# The *END*: A Network Adapter Design Tool *


Atri Indiresan
Cisco Systems Inc.
170 W. Tasman Drive
San Jose, CA 95134
*atri@cisco.com*

Ashish Mehra
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
*mehraa@watson.ibm.com*

Kang G. Shin
Real-time Computing Lab.
Dept. of EE & CS
The University of Michigan
Ann Arbor, MI 48109
*kgshin@eecs.umich.edu*



## Abstract

*We present the design and implementation of* Emulated Network Device (END)*, a network adapter design tool that facilitates accurate evaluation of alternative adapter designs. Using device emulation,* END *permits designers to couple a representative model of an adapter with a real host and its communication software. Different adapter designs can be evaluated and compared accurately in a realistic setting, i.e., while capturing host-adapter concurrency and interaction overheads, before building a prototype. We present the architectural framework adopted by* END *and demonstrate its feasibility via a case study of a commercial network adapter. Several design improvements to alleviate performance bottlenecks are realized and evaluated using* END*, highlighting its utility as a network adapter design tool.*


## 1. Introduction

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, a typical configuration for many network adapters [17]. This allows *END* to generate the same overheads as real adapters. Further, by using a synthetic network model as a sink and source of traffic, *END* can emulate all the operations of a network adapter without interfacing to a real network.

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 prototyping 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 *co-design*, where adapter design tradeoffs can be explored early in the design cycle.

Device emulation has been used before to model source/sink behavior of I/O devices (disks, networks, sensors). Our distinct contribution lies 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, and (ii) to explore design alternatives that improve the performance of an existing prototype. We have also used *END* to study QoS issues in adapter design and solutions to the receive livelock problem [10]. In this paper, we demonstrate how *END* was used to explore design alternatives for an existing device by conducting a case study


*The work reported in this paper was performed while the authors were at the University of Michigan. It 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.

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*, using *END*. The base model is then modified to incorporate design changes that improve communication performance for outgoing traffic. 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.

In the rest of the paper, Section 2 highlights issues in network adapter design and compares device emulation to other evaluation techniques. The architectural framework and our implementation of *END* are presented in Section 3. Section 4 presents the case study of the Ancor CIM 250 network adapter and its emulation using *END*. Section 5 illustrates use of *END* to evaluate design improvements for outgoing traffic. The general applicability of *END* is highlighted in Section 6. Section 7 discusses related work, and Section 8 concludes the paper.

## 2. Network Adapter Design Issues

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

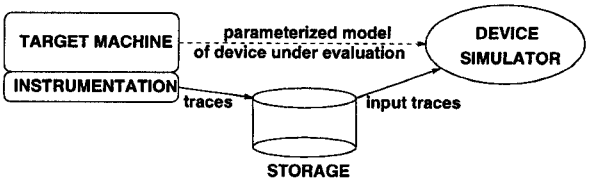
Within end hosts, communication performance is largely determined by the efficiency with which communication software and hardware/firmware components interact to move data between applications and the network (Figure 1(a)). Data transfer involves traversing a protocol stack, moving data between the host memory and the network adapter, and between the network adapter and the network. Figure 1(b) illustrates the five basic components of a typical network adapter: host-adapter interface, data-transfer control module, transmission/reception queuing module, buffer-management module, and adapter-network interface. Typically, most network adapters are accessed by the host via the system I/O bus [17], and employ one or more general-purpose microprocessors and custom hardware that operates under control of the adapter firmware.

**Host-Adapter Interface:** This interface typically has a device driver running on the host, and a "host driver" on the adapter. The two drivers exchange information across the system I/O bus, and may synchronize their operations either via interrupts and/or polling [20].

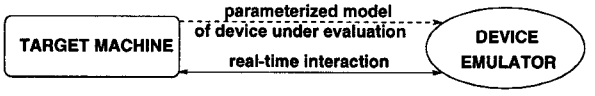
**Adapter Internals:** The following three modules provide the internal functionality of the adapter:

*Data Transfer Control:* Data transfer between host and adapter memory may be via DMA or programmed I/O (PIO). PIO has no startup latency, but requires the CPU for the duration of the transfer, while the CPU is required only to set up the DMA.

*Tx/Rx Queuing:* Best-effort adapters may provide simple first-in-first-out (FIFO) queuing of packets with deep pipelining of




(a) Trace-driven simulation




(b) Device emulation

**Figure 2. Simulation vs emulation.**

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.

*Buffer Management:* The adapter may need to manage packet buffers that may reside either in adapter or host memory. It must provide buffer management policies, such as reserving buffers for connections with QoS guarantees, as well as handle buffer overload conditions correctly.

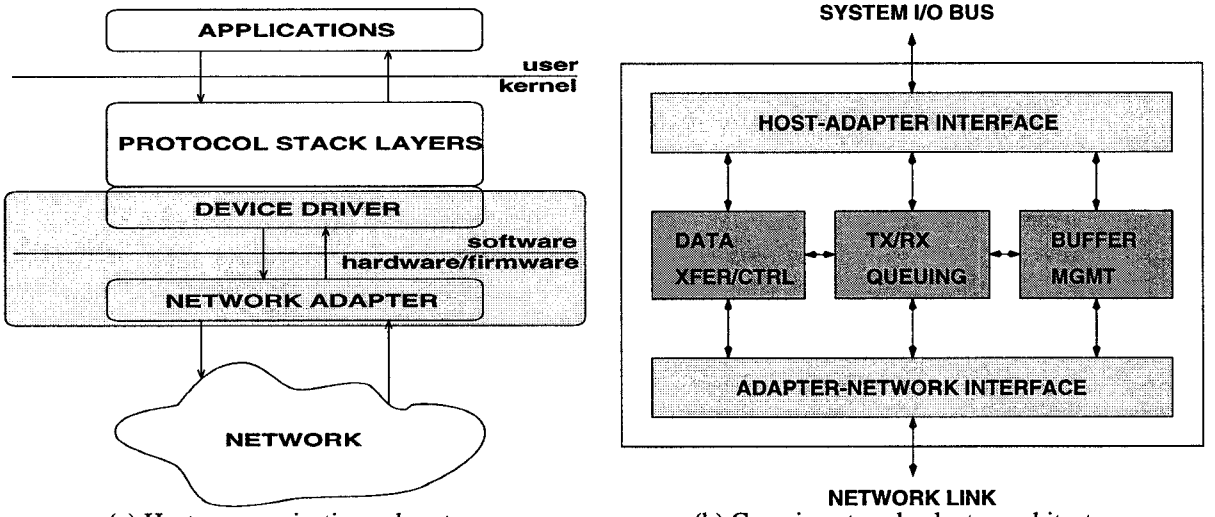
**Adapter-Network Interface:** The adapter may use PIO or DMA to move data between the network and packet buffers. If packet buffers reside in host memory, the adapter needs to inform the host on completion of packet transmission so that the buffer may be reused. On packet reception, it must stage data in its buffers and subsequently transfer the data to the host.

### 2.2. Design and Evaluation Techniques

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: mathematical modeling, simulation, emulation and prototyping.

*Mathematical modeling* is typically used to study the queuing behavior of network traffic. Though mathematical models are relatively inexpensive to develop for simple systems, they rarely account for system overheads encountered in practice. Detailed models capturing concurrency, contention, and component interaction in real systems rapidly become intractable.

(Trace-driven) *simulation* has several advantages [2]. Since simulators are built in software, they are usually easier and cheaper to build than real systems and can be readily modified to test new features and interfaces. However, a simulator typically has no real system components and must be accurately parameterized via performance measurements. Exceptions to this do exist in approaches that execute actual software under control of the simulator [4]. However, such approaches may not be applicable when hardware components (such as caches and 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




(a) Host communication subsystem (b) Generic network adapter architecture

**Figure 1. Focus of this paper.**

of real systems. Increasing the accuracy of the simulation model increases its cost while further slowing it down.

*Prototyping* a device and interfacing to higher software layers is time-consuming and expensive. While highly accurate, prototypes are not easily modifiable. The on-board firmware may be modified to study different design options [8], or one may employ programmable adapters [5] or use FPGAs for some hardware components, but the internal hardware architecture is typically impossible to modify without developing a new prototype. More importantly, hardware engineers designing network adapters are often far removed from the concerns of those writing the communication software, and vice versa, producing a design poorly integrated with the host operating system.

### 2.3. The Case for Device Emulation

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)), giving the latter the impression that it is interacting with the actual subsystem being emulated in real-time. For trace-driven simulation (Figure 2(a)), the target machine is instrumented to generate run-time traces that are 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. 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.

*END* emulates a network adapter and interfaces with the target host, giving the impression that the host 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 software that interfaces to the device, and rapid integration of the hardware device when it becomes available.

## 3. *END* Emulation Architecture

In this section, we present the emulation framework for *END*, define the functional interface it must export to the host, and describe our prototype implementation.

### 3.1. Host View of the Network

The host views the network adapter as a sink for data transmission, and a source for data reception. It exchanges commands and data with the adapter, the timing of these events being determined by network characteristics and traffic conditions. To accurately emulate a network adapter, *END* must export to the host the same functional interface as, and have performance characteristics similar to, that of a real adapter.

*END* interacts with the host like a real device, exchanging commands/data and synchronizing via interrupts or polling. Typically, the communication subsystem acknowledges data transmission after a delay determined by the size of data, and the system and network load. As long as applications do not expect acknowledgments, all they see is data being transmitted at a certain rate. A transmitting application will not be able to distinguish between *END* and a real network as long as *END* captures this timing behavior. For data reception, *END* generates incoming traffic based either on stochastic models or real traffic traces. Genuine two-way traffic may also be generated.

### 3.2. Emulator Components

Figure 3 shows our emulator-based architecture for studying adapter design. Each network "node" corresponds to a host and an emulator processor board on the same I/O bus.

**Host Node:** The host node is precisely the target host. Communicating applications send and receive data via the protocol stack which has a device driver for the target network adapter.

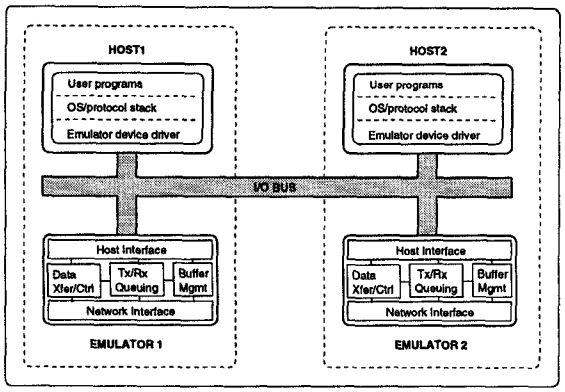



**Figure 3. *END*-based emulation architecture.**

Though this driver communicates with *END*, it implements the real host-adapter interface. This permits the implementation and testing of a complete driver even while the network adapter is being designed, and ensures that observed performance of the driver is comparable for the emulated and real adapters.

**Network Adapter:** The network adapter node is a general-purpose processor board (running *END*), 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.

Packet transmission is modeled as a transmission-complete notification after a suitable delay. Since actual data is not transferred, this does not capture the overhead of cycle stealing during DMA data transfers. However, this is also an advantage in that the system may be configured for arbitrary network speeds, allowing *END* to operate at different network speeds.

Packet arrivals can be generated either using a stochastic model, or using two-way communication. 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 simulate transmission and reception delays in real time. 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 can be polled by the adapter.

### 3.3. Two-Way Communication Across the "Network"

For two-way communication, we utilize the system I/O bus as the "network" (similar to [9]), with data being transferred from one adapter node to another (see Figure 3). Though the I/O bus limits the speeds that can be emulated, this approach is quite useful, since it can be used to verify the functional correctness of protocols and conduct relative performance comparisons rather than derive 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 packets, I/O bus contention is low; this permits evaluation of adapter performance for a wide range of network speeds.

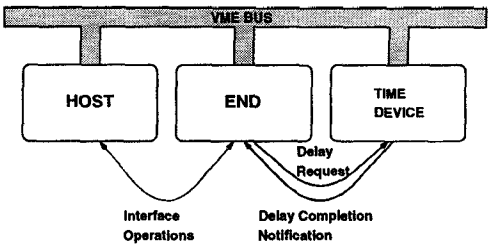



**Figure 4. Emulator system configuration.**

### 3.4. Implementation

Figure 4 depicts our implementation of one emulator node (each additional node is another HOST-*END* pair; the time device is shared). The nodes communicate with each other via the VME-bus. The internal structure of *END* comprises an *emulation core* and several components (described below). The emulation core is a minimal executive supporting threads, interrupt handling, and semaphores. It is essentially a cyclic executive that polls the host for commands, executes them, and notifies the host of command completion. Its components may be configured and parameterized to realize any desired adapter behavior.

**Host–emulator interface:** The host–emulator interface uses *command/response queues* to exchange information. The interface is determined by the actual commands, and whether synchronization is via interrupts and/or polling.

**Data transfer control:** In our prototype, no actual data is transferred, except for packet headers for two-way traffic and pure reception. We supply generic time models for DMA and PIO in order to capture the associated timing correctly. 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 is idled for the duration of the PIO, and not allowed to perform 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. FIFO and Earliest Due Date (EDD) queuing policies have been implemented, and the selection of queues may either be round robin or based on their relative priorities.

**Buffer management:** Buffers are allocated for outgoing packets from a shared pool on the host, while incoming packets are allocated buffers on the adapter. Buffers on the host and/or adapter may also be reserved on a per-connection basis [15].

**Network interface:** The network interface for data transmission is a 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:** To model a delay operation, *END* computes the completion time by adding the desired delay to the current time, and writes a delay request to the *time device*. The time device is a cyclic executive that executes on a processor board without

any OS support, reading delay requests and issuing completion notifications. A *completion function* associated with each request is executed when the time interval expires. In operations involving more than one emulator (as in two-way communication), all nodes are notified of the completion. The time device also provides for creating and destroying event-generating stochastic processes. Any arbitrary function or stored traces may be used to determine event arrival times.

*END* is implemented as a multi-stage device using the above components. 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, buses, network links). Activities in different stages may proceed concurrently as long as they do not contend for the same resources.

## 4. 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*. The CIM is a suitable candidate for this study given that our version of the adapter had several performance bottlenecks [11] caused by poor design decisions. To identify some of the performance characteristics of the CIM, we have used the results obtained in [13]. Note that *END* has been designed independently of the CIM; this study therefore serves as a fair test of *END*'s ability to model real adapters.

Our methodology is to first construct a *base model* representative of the device. A representative model is one that performs the same functions and has performance equivalent to that of the target device. Once the base model is constructed, design changes can be incorporated in *END*. Subsequent changes in observed performance of the modified model would be similar to those due to similar design modifications in the real adapter. Constructing a representative model of an existing adapter requires systematic "black box" performance analysis of the device. While we have implemented receive mechanisms and performed end-to-end experiments, we focus on data transmission in a single-node configuration with a single transmitting node.

### 4.1. Ancor VME CIM 250

The Ancor VME CIM 250 [1] (CIM) is a network adapter for ANSI Fiber Channel (FC) 3.0 networks. Besides communication interface hardware, the CIM has an NEC 32 MHz 70236 I/O processor, 8MB DRAM, independent DMA controllers, and VMEbus interface logic. The CIM communicates with the host using command/response FIFOs; it supports commands to initiate read and write operations, set up DMA, and signal the completion of DMA and network transmission. The CIM polls the command FIFO for commands from the host, and uses interrupts to notify the host of responses in the response FIFO. A simplified sequence of the events involved in CIM transmission is listed in Figure 5. Similar events occur on reception as well (see [1, 13] for details). Note that the host may issue additional commands before a previous one completes. This increases concurrency since different operations may now occur

| Phase | Device driver | | VME CIM 250 |
|---|---|---|---|
| Initialization | send `write` | $\longrightarrow$ | start transmit sequence |
| | | $\longleftarrow$ | send `write ack` |
| | | $\longleftarrow$ | Interrupt Request |
| DMA | send `address list` | $\longrightarrow$ | DMA list of address commands to local memory<br>DMA data to local memory |
| | | $\longleftarrow$ | send `address ack` |
| | | $\longleftarrow$ | Interrupt Request |
| Fiber Channel | | | Transfer data to Fiber Channel network interface |
| | | $\longleftarrow$ | send `transmit ack` |
| | | $\longleftarrow$ | Interrupt Request |

**Figure 5. Host–adapter interaction (transmit).**

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. The only difference is that the CIM has hardware support for command/response FIFOs, 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 functions supported are adequate to completely define the host–*END* interface, i.e., this model interacts correctly with the host device driver.

### 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], our own measurements [11], and conversations with staff at Ancor Communications, Inc. Lin *et al.* [13] characterized the delays of various components of the CIM. These delays deliver a performance significantly different from ours since they used a different platform. We utilized only those measurements that are internal to the CIM, and hence, not affected by the platform. These include the delays for setting up the DMA and the FC phase. The delays during DMA, for example, are computed from measurements made on our platform.

**Accurate Base Model:** We construct an accurate base model for the CIM by computing $\mathcal{T}_D$ and $\mathcal{T}_F$, the times for the DMA and FC phases, respectively. While $\mathcal{T}_F$ is obtained from [13], $\mathcal{T}_D$ must be computed via measurements since it is platform-dependent. The definitions and values of the terms used in this section are given in Table 1. $\mathcal{T}_F$ is given by

$$\mathcal{T}_F(\mathcal{S}) = \mathcal{F}_o + \lceil \frac{\mathcal{S}}{\mathcal{F}_s} \rceil \times \mathcal{F}_d \quad (1)$$

We compute $\mathcal{T}_D$ by measuring $\mathcal{T}_m^{p=1}$, the transmission time on the CIM with a pipeline depth of 1, as follows:

$$\mathcal{T}_m^{p=1}(\mathcal{S}) = \mathcal{M}_o + \mathcal{T}_D(\mathcal{S}) + \mathcal{T}_F(\mathcal{S}), \quad (2)$$

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

**Table 1. Important system parameters**

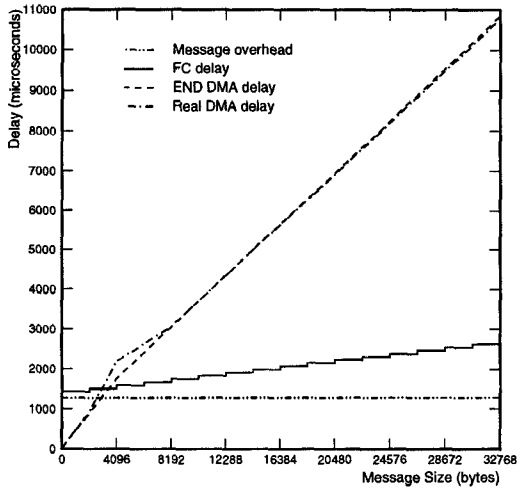



**Figure 6. Sources of delay in the model.**

where $\mathcal{M}_o$ corresponds to the cost of generating the message, traversing the protocol stack, and crossing the host–CIM interface for the entire message. $\mathcal{T}_D$ can now be computed as the only unknown in Eq. (2). Our computed value for $\mathcal{T}_D$ showed that the incremental delay is a non-linear function of $\mathcal{S}$, being much higher for messages smaller than $\mathcal{S}_s$, and lower for larger messages (see Figure 6), as shown below:

$$\mathcal{T}_D(\mathcal{S}) = \begin{cases} \lceil \frac{\mathcal{S}}{\mathcal{D}_s} \rceil \times \mathcal{D}_o + \mathcal{S} \times \mathcal{D}_b^s & \text{if } \mathcal{S} < \mathcal{S}_s \\ \lceil \frac{\mathcal{S}}{\mathcal{D}_s} \rceil \times \mathcal{D}_o + \mathcal{S}_s \times \mathcal{D}_b^s & \\ \quad + (\mathcal{S} - \mathcal{S}_s) \times \mathcal{D}_b^l & \text{otherwise.} \end{cases} \quad (3)$$

This approximation works well except for the region around $\mathcal{S} = 4KB$, where the CIM's DMA performance diverges.

**Accounting for Concurrency and Contention:** When multiple packets are being transmitted or received on the CIM, $\mathcal{T}_D$ of one message may (partially) overlap $\mathcal{M}_o$ 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 $\mathcal{M}_o$. Since the DMA and FC phases take longer than $\mathcal{M}_o$, the average message delay with pipeline depth of 2 may be estimated as:

$$\mathcal{T}_m^{p=2}(\mathcal{S}) = \mathcal{T}_D(\mathcal{S}) + \mathcal{T}_F(\mathcal{S}) \quad (4)$$

In Figure 7, Eq. (2) corresponds to the timeline for `Message 1`, while Eq. (4) corresponds to `Message 2 (case 1)`. However, some of $\mathcal{F}_o$ may overlap with other operations. From empirical observations, the revised equation below yields good agreement between the CIM and *END*.

$$\mathcal{T}_m^{p=2}(\mathcal{S}) = \max(\mathcal{T}_D(\mathcal{S}), \frac{1}{3}\mathcal{F}_o) + \frac{2}{3}\mathcal{F}_o + \lceil \frac{\mathcal{S}}{\mathcal{F}_s} \rceil \times \mathcal{F}_d \quad (5)$$

This corresponds to `Message 2 (case 2)` in Figure 7, 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 transfer. The remaining part, $T_{d_2}$, resumes when Message 1 completes transmission.

**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 8 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. Similar results are observed for message delays (not shown here). 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 adapter, 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. Improving the Design of Ancor CIM 250

We now demonstrate the use of *END* as a design tool to evaluate design alternatives to improve the handling of outgoing traffic. 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 design modifications to improve performance: (i) simplification of the host-adapter interface to reduce overhead, and (ii) architectural changes to increase concurrency in CIM operations.

### 5.1. Reducing Host-Adapter Interface Overhead

Figure 5 describes the CIM `write` operation. The host sends two commands to the CIM, and gets three responses, each with an interrupt. In the initialization phase, the host and the CIM exchange transaction identifiers. However, the initiator of a transaction (the host for `write` and the CIM for `read`) can create a unique transaction ID. The CIM also sends a response for both the DMA and FC phases, when a single response should suffice.

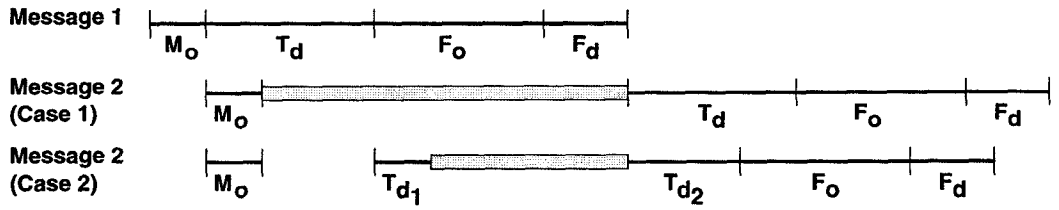
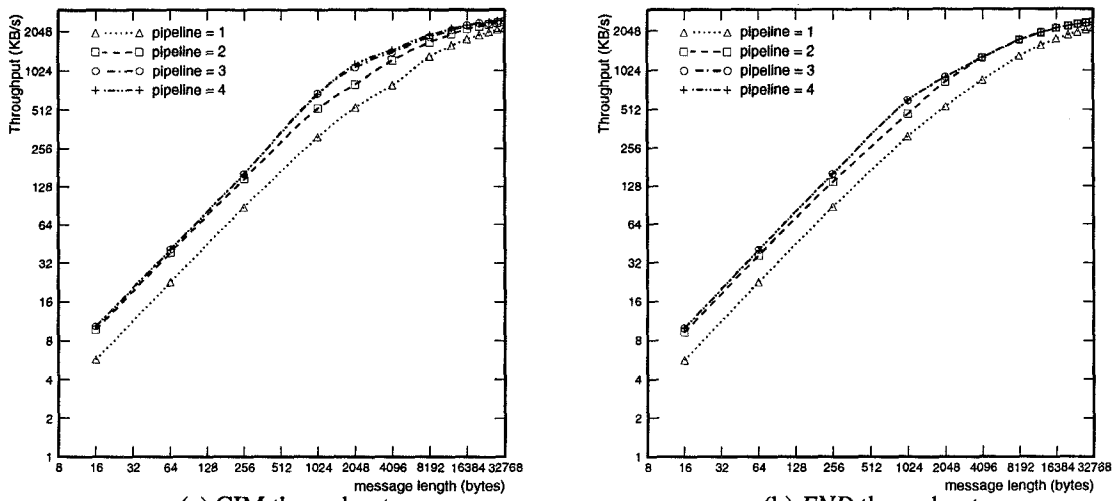



**Figure 7. Inter-message overlap during CIM transmissions.**




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

**Figure 8. Comparison of CIM and *END*'s transmission throughput. Note that the graph uses a log-log scale, and throughput increases by about 50% when maximum pipeline depth increases from 1 to 2.**

The equations for delay in this case will be almost identical to Eq. (2) and Eq. (5); only the value of $\mathcal{M}_o$ will be reduced.

### 5.2. Exploiting Increased Concurrency

The DMA and FC phases cannot proceed in parallel since they share the same memory bus. Dual memory banks with independent memory buses 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 [18]. 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 buses. When the pipeline depth is 2, however, the DMA and FC phases proceed concurrently. As in Eq. (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:

$$\mathcal{T}^{p=2}_{m(2bus)}(N) = \max(\mathcal{T}_D, \mathcal{T}_F) \quad (6)$$

### 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.

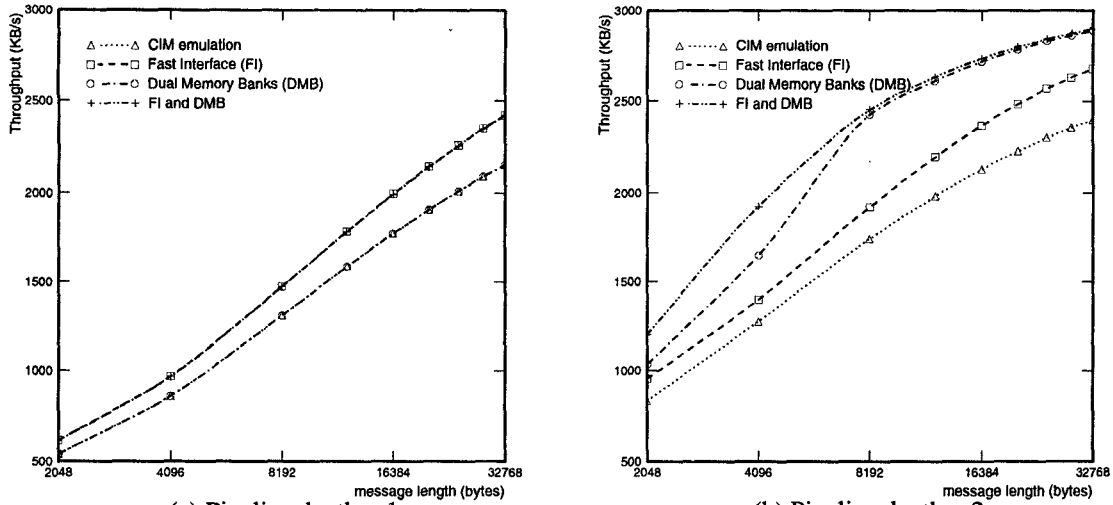
With a pipeline depth of 1 (Figure 9(a)), since at most one message is processed at a time, there is no benefit from dual memory banks; all performance improvements arise from the faster interface. Throughput increases by 12–15%, with a greater increase for smaller packets since, in these cases, the cost of the interface was more significant compared to the cost of transmission[1]. With a pipeline depth of 2 (Figure 9(b)), 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, throughput increases by 15–50%.

## 6. General Applicability of *END*

As outlined below, *END* can be readily ported to different platforms and utilized in several interesting ways.

**Portability and Scalablity of *END*:** *END* is written almost entirely in C (about 3000 lines of code for *END* and 2000 lines for the time device) and is easily portable. It requires a minimal

[1]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 9. Transmission throughput for improved CIM models.**

executive for process management and interrupt handling services, and a high resolution clock for time services, all widely available on other platforms.

While *END* operates in real time and interacts with a real system, the *END* model is essentially a real-time simulation and only as accurate as the underlying model. Since *END* runs on a general-purpose processor board, it may not always be able to represent all hardware/firmware interactions on an actual adapter. Further, the accuracy of *END* is limited by emulation overheads and the relative CPU speed, interrupt latency, and memory bandwidth on *END* and the target device. Some of these concerns can be circumvented by using a processor significantly faster than the host CPU. With two processor boards per node, and nodes interacting across the I/O bus, bus contention may limit the scalability of *END*. I/O bus contention may be partially alleviated by only exchanging packet headers instead of the entire data payload.

***END* as a Prototype:** As mentioned in Section 4, the device drivers for the real and emulated adapters were almost identical. The high degree of code reusability afforded by *END* allows software development and testing to proceed in parallel with hardware design and implementation. In addition, the *END* model itself can serve as a prototype for the adapter firmware.

***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 potentially be extremely useful to designers for debugging and/or evaluating protocol implementations.

In other work [10], we have demonstrated the versatility of *END* by studying a range of issues affecting end host communication subsystem design, as summarized below.

**Quality-of-Service:** Many applications have specific quality-of-service (QoS) requirements on communication, such as a minimum bandwidth and maximum delay in end-to-end communication. Support for QoS is necessary not only in the network, but also in the end-host operating system and communication subsystem. We studied network adapter design to support QoS for two distinct network configurations: point-to-point networks and shared networks. Our study highlighted the tradeoffs involved in partitioning QoS support between the adapter and the host, especially the impact of the relationship between available CPU processing capacity and network bandwidth.

**Receive Livelock:** In a poorly designed communication subsystem, a network host can suffer from *receive livelock* [17] under sustained network input overload. In this condition, since packet reception is performed at a very high priority, the host is swamped with handling arriving packets, to the extent that effective system throughput falls to zero. Receive livelock may be avoided via careful modifications applied to the operating system [16]. We used *END* to propose and evaluate *intelligent interface backoff*, a novel approach to eliminating receive livelock in which the adapter dynamically adjusts its interrupt generation rate based on host processing load.

## 7. Related Work

Our work relates to the following areas of research.

*Communication subsystem design and performance:* Several researchers have studied the issues affecting the design and performance of network adapters [6, 17, 21, 8], and communication subsystems in general [7, 19]. While many of these studies have influenced our work, we have focused as much on the design process, as the design itself.

*Simulation-based evaluation:* Recently, significant attention has been given to accurate, low-level simulation to study machine architectures while capturing operating system overheads [22, 2]. Other efforts have focused on protocol-level simulation with the ability to run the actual protocol stack during simulation [4], and network-level simulation with a focus on routing and end-to-end protocol performance [12, 14]. Most relevant to our work is architecture-level and protocol-level sim-

ulation. Since *END* executes on its own processor concurrently with the host, it avoids any intrusion on the host operating system 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 [3], with the models executing on a separate processor within the host. Besides modeling network source/sink behavior, 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 [23]. 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. This would provide a reasonable mechanism to study storage subsystem performance.

## 8. Conclusions and Future Work

In this paper, we described the architecture and implementation of *END*, a tool for designing network adapters via device emulation. We used *END* to accurately model 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 argued that *END* is a versatile tool that can be employed by device designers to compare and evaluate design alternatives before actually building a prototype.

We have implemented reception functionality in *END*, using both true two-way communication as well as synthetic traffic generation. We have also added a control interface to *END* through which protocol designers and applications can request various synthetic traffic patterns and exercise dynamic control over 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. To further enhance the usability of *END*, we are considering developing libraries for interrupt wrappers, FIFO and priority queues, and stochastic models for traffic sources. Finally, we would also like to extend *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] ANCOR Communications, Inc. *VME CIM 250 Reference/User's Manual*, 1992.

[2] R. C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of Sigmetrics 95/Performance 95*, pages 14–24, May 1995.

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

[4] L. S. Brakmo and L. L. Peterson. Experiences with network simulation. In *Proceedings of ACM Sigmetrics 96*, pages 80–90, May 1996.

[5] R. K. Budhia, P. M. Melliar-Smith, L. E. Moser, and R. Miller. Higher performance and implementation independence: Downloading a protocol onto a communication card. In *Proc. of the Intl. Conf. on Comm.*, June 1995.

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

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

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

[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] A. Indiresan. *Exploting Quality-of-Service Issues in Network Adapter Design.* PhD thesis, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109, October 1997.

[11] A. Indiresan, A. Mehra, and K. Shin. Design tradeoffs in implementing real-time channels on bus-based multiprocessor hosts. Technical Report CSE-TR-238-95, University of Michigan, Apr. 1995.

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

[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*, Nov. 1994.

[14] S. McCanne and S. Floyd. NS (network simulator), 1995. Available via http://www-nrg.ee.lbl.gov/ns.

[15] A. Mehra, A. Indiresan, and K. Shin. Structuring communication software for quality-of-service guarantees. In *Proc. 17-th Real-Time Systems Symposium*, pages 155–164, Dec. 1996.

[16] J. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Computer Systems*, 15(3):217–252, Aug. 1997.

[17] K. K. Ramakrishnan. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications*, 11(2):203–219, February 1993.

[18] 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.

[19] D. C. Schmidt and T. Suda. Transport system architecture services for high-performance communications systems. *IEEE Journal on Selected Areas in Communications*, 11(4):489–506, May 1993.

[20] J. M. Smith and C. B. S. Traw. Giving applications access to Gb/s networking. *IEEE Network Magazine*, pages 44–52, July 1993.

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

[22] E. Witchell and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM Sigmetrics 96*, pages 68–79, May 1996.

[23] 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*, pages 146–156, May 1995.