and in relation to that on the host. However, some of these can be circumvented by using a processor significantly faster than the host CPU.

Another potential limitation of *END* is scalability. Each *END* node needs at least two (maybe 3) processor boards and, assuming that different nodes communicate over the I/O bus, no more than 2–4 nodes would be possible in any configuration. As discussed in Section 3.3, limitations due to contention for I/O bus bandwidth may be partially addressed by using the bus only for sending message headers, rather than the entire data payload. Scalability and contention limitations may also be addressed by using WAN emulation in the style suggested by *hitbox* [16].

## 7 Related Work

The present work was motivated by the need to find an accurate and flexible tool for network adapter design, keeping in mind hardware-software codesign. Our work relates to and builds upon the following areas of research.

*Communication subsystem design and performance:* Several researchers have studied the issues affecting the design and performance of network adapters [1, 4, 17, 18], and communication subsystems in general [19, 20]. While many of these studies have influenced our work, we have focussed as much on the design process, as the design itself.

*Simulation-based evaluation:* Performance evaluation via simulation can be conducted at various levels of detail, and hence, accuracy. Recently, significant attention has been given to accurate, low-level simulation to study machine architectures while capturing operating system overheads [5, 21]. Other efforts have focused on protocol-level simulation with the ability to run the actual protocol stack during simulation [6], and network-level simulation with a focus on routing and end-to-end protocol performance [22, 23]. Most relevant to our work is architecture-level and protocol-level simulation. Since *END* executes on its own processor concurrently with the host, it avoids any intrusion on the host operating system (including the protocol stack) executing on the host CPU. More importantly, *END* utilizes actual hardware resources on the host (system I/O bus, interrupts) just as a real adapter would; capturing events at this low granularity in simulation models would require highly accurate resource models and render the simulation extremely slow.

*Network adapter as source/sink of data:* Network adapters have been modeled as simple data sources and sinks for parallel protocol implementations [24]. Interestingly, these models are executed on a separate processor within the host. Besides modeling the source/sink behavior of the network, our approach captures significantly more details of adapter design and interaction with the protocol stack.

*I/O device modeling:* Other researchers have studied the issues involved in modeling disks [8]. While the focus of our work is network device emulation, our emulation framework can be extended to emulate disks and interact with the file system layer in the operating system. We believe that this would provide a reasonable mechanism to study storage subsystem performance.

## 8 Conclusions and Future Work

In this paper, we proposed network device emulation as a performance evaluation technique and described the architecture and implementation of *END*, a tool for designing network adapters. We used *END* to accurately model the behavior of a real network adapter, the Ancor VME CIM 250, and extend its original design to remove performance bottlenecks. These design improvements yielded throughput increases of 15–50%, depending on the traffic load. We also discuss issues relating to portability and extensibility of *END* to other host architectures, network adapters, and I/O devices. We believe that *END* can be employed by device designers to compare and evaluate design alternatives before actually building a prototype.

We are exploring several avenues of future work for studying communication subsystem performance using *END*. We are in the process of completing reception functionality in *END*, using true two-way communication as well as synthetic traffic generation. We are also adding a control interface to *END* through which protocol designers and applications can request various synthetic traffic patterns and exercise dynamic control over the incoming traffic. *END* is also being extended to make it more generally applicable, especially in WAN environments. This will enable us to model larger networks and study end-to-end protocols and networking issues with traffic exchange between multiple network nodes. To further enhance the usability of *END*, we are also looking into developing libraries for interrupt wrappers, FIFO and priority queues, and stochastic models for traffic sources. Finally, we are also interested in extending *END* to inject communication faults in a data stream, allowing it to be used to study the reliability and performance of communication protocols.

## References

[1] K. K. Ramakrishnan, "Performance considerations in designing network interfaces," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 203–219, February 1993.

[2] *Reference removed to maintain anonymity.*

[3] *Reference removed to maintain anonymity.*

[4] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *Proc. of ACM SIGCOMM*, pp. 2–13, London, UK, October 1994.

[5] R. C. Bedichek, "Talisman: Fast and accurate multicomputer simulation," in *Proceedings of Sigmetrics 95/Performance 95*, pp. 14–24, May 1995.

[6] L. S. Brakmo and L. L. Peterson, "Experiences with network simulation," in *Proceedings of ACM Sigmetrics 96*, pp. 80–90, May 1996.

[7] R. K. Budhia, P. M. Melliar-Smith, L. E. Moser, and R. M. er, "Higher performance and implementation independence: Downloading a pro tocol onto a communication card," in *Proc. of the Intl. Conf. on Comm.*, June 1995.

[8] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "On-line extraction of SCSI disk drive parameters," in *Proceedings of Sigmetrics 95/Performance 95*, pp. 146–156, May 1995.

[9] A. Gokhale and D. C. Schmidt, "Measuring the performance of communication middleware on high-speed networks," in *Proc. of ACM SIGCOMM*, August 1996.

[10] *Reference removed to maintain anonymity.*

[11] *Reference removed to maintain anonymity.*

[12] *VME CIM 250 Reference/User's Manual*, ANCOR Communications, Inc., 1992.

[13] M. Lin, J. Hsieh, D. H. C. Du, and J. A. MacDonald, "Performance of high-speed network I/O subsystems: Case study of a fibre channel network," in *Supercomputing '94*, November 1994.

[14] *Fibre Channel Physical and Signalling Interface (FC-PH)*, American National Standards Institute, rev. 3.0 edition, June 1992.

[15] M. A. R. Saghir, P. Chow, and C. G. Lee, "Exploiting dual data-memory in digital signal processors," in *Proc. of the 7th ACM conf. on Arch. Support for Prog. Lang. and Oper. Sys.*, October 1996.

[16] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan, "Evaluation of TCP vegas: Emulation and experiment," in *Proc. of ACM SIGCOMM*, pp. 185–195, October 1995.

[17] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley, "Afterburner," *IEEE Network Magazine*, pp. 36–43, July 1993.

[18] P. A. Steenkiste, "A systematic approach to host interface design for high-speed networks," *IEEE Computer*, pp. 47–57, March 1994.

[19] P. Druschel, M. Abbott, M. Pagels, and L. Peterson, "Network subsystem design," *IEEE Network Magazine*, pp. 8–17, July 1993.

[20] D. C. Schmidt and T. Suda, "Transport system architecture services for high-performance communications systems," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 4, pp. 489–506, May 1993.

[21] E. Witchell and M. Rosenblum, "Embra: Fast and flexible machine simulation," in *Proceedings of ACM Sigmetrics 96*, pp. 68–79, May 1996.

[22] S. Keshav, "REAL : A network simulator," UCB CS Tech Report 88/472, University of California, Berkeley, December 1988.

[23] S. McCanne and S. Floyd. *NS (Network Simulator)*, 1995. Available via http://www-nrg.ee.lbl.gov/ns.

[24] M. Bjorkman and P. Gunningberg, "Locking effects in multiprocessor implementations of protocols," in *Proc. of ACM SIGCOMM*, pp. 74–83, September 1993.