EXPLORING QUALITY-OF-SERVICE ISSUES IN NETWORK INTERFACE DESIGN

by

Atri Indiresan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1997

To my parents and Rohini.

ACKNOWLEDGEMENTS

I would like to thank everyone who supported and assisted me during my academic career at the University of Michigan. First and foremost, I would like to thank my advisor, Professor Kang G. Shin for his encouragement and support during my Ph.D. program. He allowed me to pursue my own research interests, and was always available to discuss various problems and provide feedback. In addition to his role as an academic mentor, he was always concerned about his students as individuals. I would like to thank Professors Peter Chen, Farnam Jahanian, Toby Teorey and Kimberly Wasserman for serving on my thesis committee and for their advice and support. A note of thanks to Professors Stuart Sechrest and Anthony Woo for serving on my thesis proposal committee. I would like to extend special thanks to Professors Farnam Jahanian and John Meyer for discussions on research, and their encouragement and friendship.

I would like to gratefully acknowledge the National Science Foundation and the Office of Naval Research for providing financial support during the course of my graduate program.

Many people have contributed to this dissertation in one way or the other. I had a very fruitful collaboration with Ashish Mehra, and our work together greatly contributed to this thesis. I would like to thank everyone who collaborated with me and contributed to the development of HARTS: Dilip Kandlur, Jim Dolter, Harold Rosenberg, Seungjae Han and Jaehyun Park.

The city of Ann Arbor and the University of Michigan gave me some of the best years of my life, and it is with deep regrets that I take my leave. It is a beautiful and vibrant place, but most important of all, it is where I met many wonderful people. I treasure the friendship of fellow students in the RTCL, including Ashish Mehra, Chao-ju Hou, Dan Kiskis, Dilip Kandlur, Harold Rosenberg, Jennifer Rexford and Sushil Birla. I will always remember Beverly J. Monaghan, the RTCL administrative assistant, for her friendship and many enjoyable conversations. Karen Liska, the EECS graduate secretary, always went beyond the call of duty to be helpful.

I cherish the friendship of the people whom I met during my stay in Ann Arbor. I wish to thank John Lynn, Melora Goosey, S. Mohan, Raghu Mani, Sushila Subramanian, Vidyullata Waghulde and many others for their friendship and hospitality.

The Shotokan Karate of Michigan has been an extremely important part of my life in Ann Arbor. While I learned a lot from everyone in the dojo, my special thanks to John Teramoto and to my instructors, Dean Askounis, Ravindra Prasad and Adam Liebowitz. Sergej Roytman, Toni Guzzardo, Sandeep Rao, Mary Prasad, Michael Sandoval, Tracy McComb, Tina Yee and many others – thank you for the wonderful times we had together, both in and out of the dojo.

My parents, Jayalakshmi and P.V. Indiresan, always taught me the importance of knowledge, and encouraged me to acquire it and use it wisely. Their example and their Ph.D.'s have always been an inspiration to me. I would like to thank them and my sisters, Nitya and Gowri, for their love, patience and understanding over the long years of my Ph.D.

The last few months of my Ph.D. were the hardest, but my wife, Rohini, made it all possible through her love and encouragement. She also spent endless hours reading multiple drafts of my thesis and helping me improve the presentation. I truly owe her a debt of gratitude for all that she has done for me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This dissertation examines the issues related to the design and implementation of communication subsystems, in particular, of network adapters, for quality-of-service (QoS). Applications with QoS requirements on communication (e.g., bandwidth, delay) are now being made possible by the advent of high-speed networks and the development of mechanisms for real-time communication. End-to-end communication performance is determined by a variety of factors, such as the underlying network technology, the end-host operating system, and the host-network interface. As network speeds increase, the performance bottleneck shifts to the end hosts, especially to the hardware and software components of the communication subsystem.

The tradeoffs involved in implementing communication services with QoS guarantees are explored using the example of real-time channels (RTC), a mechanism for providing guaranteed real-time communication services. RTCs were implemented using a commercial, off-the-shelf, network adapter, the Ancor VME CIM 250 (CIM), which does not provide support for QoS. Though it is demonstrated that this implementation does provide QoS guarantees, the throughput achieved using the CIM is much less than the capacity of its network link or the host's memory bandwidth. This is due to poor design of the CIM, including an excessively complicated host interface, and other architectural deficiencies.

An *Emulated Network Device* (*END*) is proposed and implemented to address the above deficiencies. *END* is a tool that couples a representative executable model of an adapter to a host and allows the host's communication software to interact with this model in real time. It is argued that *END* provides a sufficiently flexible environment for network adapter design and evaluation. A case study of the CIM is performed to demonstrate how *END* may be used to design and analyze network adapters before they are built, thus helping avoid costly design errors. A representative model of the CIM was built using *END*, and this model was modified by simplifying the host-adapter interface and adding dual-ported memory to the adapter. These design improvements increased the throughput of the CIM by up to 50%.

END is also used to study QoS issues in end-host communication subsystems by constructing models of adapters interfacing to point-to-point and shared networks. Various adapter configurations are considered for each network architecture to examine how QoS support may be divided between the host operating system and the network adapter. These experiments establish, particularly for shared networks, that QoS support on adapters significantly enhances delivered QoS.

Since a host can completely regulate data transmission, but does not have similar control over incoming data, the issues surrounding data transmission and reception can be quite different. In an interrupt-driven operating system, high packet arrival rates can result in *receive livelock*, a situation where the host uses all of its capacity to receive incoming data, and cannot usefully process any of it. Solutions to receive livelock typically involve extensive modifications to the host operating system. A novel solution is proposed that makes simple modifications to the device driver and network adapter, and does not require any modifications to the host operating system. Its feasibility and advantages are demonstrated by using *END* to compare it with various host-based solutions. This solution is sufficiently general that it can handle multiple network interfaces simultaneously and also guarantee minimum bandwidths to each interface even under extreme overload conditions.

The results obtained in this research demonstrate the versatility of *END* as a design and analysis tool by showing how it may be used for: (a) construction of representative models of existing adapters, (b) introducing and evaluating design improvements, (c) evaluating the efficacy of QoS support on the host and/or adapter, and (d) studying reception issues in interface design, in particular, the relative merits of host- and adapter-based solutions for avoidance of receive livelock.

CHAPTER 1

INTRODUCTION

Recent advances in high-speed networking have made it possible to realize a new class of real-time applications such as distributed multimedia services, remote medical diagnosis and sensor-based control systems. These applications have quality-of-service (QoS) requirements on end-to-end communication that are typically defined in terms of a desired minimum bandwidth and a maximum delay from the network; additional requirements on delay jitter and packet loss may also be specified [9]. In general, these applications require high data-transfer throughput and low, bounded, end-to-end delay. A high-speed network by itself cannot guarantee high application-level throughput and/or bounded data-transfer delays. In addition, precise characterization and control over system overhead in the end-host communication subsystem is required for guaranteed QoS.

End-to-end communication performance between two end-hosts connected by a network is largely a function of the end-host operating system, the interface between the host and the network, and the underlying network technology. Each of these components must provide support for QoS guarantees in order to meet application requirements. As network speeds increase, the performance bottleneck tends to shift to the end-hosts, in particular to the hardware and software components of the host's communication subsystem. The design of the network adapter (the hardware component that connects the host to the network), and the division of functionality between the adapter and the host communication software (the software component that connects the network to the applications), can have a significant impact on the performance delivered to applications.

There are several mechanisms for providing guaranteed real-time communication. In the simplest form, applications may simply specify their peak bandwidth requirements, and reserve that capacity for the entire duration of the connection, as in circuit-switched telephone networks. However, this can be very inefficient since real-time traffic is often bursty, and bandwidth utilization will be low unless idle time can be utilized by non-real-time traffic. Integrated services networks [19] are expected to carry a mix of traffic with different requirements on QoS. These networks typically use packet- or cell-switching technologies. In contrast to circuit-switched networks, due to statistical multiplexing of traffic, unused portions of capacity reserved by real-time applications can be reallocated to other applications.

Various service disciplines have been proposed that provide real-time guarantees in these networks [9]. Desirable characteristics of such service disciplines include the ability to provide real-time guarantees, low latency and jitter, ability to mix real-time and non-real-time traffic, scalability, low buffer utilization and high bandwidth utilization. In this dissertation, we study real-time communication using *real-time channels* (RTC) [43,63], a service discipline for packet-switched networks that provides end-to-end deadline guarantees to real-time traffic, while minimizing degradation of total bandwidth utilization or service to best-effort traffic. RTCs include admission control, traffic policing and scheduling, and buffer management techniques that ensure that real-time guarantees are met.

This dissertation deals with the issues involved in the design and implementation of end-host communication subsystems. In particular, it studies various aspects of network adapter design, and how its interaction with the host operating system influences delivered QoS. It explores these issues via an *Emulated Network Device* (*END*), a tool that creates a representative model of a network adapter and interfaces it to a real host. This permits real applications to be executed on the host, and to evaluate their performance accurately by capturing the overhead of the adapter-host interaction. This dissertation describes the architecture and implementation of *END*, and argues that it provides a sufficiently flexible environment to design and evaluate network adapters before building a prototype. The versatility of *END* is demonstrated by using it for: (a) the construction of representative models of existing adapters, (b) introducing and evaluating design improvements, (c) evaluating the efficacy of QoS support on the host and/or adapter and (d) studying reception issues in interface design, in particular, the efficacy of host- and adapter-based solutions for avoidance of receive livelock.

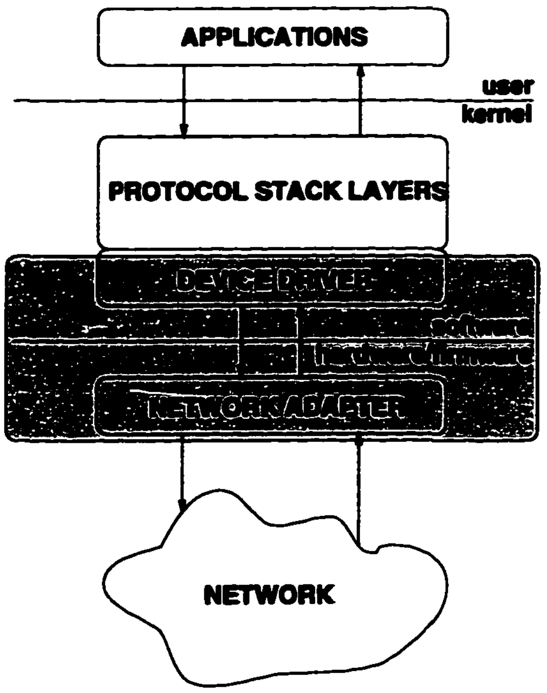


Figure 1.1: Typical communication subsystem at end hosts.

1.1 Scope of the Research

As mentioned earlier, 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 dissertation. Within 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 1.1). Data transfer typically 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, scheduling and synchronization mechanisms on the adapter, and the adapter firmware. The hardware features 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.

Several researchers have studied many of the issues above [14,33,41,83,95], and communication subsystems in general [38,39,89]. While many of these studies have influenced this work, they are often constrained by their host and network adapter architectures, thus preventing them from exploring novel architectural features. In contrast, our approach, using *END*, permits us to build software models of arbitrary network adapter architectures, and to study the behavior of applications as they interact with these models in real time.

This dissertation studies the interactions of the components of the communication subsystem, focusing in particular on the network adapter and its interface to the end host (the shaded region in Figure 1.1), and how they affect QoS. It is based on the premise that the design of network adapters should not be performed in isolation, but in the context of applications that require communication services and the architecture of the protocol stacks and operating system software. This implies that the design process is as important as the design itself.

The communication subsystem architecture also influences the implementation of service disciplines for real-time communication. These service disciplines are usually based on idealized theoretical models, and are not easily realizable on real systems, especially if it is necessary to support multiple classes of traffic efficiently. Implementing real-time communication services using commercial, off-the-shelf, hardware involves detailed analysis of the performance of the hardware and software components of the platform. There may be significant challenges and compromises involved in such an implementation since it is often necessary to overcome limitations of the platform, in particular, to provide QoS guarantees when the hardware does not provide explicit support for it. While there have been some implementations of real-time communications services, they have been constrained in many ways by their hardware and/or software characteristics. Some have been designed for particular classes of networks and are applicable in an extremely limited domain [66]. Others have been more general [11,12], but did not consider how best to exploit the underlying communication hardware. As a result, these implementations sacrifice either generality or performance. These problems may be further exacerbated due to *ad hoc* design of system components. The communication subsystem architecture must be designed in an integrated manner by considering the desired performance of the target system, and determining not only the required functionality, but also the division of such functionality between the hardware and software components, as well as their interface and interaction. Such a process is called *hardware/software codesign* [21,44,96].

Various techniques may be used in the design process. It is desirable that these techniques not only help solve the problem at hand, and for the given platform, but are also sufficiently general to handle a wide range of design and performance analysis applications for a variety of platform configurations. Further, these techniques must be easy to use without sacrificing generality, performance or accuracy.

The above-mentioned needs motivated the following research goals:

- To study the issues involved in providing QoS support on real platforms.
- To design tools and techniques to help integrate the study of these issues with the actual design and implementation of the communication subsystem.
- To demonstrate the versatility of these techniques by using them to study a variety of problems in the design of communication subsystems.

1.2 Contributions of the Dissertation

This dissertation makes several research contributions related to the design of communication subsystems.

Implementation of real-time communication services: We describe the architecture and implementation of real-time channels on HARTS [91,92], an experimentation testbed for studying architectural and operating systems issues in distributed real-time systems. We explore the tradeoffs between throughput and real-time behavior, and demonstrate how the host software can overcome the limitations of a best-effort network adapter to provide real-time communication guarantees.

Network adapter design: We show how hardware/software codesign may be used in the development of network adapters using *END*, an emulation-based network adapter design tool. *END* can capture low-level architectural details of network adapters, and interface them to real hosts. This enables the study of network adapters in the presence of real hosts, running real applications and protocol stacks. *END*-based models interact with the hosts in real time and are sufficiently detailed to study communication subsystems while including the effects of overhead like interrupts and cache behavior.

Modeling network adapters using *END*: A case study is performed using *END* to improve the design of a real network adapter. Experiments with the improved model of the adapter show throughput improvements of up to 50%. Further, most of the software used in building the model of the network adapter can be reused for the real network adapter, showing how significant pieces of software may be implemented and debugged, and their performance evaluated, even before building the hardware that would either run or interface to this software.

QoS issues in adapter design: *END* is used to build models of network adapters interfacing to point-to-point or shared networks. Various kinds of QoS support are implemented and divided in different ways between the host software and the adapter firmware. We establish that adapter-based QoS support significantly improves delivered QoS, especially in shared networks.

Receive livelock: We study various techniques for the avoidance of receive livelock. We propose a novel technique, ***adaptive backoff***, that eliminates receive livelock without requiring any modifications in the operating system scheduler; it only involves minor modifications to the adapter's firmware and device driver. This technique is very general, and works even when the host is connected to multiple network interfaces. The host's reception capacity may be partitioned between the network interfaces by suitable parameter settings. This partition of resources is not rigid, and a busy network interface can absorb reception capacity that is not used by a temporarily idle interface.

1.3 Outline of the Dissertation

The rest of this dissertation is organized as follows:

Chapter 2 surveys related work in the areas of real-time communication and network adapter design. In addition, it examines the pros and cons of various design and analysis techniques that may be used in designing communication subsystems while keeping in mind the goal of hardware/software codesign.

Chapter 3 describes our implementation of real-time channels on HARTS. The performance of a best-effort network adapter is characterized, and tradeoffs involved in using it to provide real-time guarantees are studied.

Chapter 4 addresses the problem of network adapter design using device emulation. Using a model of a generic network adapter, it considers the issues involved in the design of each component, and its potential impact on QoS. It describes in detail the design and implementation of *END*, an emulation-based network adapter design tool, and explains how it may be used to build models of arbitrary adapters, and to study their performance when interfaced to a real end host.

Chapter 5 uses *END* to study the performance of a real network adapter, the Ancor VME CIM 250, by building a representative model of it. Some of the performance bottlenecks due to the design of this adapter are identified, and the model is then modified to incorporate suitable design improvements and study the potential performance improvement.

Chapter 6 revisits issues on QoS support in communication subsystems. It describes a QoS-sensitive communication architecture that integrates CPU and link scheduling to provide QoS guarantees. *END* is used to build various models of network adapters interfacing either to a point-to-point network, or to a shared network medium. Various configurations of the host operating system, network adapter and networks are studied to determine how delivered QoS for transmitted data depends on the level of QoS support provided by the different components of the end-host communication subsystem.

While many of the issues in the design of transmission and reception parts of the communication subsystem are quite similar, there are also significant differences on account of the fact that the end host can completely control outgoing data, but it does not have similar control over incoming data, which could arrive at the network interface at any time, from one or more sources. Chapter 7 addresses some of these reception issues. In particular, *END* is used to study the problem of *receive livelock* and demonstrate how this problem may be solved by suitable modifications of the host scheduler and/or the network adapter.

Chapter 8 concludes this dissertation by recapitulating its contributions and suggesting possible directions for future work.

CHAPTER 2

COMMUNICATION SUBSYSTEM DESIGN

This dissertation deals not only with the actual design of network adapters for QoS, but also with the process of designing them. Designing a network adapter involves studying its architectural features, and how they interact with the other components of the communication subsystem, and how these features support or hinder the implementation of real-time communication services. Communication subsystems include host software like the protocol stack, parts of the operating system and the device driver, and hardware like the network adapter and its interface to the host. This chapter presents an overview of past research in areas related to end-host communication subsystem design, in particular for implementing QoS support. To provide appropriate QoS, it is not only necessary to select the appropriate service discipline, but also to implement them in a manner that minimizes their overhead and maximizes the admissibility and performance of real-time traffic. Section 2.1 provides a brief survey of various service disciplines for real-time communication, and Section 2.2 describes some implementations of real-time communication services and the issues addressed by them. Section 2.3 describes other research efforts involving network adapter design. Section 2.3.1 deals with various techniques that may be used in network adapter design, and the advantages and disadvantages of those techniques. Finally, Section 2.3.2 highlights the advantages of network device emulation as a design technique, and introduces *END*, which was implemented in the course of this research.

2.1 Real-time Communication

With the rapid growth of the Internet and the Web, and servers being used as sources for multimedia data, real-time communication services are becoming increasingly important. Some of the issues involved in providing these services have been studied extensively in other contexts like embedded systems and real-time CPU scheduling. Traditional packet switched networks have primarily addressed issues of network throughput, whereas real-time applications require stringent guarantees with respect to delay, delay jitter, throughput and packet loss. For instance, real-time applications obtain communication services from a network by having resources allocated to them. Factors that have a significant impact on communication services include routing, buffer management and packet scheduling algorithms. In addition, since the network has finite bandwidth, it must also enforce admission control to provide any kind of service guarantee. An application requesting service must therefore characterize its communication traffic and specify its QoS requirements. The network computes the resources required and can accept a connection if the resources are available. Once a connection has been established, the network's admission control policies preserves the QoS of already accepted applications. Thus, real-time applications place stringent demands on the communication subsystem.

A detailed survey of various proposed techniques for real-time communication can be found in [9]. Some of these techniques are presented here to provide an insight into the variety of paradigms used, and the issues involved in selecting a scheme appropriate to the system requirements. In *rate-based* methods, the requested QoS is translated into a transmission rate or bandwidth. The rate determines the priority given to a connection, and is used to determine the end-to-end delay bounds. *Scheduler-based* methods examine how the packets of different connections interact, and if it is possible to find a feasible schedule in which all packets meet their deadlines. Priorities are assigned dynamically for each packet at run-time based on their deadlines. While rate-based schemes are usually simpler to implement, the rate and data priority cannot be assigned independently. Scheduler-based schemes are more complex to implement, but allow greater flexibility in independently selecting bandwidth, deadlines and delay jitter.

Some of the notable rate-based schemes include Weighted Fair Queuing [36], Packet-by-Packet Generalized Processor Sharing (PGPS) [82], Stop-and-Go [48], Hierarchical Round-

Robin (HRR) [60], and Rate-Controlled Static-Priority Queuing (RCSP) [105]. Scheduler-based schemes include the Real-time Channel (RTC) [43] and its variants and enhancements [63,109,111]. While the schemes mentioned so far are for hard real-time communication, with strong guarantees, proposals have been made for predicted (or best-effort) real-time communication, which provide some kind of statistical guarantees. These include FIFO+ [28] and Hop-Laxity [90]. The Internet Engineering Task Force (IETF) is examining these issues in the context of providing integrated services on the Internet [19].

Generalized Processor Sharing (GPS) was first proposed as Weighted Fair Queuing (WFQ) [36]. This is a work conserving scheme that guarantees bandwidth to applications based on their average traffic rate. When combined with a leaky bucket admission scheme, a generalized form of the packet-based GPS (PGPS) [82] provides performance guarantees in a flexible environment. For leaky-bucket constrained sources, PGPS may be used to tightly bound the end-to-end delay [81]. While GPS has been proven to be optimal, and PGPS is a good approximation of it, these schemes are very complicated and hard to implement. Self-Clocked Fair Queuing (SCFQ) [46] is somewhat simpler, with comparable performance, but, in the worst case, does not guarantee fairness. These schemes often serve as a benchmark to compare against the performance of simpler schemes [88].

The Stop-and-Go [47–49,97] queuing framework includes a packet admission policy at the edge of the network, and a framing strategy at all nodes. This bounds the end-to-end delay, and guarantees a low jitter. The main drawback is that the end-to-end delay is tied to the sizes of frames, reducing the flexibility in satisfying different delay requirements.

VirtualClock [106,107] is a scheduler-based flow control mechanism that supports diverse performance requirements by enforcing resource usage based on prior reservations. Though the formulation of the algorithm is different, it is equivalent to the logical arrival time method used for policing traffic in RTCs [43,63]. Unlike RTCs, this scheme only provides guaranteed bandwidth for connections, but does not guarantee specific deadlines. However, extensions of the VirtualClock scheme have been used to compute end-to-end delay bounds as well [104].

Our approach to real-time communication is based on the *real-time channel*, a scheduler-based scheme. Since we consider platforms that can support a mix of real-time and best-effort traffic with a wide variety of QoS requirements, the advantages provided by the flexibility of this scheme outweigh the complexity of its implementation. The RTC was first proposed by Ferrari and Verma [43] as part of the Tenet project. RTCs provide end-to-end guarantees by computing link deadlines for each link along the path for the message. This method is valid under the assumption that the sum of all packet times is less than the shortest period of any connection using that link. This restriction was removed by Kandlur *et al.* [63]. They used a variant of the critical zone analysis used in the Rate Monotonic algorithm [71] to assign static priorities to existing connections, and to check if introducing a new connection would affect the guarantees of existing channels. Though static priorities are used to test for admission of new connections, a multi-class Earliest Due Date algorithm [43] is used for run-time scheduling. This scheduling scheme uses *logical arrival time* to set deadlines, and this provides flow control and also protects connections from one another. The details of our implementation of this scheme are described in Chapter 3. RTCs have also been adapted for other communication architectures. For example, extensions to the RTC scheme have been proposed for local area networks and FDDI rings [109, 110].

The examples selected here are not a comprehensive survey of real-time communication. However, they do illustrate that there are many different schemes for admitting packets, managing flow control and providing different kinds of guarantees for real-time packets.

2.2 Implementation of Real-time Communication Services

While there are many theoretical models for real-time communication, there are significant problems involved in actually implementing them. Most theoretical models of service disciplines deal with scheduling algorithms for data in packet- or cell-switched networks. Typically, they do not deal with issues like the implementation costs of such algorithms, the required CPU resources, CPU preemptibility and preemption costs, effects of CPU and cache behavior, protocol processing costs, the relationship between the memory and I/O bandwidths, and the interaction between the CPU and network link scheduling algorithms. Recently, there have been a few papers in the literature that address the issues related to the constraints imposed by real systems, and how the admission control and run-time scheduling of existing service disciplines need to be modified to provide QoS guarantees under these circumstances [50, 74, 75]. Given below are a few examples of implementations of real-time communication services, and how they address these issues.

The *Time-Triggered Protocol (TTP)* [66] is an integrated communication protocol for time-triggered architectures. It provides predictable delays, clock synchronization and fault-tolerant communication on replicated broadcast channels using time-division multiplexing. While this has been implemented successfully, it is narrowly targeted towards embedded systems which have workloads that can be precisely defined in advance. Though it is interesting to study how the system is designed by considering the issues related to the targeted application requirements and platform architecture, and their impact on fault-tolerance and scheduling, TTP is not sufficiently flexible to be directly applicable to more general systems.

The *Session Reservation Protocol (SRP)* [8] was proposed as a (compound) session establishment protocol for IP networks as part of the DASH project [5,6]. The *Tenet* real-time protocol suite [11] is a successor to DASH, and is an advanced implementation of real-time communication on wide-area networks (WAN). This protocol suite comprises the RCAP channel administration protocol and the RTMP/RTIP transport and network layer protocols, which implement unicast real-time channels [43] in Unix. Since Unix-based uniprocessor workstations are the implementation platform, the Tenet approach uses the socket application programming interface (API) and implements the real-time channel scheduling discipline for ordering packet transmissions. Tenet assumes that the network, rather than the end-host CPU, is the bottleneck, and does not address the problem of making protocol processing inside the host more predictable. While the Tenet implementation uses standard network adapters, it does not consider the impact of adapter characteristics on the ability to support real-time communication effectively. Though the effectiveness of these protocols in providing and maintaining bandwidth and delay guarantees has been demonstrated [12], delay jitter is quite large, probably due to the variance in the CPU processing delays.

The *resource ReSerVation Protocol (RSVP)* has been proposed for use in the Internet [108]. While SRP and Tenet[1] were geared towards unicast sessions with performance guarantees, RSVP is geared more towards multi-point multi-party communication. RSVP is a signaling protocol that permits resource reservation for real-time communication in the Internet. Currently, standards are being developed for two classes of traffic: (a) *Guaranteed service* which provides end-to-end bandwidth and delay guarantees, provided the connection does not violate the traffic parameters it provided, and (b) *Controlled load service* which provides looser guarantees, but tries to ensure that most packets meet their deadlines.

[1]Tenet Protocol Suite 2 [16,52] considers extensions for multi-point, multi-party communication.

An implementation of RSVP for Unix-based servers supporting Integrated Services [19] has been described by Barzilai *et al.* [13]. It is an enhancement of traditional sockets-based communications that preserves the API and binary compatibility of existing applications. It supports a wide variety of network interfaces ranging from legacy LANs, like Ethernets and Token Rings, to high-speed ATM interfaces. A key component of this implementation is a software module, called the *QoS Manager*, that is responsible for connection management as well as run-time shaping, buffer management and scheduling of traffic. Based on the QoS support provided by the network adapter, the QoS Manager can provide different levels of scheduling support needed for each connection. For example, the ATM adapter provides shaping and scheduling, and does not need software support for QoS. On the other hand, the Token Ring based network requires run-time support from the QoS manager since it does not provide this on its own. This implementation of RSVP also assumes that CPU resources are adequate, and does not provide any explicit CPU capacity reservation or scheduling support for protocol processing.

A common feature of the implementations described above is that they concentrate on implementing particular service disciplines and provide appropriate operating system support (API and link scheduling) for them. An orthogonal requirement is making protocol processing predictable within hosts. The need for scheduling protocol processing at priority levels consistent with those of the communicating application is highlighted in [7] and some implementation strategies demonstrated in [51]. More recently, processor capacity reserves in Real-Time Mach [78] have been combined with user-level protocol processing [72] to make protocol processing inside hosts predictable [79].

Some other implementations have used CPU scheduling to address some problems related to network interfaces [39, 80], but not in the context of QoS. Our implementation of real-time channels [56, 74, 75] is described in detail in Chapter 3. It discusses the efficacy of link scheduling for our hardware and software architecture, and explicitly considers issues related to the network adapter interface to the host. Chapter 6 presents a brief description of our QoS-sensitive communication architecture, and discusses how CPU and network scheduling were integrated on this platform [74, 75]. This provides a basis for a detailed examination of the issues involved in CPU and adapter support for QoS.

2.3 Network Adapter Design

The design and performance of network adapters, and communication subsystems in general, have been studied with a view to optimize performance and minimize the overhead related to the adapter-host interface. While early work was mainly targeted towards improving throughput and reducing latency, more recently, adapters have been designed with explicit support for QoS.

The *Aurora* project has studied the design and implementation of high-performance ATM adapters. Early work [35,98] focused on the hardware aspects of the design, and optimized performance by implementing fixed services like Segmentation and Reassembly (SAR) in hardware, and allowing parallel operations for each communication channel. A separate controller was provided for the transmit and receive directions, allowing a variety of scheduling and SAR algorithms to be implemented. Since ATM cells have a very small payload (48 bytes), "blocking" may be used to process more than one cell at a time. Polling for per-cell operations is also necessary since process contexts that need to be saved during interrupts are often larger than the cell size. Some operations were further optimized by using Programmable Logic Devices (PLD) for per-cell operations [34].

Later work on Aurora shifted the focus to software issues [38,93,99]. Per-cell operations were now moved completely to hardware, and the software abstraction of the hardware was that of a device that transferred arbitrary sized data between the host memory and the network. Device-dependent parts of the software were encapsulated in a device driver, with service primitives like sockets as the system entry points. Since operating systems typically have multiple address spaces, and data needs to be moved across them, solutions were proposed to minimize data copying. Other issues that were addressed were the interaction of cache behavior with Programmed I/O (PIO) or Direct Memory Access (DMA) transfers, and issues related to polling vs. interrupts, and hybrid solutions like clocked interrupts. This line of research was further developed in the *Osiris* project [41], which considered software issues from the perspectives of the adapter firmware, the operating system interface and the applications. Application Device Channels (ADC) were proposed as a means to give applications direct access to the network interface bypassing the overhead of the operating system.

The *Nectar* project [10,29,94,95] was another significant effort in the design of high-performance network interfaces for heterogeneous multicomputers. This has now evolved into the Gigabit Nectar project which focuses on network interfaces for supercomputers [54]. The Nectar CAB (Communications Accelerator Block) ensures high throughput and leaves sufficient resources for applications by moving some of the protocol processing to the network adapter. The design was influenced by the bimodal nature of data (most packets are either small, or are maximum sized), and appropriate optimizations were made for each kind of packet. Software optimizations included eliminating unnecessary data copying, and computation of checksums along with data movement.

Placing a network adapter on the system memory bus provides very tight coupling between the host and the network. However, such adapters are typically found only for very mature and ubiquitous technologies (e.g., Ethernet). Most other adapters lie on the system I/O bus, allowing them to be designed independently of the CPU architecture. One such network adapter for FDDI networks is described in [83]. It considers various design alternatives for partitioning the functions between the network interface and host software and proposes a simple model for predicting user-perceived throughput. It also demonstrated that straight-forward design of the operating system and network interface makes the system susceptible to *receive livelock*. Operating system modifications to avoid receive livelock are proposed in [80]. Other (partial) solutions based on modifications to the network adapter and/or operating system have also been proposed [39]. We study receive livelock and its solutions in detail in Chapter 7.

Issues similar to those described above have been studied in the other network adapter implementations including the Afterburner [33], Jetstream [42] and APIC [37]. QoS support in network interfaces has been studied in [18,30]. While most designs are qualitative in nature with measurements to verify performance, quantitative studies predict network performance based on the analysis of the system configuration and costs of individual communication subsystem components [74,76].

The examples discussed in this section present an overview of the design of specific network adapters and the design goals and issues addressed in each case. They illustrate the increasing importance of integration of hardware and software for optimal performance. However, in each case, the hardware was designed independently of the host software, and the software issues were addressed after the hardware was built. Adapters like Osiris and Nectar were built with programmable components specifically to permit software experimentation. While flexible designs like these are significantly more versatile, they are also very expensive to build, and might still be constrained by their hardware architectures. The rest of this section discusses the network adapter design process, and proposes our techniques for hardware/software codesign that permit evaluation of a hardware architecture even before it is built.

2.3.1 Adapter Design and Evaluation Techniques

A network adapter is a complex device whose design can have a substantial impact on communication performance. The process of designing network adapters is often an important part of meeting the design goals. 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. Building and testing hardware and interfacing to higher software layers in the operating system is time consuming and expensive. Most adapters do not allow on-board firmware to be modified or programmed to experiment with different design tradeoffs. More importantly, more often than not, the hardware engineers designing network adapters are far removed from the concerns of those writing communication software, resulting in a design poorly integrated with the host operating system.

Several techniques may be used to design and evaluate network devices (I/O devices in general): mathematical modeling, simulation, emulation and prototyping. Figure 2.1 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 overhead encountered in practice (such as interrupt handling and context switches). More detailed models capturing concurrency, contention, and dynamic component interaction have been constructed for some systems [23, 65], but these rapidly become intractable.

Another technique is *simulation*, which has several advantages [15]. 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

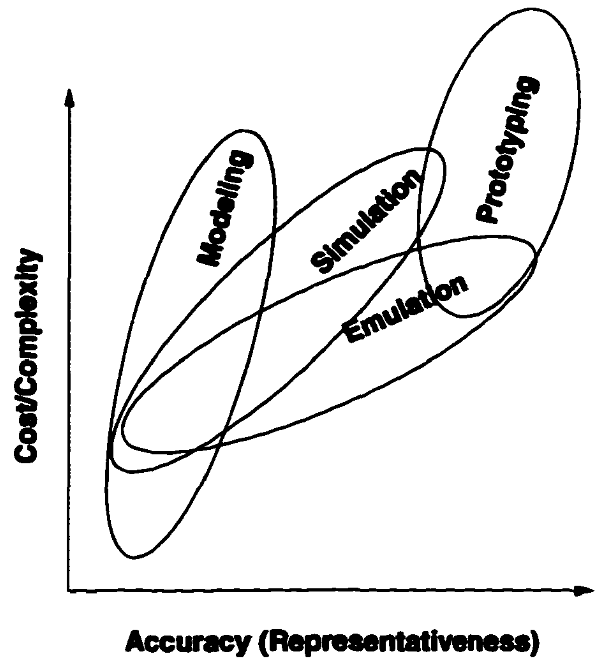


Figure 2.1: Performance evaluation techniques and tradeoffs.

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 [20]. 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. As seen in Figure 2.1, increasing the accuracy of the simulation model increases its cost while slowing it down even further.

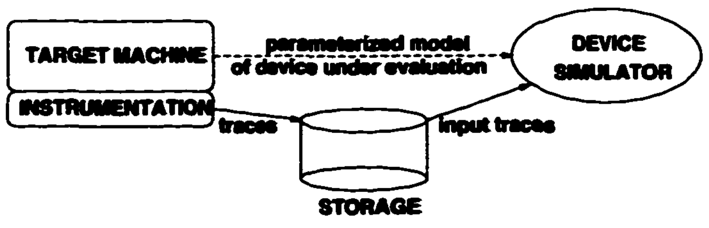
Prototyping a device and interfacing to higher software layers in the operating system is time-consuming and expensive. While prototypes can be highly accurate (see Figure 2.1), they are not easily modifiable. The on-board firmware may be modified to study different design options [41], or one may employ programmable adapters [22], 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.

A hybrid solution between simulation and prototyping is *emulation*. Device emulation is a technique that permits ***hardware/software codesign***, where adapter design tradeoffs can be explored early in the design cycle. While simulating a complete system, it is necessary to construct models of each component, and also capture all the interactions between these components. In contrast to a simulator, an emulator is a software module that interfaces to a real host (Figure 2.2(b)), giving the latter the impression that it is interacting in real-time with the actual subsystem being emulated. Device emulation shares the advantages of simulation in that it is a flexible technique that allows rapid design and evaluation of various interfaces and adapter design policies and/or algorithms. However, emulation is simpler than simulation since it is only necessary to create a model of the component being designed/evaluated, with the rest of the system comprising real components that already exist. Further, this may be as accurate as a prototype since it interacts with the rest of the system in the same way a real component/prototype would. As Figure 2.1 illustrates, device emulation can provide reasonable accuracy at an acceptable cost/complexity. Due to these advantages, emulation has been chosen as the design and evaluation technique used in this dissertation. The rest of this chapter presents a more detailed discussion of the advantages of emulation, and a comparison of *END*, the emulator used in this research, with other emulators used for network evaluation.

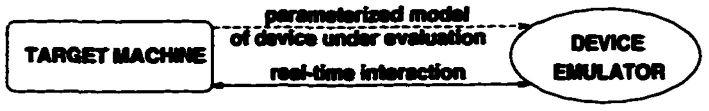
2.3.2 The Case for Device Emulation

Hardware/software codesign involves identifying the various functions that need to be implemented, and how to partition these functions between the hardware and software for optimal cost and performance [21,44,59,96]. Typically, hardware models are built using a hardware description language like VHDL, and simulating the software in conjunction with these hardware models [25,67,101]. Emulation is another hardware/software codesign technique, but, in contrast to the methods used in [25,67,101], instead of simulating the hardware model with the software, the software runs on the target host, and interacts with an executable model of the hardware in *real time* while capturing the details of the actual hardware/software interface.

Like simulation, device emulation is a flexible technique that allows rapid evaluation of various design alternatives. For a more accurate simulation, the target machine (for the



(a) Trace-driven simulation



(b) Device emulation

Figure 2.2: Trace-driven simulation vs. device emulation.

device being designed/evaluated) may be instrumented to generate run-time traces that are later fed as input to the device simulator (Figure 2.2(a)). Not only must the host software be instrumented to generate sufficiently detailed traces, but also 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. On the other hand, device emulation (Figure 2.2(b)) does not require that the target machine be instrumented, as long as it has the 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 [103].

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 overhead, 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.

Emulation has been used before for network evaluation. For example, a transputer-based network was used to emulate four hosts interconnected on a FDDI ring, and to evaluate a *multimedia network interface* design [18]. A more general-purpose emulator, *Hitbox* [1] is a layer just above the device driver in a network connected by Ethernets in a point-to-point configuration. Hitbox can be programmed to insert delays and/or errors to emulate the latency and noise characteristics of a WAN link. Of course, it can not be used to emulate links with a bandwidth greater than that of the Ethernet, i.e., 10 Mb/s. *Delayline* [57] is another WAN emulation tool. Here, an arbitrary WAN topology is superposed on a LAN configuration, and the application code is intercepted to insert the appropriate queuing delays. A single node of Delayline may be used to emulate more than one network host, or even a set of hosts on a LAN, and can handle more complex topologies than Hitbox. One of the disadvantages of Delayline is that it requires modification of the application code to manage these intercepts. *Dummynet* [85] is very similar to Delayline. Here again, arbitrary topologies are emulated on a LAN. A thin layer of code is added between protocol layers to capture the delays of the emulated network, thus making it completely transparent to the application code.

In each of the examples above, we see the advantages of emulation: using real components and operating system and network code to examine the behavior of entire systems that are different from the given platform in some aspect. However, they simply insert high-level queuing and data-loss models into the platform. The advantage of *END* is that it can not only be used in the same manner, but can also capture the low-level details of the host-adapter interface. The architecture of *END* is described in detail in Chapter 4. Examples of its use as a design and evaluation tool are presented in Chapter 5.

CHAPTER 3

DESIGN TRADEOFFS IN IMPLEMENTING REAL-TIME CHANNELS

The *real-time channel* (RTC) model [43, 63] provides a paradigm for real-time communication services in packet-switched networks. In this model, an application requesting service must specify its traffic characteristics, including the rate at which data is generated and QoS requirements, to the network. Since the network has finite bandwidth, it must perform admission control to provide any kind of service guarantees. The network computes the resources required and accepts the request if sufficient resources can be reserved for it. Once a requested RTC has been established, the network's policing and enforcement policies prevent an application from consuming more network resources than reserved, so as not to affect the services offered to other applications. RTCs provide delivery-delay guarantees for real-time traffic and, at the same time, allow reasonably good performance for best-effort traffic.

This chapter examines design tradeoffs involved in implementing RTCs in end-hosts using commercial, off-the-shelf, adapter hardware for access to the network. With multiple independent RTCs, it becomes important to minimize the interference between them, and to isolate real-time traffic from best-effort traffic. This requires that system resources such as memory bandwidth, protocol processing bandwidth, and network bandwidth be consumed according to a (dynamic) global transmission/reception order as determined by the QoS requirements and traffic load of the individual RTCs. The implementation of RTCs described here is on a bus-based multiprocessor system. Although most of the design principles do not depend on whether the end-hosts are uniprocessors or multiprocessors, cases where they differ have been highlighted.

A hardware and software architecture for real-time communication is presented that guarantees communication processing bandwidth by dedicating a processor for protocol processing and link scheduling. The application programming interface (API) for accessing communication services is split between this protocol processor and the other (application) processors. Dedicating a processor for protocol processing has also been proposed by other researchers [29] in order to offload all protocol processing from the application processors, freeing them from adapter handshake overhead and permitting greater overlap between useful computation and communication processing. In addition to these benefits, a dedicated protocol processor enables global coordination between the active real-time channels to schedule protocol processing, link access and data transfer bandwidth. To ensure that protocol processing bandwidth is consumed in a global transmission/reception order, the protocol processor provides priority-based scheduling of protocol threads. Similarly, access to the link is regulated through link scheduling on the protocol processor. On a uniprocessor system there is no need to coordinate between the activities of multiple processors. However, having a dedicated processor for network activities trivially guarantees a minimum processing bandwidth for these activities. On a uniprocessor system, other approaches, such as *Processor Capacity Reserves* [68, 77], could be used to ensure that sufficient CPU capacity is available for protocol processing. We optimize the data transfer path such that there is no unnecessary data copying and bus bandwidth on transmission is consumed in the link-access order determined by the link scheduler. Note that bus bandwidth is a concern on uniprocessor hosts as well, since most network adapters are on the system I/O bus, rather than on the CPU's memory bus [83], and hence will need to share the bus with other I/O activities.

Three aspects pertaining to the performance of real-time and best-effort traffic on our hardware and software architecture are explored. For this, we use a VMEbus-based multiprocessor as the end host, with hosts connected by a network constructed using an Ancor CXT 250 crossbar switch [4] and Ancor CIM 250 network adapters [3]. We highlight the performance implications of the design features and interface characteristics of the network adapter, especially for real-time communication. These observations are used to motivate desirable features in network adapters to support real-time communication, and hence, the implementation of real-time channels. Next, we consider the software overhead of protocol processing and link scheduling on the (dedicated) protocol processor. When there is no copying of data during protocol processing, protocol-processing and link-scheduling overhead is directly proportional to the number of fragments and the per-fragment processing cost. Using the overhead as motivation, simple CPU scheduling mechanisms are proposed to preserve QoS guarantees to real-time channels. Lastly, we study the effectiveness of the link scheduler in preserving QoS guarantees on individual channels as well as servicing best-effort traffic under varying traffic loads.

The rest of this chapter is organized as follows. Section 3.1 describes the hardware and software organization of our experimentation platform and Section 3.2 gives an overview of the real-time channel implementation on this platform. The implications of network adapter characteristics, including data-transfer performance for real-time and best-effort traffic, are discussed in Section 3.3, which also highlights desirable features in network adapters to facilitate real-time communication. Section 3.4 describes the optimizations we applied to minimize/eliminate redundant data copying such that bus bandwidth is consumed in the global transmission order determined by the link scheduler. The protocol-processing and link-scheduling overhead of our implementation is presented next, along with CPU scheduling mechanisms to preserve QoS guarantees. Section 3.5 evaluates the implementation of the link scheduler controlling access to the network and demonstrates how the scheduler not only insulates real-time traffic from best-effort traffic, but also insulates traffic belonging to different real-time channels from each other. Section 3.6 concludes this chapter.

3.1 The Experimentation Platform

This section describes the hardware and software architecture of the experimentation platform which is being developed as a part of the HARTS project [92]. The primary goal of HARTS is to investigate architectural and operating system issues in distributed real-time computing.

3.1.1 Hardware

Each HARTS node (also referred to as end host) is a VMEbus-based multiprocessor with 2–4 processors, as shown in Figure 3.1. This multiprocessor configuration provides several benefits over uniprocessor configurations. For example, many input/output devices and controllers are available for the popular VMEbus. Hence, each processor can be dedicated

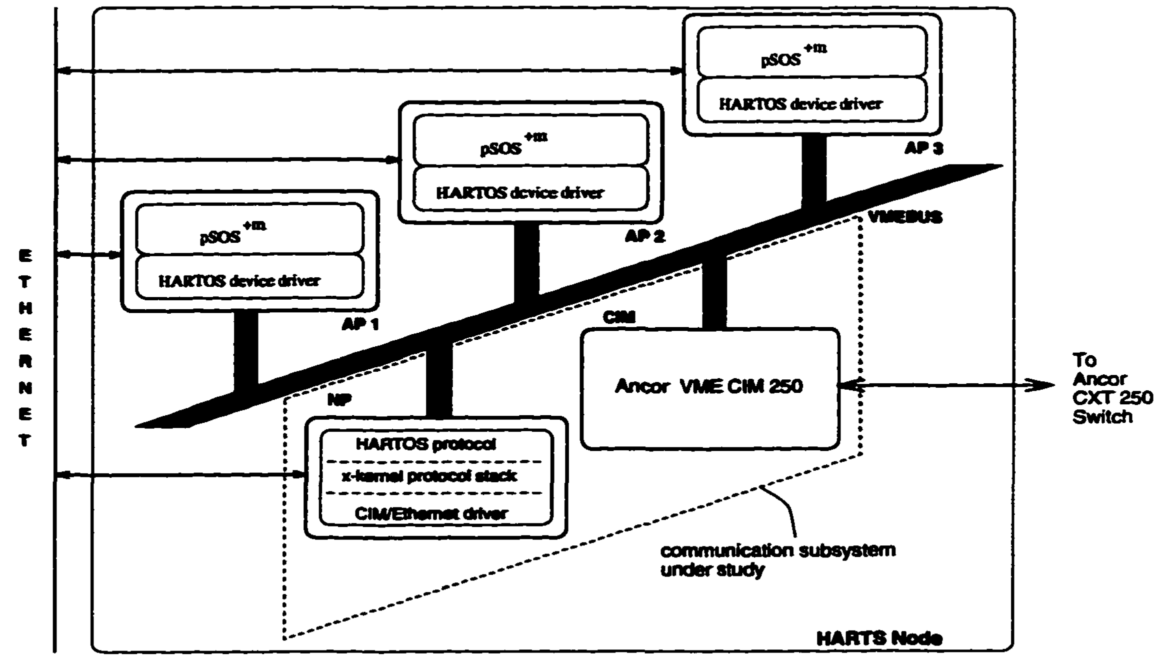


Figure 3.1: Architecture of each HARTS node

to control a different device on the VMEbus, enabling simultaneous control over the active devices. The additional processors can also provide fault tolerance at each node. Bus-based multiprocessor configurations are increasingly being used as multimedia servers and in desktop workstations. This architecture, therefore, allows us to derive important implications for platforms likely to support real-time communication.

The available processors in each HARTS node are divided into Application Processors (AP) and a Network Processor (NP); applications execute on APs while communication protocols execute on the NP. Dedicating a processor to control the VMEbus-based communication devices has several advantages. The NP offloads all communication processing from the APs, frees the APs from device handshake overhead, and permits greater overlap between useful computation and communication processing[1]. Since the communication device is located on the VMEbus, any processor can interact with it. However, device handshake overhead is minimal if done by a designated processor. More importantly, with real-time channels originating from multiple APs, a dedicated NP ensures that the channels are serviced in a certain global (node-wide) order as determined by their traffic parameters. Each processor is an Ironics IV-3207 card with a 25 MHz Motorola MC68040 CPU (SpecInt

[1] Communication processing in general includes clock synchronization, group communication, and real-time communication services in addition to protocol processing.

92 rating of 12.3); the NP has 16 MB of DRAM while each AP has 4 MB of DRAM.

In the current configuration, the HARTS interconnection network is constructed using an Ancor CXT 250 crossbar switch [4] and Ancor VME CIM 250 network adaptors [3], which implement the ANSI Fiber Channel 3.0 standard [2]. The CXT 250 crossbar switch is fully connected, but may be used to embed various partially-connected point-to-point topologies for studying multi-hop communication. In addition to communication interface hardware, the CIM has 8 MB DRAM, independent DMA controllers for data movement, and an input/output processor that provides support for Fiber Channel operations. Though the CIM has a general-purpose I/O processor, the on-board firmware is controlled by the manufacturer and cannot be modified by the user. The NP exercises control over the CIM only through command/response FIFOs.

3.1.2 Software

HARTOS [61,91], the operating system running on each HARTS node, provides a uniform interface for application programs to access kernel and network services, and supports real-time applications in a distributed environment. Figure 3.1 highlights the main HARTOS components. The APs run the pSOS^{+m} kernel [58] while the NP runs a protocol stack based on the x-kernel [55]. Communication between the APs and the NP is provided via the HARTOS API, a command/response interface that permits pSOS^{+m} and x-kernel to provide network services to applications.

AP Kernel: pSOS^{+m} is a real-time multiprocessor OS kernel and serves as the executive for each AP. Though pSOS^{+m} can provide network services like TCP/IP that may be used for remote communication with pSOS^{+m} tasks on other HARTS nodes, these services are not real-time and hence not suited for this platform. The real-time protocols could have been implemented in pSOS^{+m} using sockets as the real-time channel API, similar to the approach adopted by the Tenet group [11]. However, this would have limited us to uniprocessor configurations, with the accompanying process scheduling interference effects, and the semantics and associated overhead of the socket API. More importantly, an expensive coordination amongst the APs may be necessitated to determine the (node-wide) global transmission order. HARTOS extends pSOS^{+m} to operate in the multicomputer environment of HARTS. HARTOS provides a pSOS^{+m} device that reads from and writes to a command/response mailbox interface (HARTOS API) for services provided on the NP,

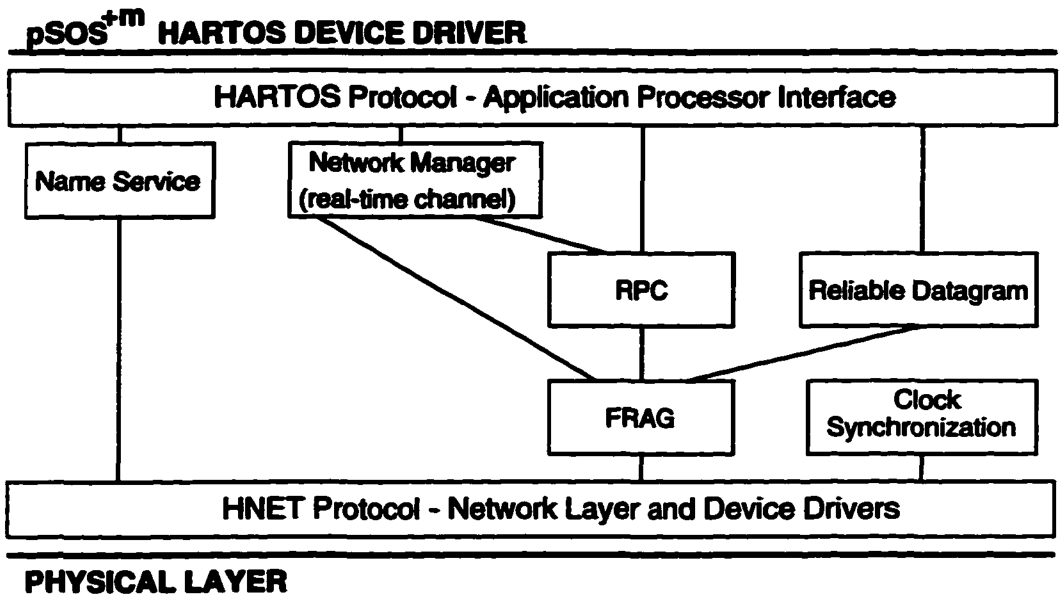

pSOS+m HARTOS DEVICE DRIVER
HARTOS Protocol - Application Processor Interface
Name Service
Network Manager (real-time channel)
RPC
Reliable Datagram
FRAG
Clock Synchronization
HNET Protocol - Network Layer and Device Drivers
PHYSICAL LAYER

Figure 3.2: The *x*-kernel protocol stack in HARTOS.

such as real-time communication and distributed name service. The HARTOS API is split between the APs and the NP, with API stubs marshaling call parameters implemented on the AP and the interface mailboxes implemented on the NP.

NP Kernel: The NP employs a derivative of the *x*-kernel [55] as the communication executive. It employs a *process-per-message*[2] model for protocol processing, in which a process or thread shepherds a message through the protocol stack. This eliminates extraneous context switches encountered in the *process-per-protocol* model [89]. A process-per-message model also allows protocol processing for each message to be independently scheduled on the processor based on a variety of scheduling policies, as opposed to the software-interrupt level processing in BSD Unix [69]. This improves the traffic insulation between different real-time channels.

Figure 3.2 gives an overview of the *x*-kernel protocol stack implemented in HARTOS. The HARTOS protocol interfaces with the HARTOS device driver on the APs to implement the HARTOS API. The Name Service protocol provides facilities to register a name locally, and to look up a name globally. Communication protocols include standard support like remote procedure call (RPC), reliable datagrams and fragmentation. The RPC and fragmentation (FRAG) protocols are modified versions of *x*-kernel's CHAN and BLAST protocols, respectively. Our implementation of the Clock Synchronization protocol uses a

[2]In Chapter 6 we extend the process-per-message model to a process-per-channel model.

System Parameter	Latency
context switching	20 μs
interrupt handling	28 μs
timer read (resolution)	22 (2) μs
NP PIO (4 KB bcopy)	350 ns per word
VME PIO (4 KB bcopy)	2040 ns per word
VME DMA (4 KB transfer - peak)	100 ns per word
CIM DMA (2 KB transfer)	1250 ns per word
CIM DMA (4 KB transfer)	2250 ns per word

Table 3.1: Baseline system performance

derivative[3] of Cristian's probabilistic clock synchronization algorithm [31], while the Network Manager protocol is the resource reservation protocol for real-time channels. These two protocols together support real-time communication services. The HNET protocol is an unreliable datagram service with addressing support for the underlying network. The HNET layer includes device drivers for the communication devices in a HARTS node (for access to Ethernet or CIM/CXT network) and implements run-time packet scheduling for network access.

3.1.3 Baseline System Measurements

Table 3.1 lists the baseline measurements on the NP, the VMEbus, and the CIM. The memory bandwidth was measured using the bcopy[4] operation, both within the NP and across the VMEbus. The CIM has been reported to deliver a maximum throughput of 6 MB/second only for very large ($\approx$ 3 MB) transfers [70]. While the throughput we obtained for smaller packet sizes ($\leq$ 16 KB) was similar to that obtained in [70], we could only obtain a data transfer bandwidth of about 3 MB/second with 3 MB transfers.

[3]This implementation of clock synchronization is completely in software, but it uses some ideas from the *Hardware Assisted Software Clock Synchronization* proposed by Ramanathan *et al.* [84] that help both to reduce the synchronization error and to increase the probability of synchronization. Even without any hardware assistance, the HARTS nodes maintain their clocks within 1 millisecond of each other.

[4] bcopy performs the transfer a word at a time, using programmed I/O. We designed the experiments carefully to minimize cache effects and adjusted the measurements to account for call and loop overhead.

3.2 Implementation of Real-time Channels

This section describes the implementation of real-time channels in HARTOS. First, the application programming interface is described, followed by the required support for channel establishment, and then the transfer of data from the sending AP to the destination AP through the intermediate nodes. Next, the optimizations applied to the transmission and reception paths through the x-kernel to minimize/eliminate data copies are discussed. Finally, the implementation of the link scheduler that coordinates packet transmission/reception and traffic enforcement at each node is presented.

3.2.1 Real-time Channel API

Applications create and use real-time channels through the real-time channel API (Table 3.2). The receiving task of a real-time channel invokes `rtc_init` to create a local pSOS^{+m} message queue for storing incoming messages. The receiving task subsequently registers the queue with the name service so that the sending task can locate it in order to create the real-time channel. The sending task establishes a real-time channel by invoking `rtc_create`, specifying the traffic parameters for the message generation process and the end-to-end delay bound desired on this channel. The traffic generation model is based on a *linear bounded arrival process* [5,32], in which the arrival process has the following parameters: maximum message size (S_{max} bytes), maximum message rate (R_{max} messages/second), and maximum burst size (B_{max} messages). In any time interval of length t, the number of messages generated may not exceed $B_{max} + t \cdot R_{max}$. Message generation is bounded by the rate R_{max}, and its reciprocal, I_{min}, is the minimum (logical) inter-generation time between messages. The burst parameter, B_{max} bounds the allowed short-term variation in message generation, and partially determines the buffer space requirement of the real-time channel. Non-periodic message generation can be represented in this model using an estimate of the worst-case inter-generation time and the average rate of generation. To ensure that a real-time channel does not use more resources than it reserved, this model defines the deadline guarantees by forcing a message inter-arrival time of I_{min}. This is achieved by defining the *logical generation/arrival time*, $\ell(m_i)$, for the i^{th} message, m_i, as:

$$\begin{aligned}\ell(m_0) &= t_0 \\ \ell(m_i) &= max\{(\ell(m_{i-1}) + I_{min}),\ t_i\}.\end{aligned}$$

Routines	Invoked By	Function Performed
`rtc_init`	receiving task	create local pSOS^{+m} queue to receive messages
`rtc_create`	sending task	create real-time channel with given parameters to remote task (queue); return channel ID
`rtc_send`	sending task	send message on the specified real-time channel
`rtc_recv`	receiving task	receive message from real-time message queue
`rtc_close`	sending task	close specified real-time channel

Table 3.2: The real-time channel API

where t_i is the actual generation time of message m_i. If d is the end-to-end delay bound for a channel, the system guarantees delivery of message m_i by $\ell(m_i) + d$. The logical generation time, $\ell(m_i)$, is the earliest time that m_i would have arrived if the maximum message rate constraint were strictly obeyed.

The call to **rtc_create** returns a local channel identifier on successful creation of the real-time channel and an error indication otherwise. Data transfer on an existing real-time channel is achieved by using the **rtc_send** and **rtc_recv** calls. The task invoking **rtc_send** is blocked until the data has been transmitted into the network; **rtc_recv** can be blocking or non-blocking. The sending task can tear down the real-time channel by invoking **rtc_close** with the local channel identifier; all the resources allocated to the channel are released at this point. This implementation splits the real-time channel API between the APs and the NP. **rtc_init** and **rtc_recv** execute entirely on the AP, which runs the receiving task, while the rest of the calls execute partly on the AP and partly on the NP. This allows serialization of channel establishment, data transfer and channel teardown on the NP, while allowing APs to exploit as much concurrency as possible. The routines available for best-effort data transfer (not shown) include **rdata_send** for transmission and **rdata_recv** for reception, with the same blocking semantics as **rtc_send** and **rtc_recv**, respectively.

3.2.2 Channel Establishment and Teardown

Upon receiving a request for a real-time channel, the network's channel establishment procedure must reserve adequate link bandwidth, buffer space, and protocol processing bandwidth from source to destination. This involves selecting a suitable path between the sending and receiving nodes, checking if adequate network resources are available on each node on the selected path, and reserving them along each such node. A channel is considered established if the necessary resources have been reserved at each node in the selected route, and the sum of link delays along the channel's path is less than the application-specified end-to-end delay bound.

A scheme for channel establishment in point-to-point networks may be found in [62,63]. In this scheme, there is a global network manager that maintains information about the network topology and resources and all established real-time channels. Applications send requests for channel establishment to this network manager which decides the route and reserves resources along the path of the real-time channel. The main advantage of a global network manager is that it has total knowledge of the system's resources and can potentially allocate resources optimally. However, it can be a performance bottleneck, especially in a large network, and it is also a single point of failure and is not suitable for fault tolerant systems. These problems may be resolved by using some kind of a distributed network manager. However, a distributed network manager may not be very efficient. If there are many routes between a source and destination, it may be too time consuming to search through all alternatives, and if all possibilities are not explored, it is possible to select a non-optimal route, or even fail to find a route when one exists. A distributed network manager that addresses these issues has been described in [27].

Our implementation uses a distributed network manager comprising network manager protocols (NMP) running on each node in the network. The NMP provides channel management services to establish and tear down real-time channels. Each NMP maintains only information about the real-time channels passing through its node. Invocations of `rtc_create` transfer control to the NMP, which must now determine if the requested channel can be established or not. Each NMP computes the smallest delay it can guarantee to this channel at the current node, and passes on the request to the next NMP along its route[5] with the balance of the delay available for it. Each NMP reserves sufficient resources (link and CPU bandwidth, network buffers) along the route to accommodate the channel's timing constraints. NMP uses the underlying RPC protocol (see Figure 3.2) to implement channel management requests. The per-link data structures maintained by the NMP on each node include a list of channels using the link and status information used in run-time scheduling (see Section 3.2.4). Each channel in the list stores a unique network-wide channel identifier (a `<node_id, local_channel_id>` tuple), traffic specification from the establishment

[5]The NMP does not address the problems of routing, and simply uses pre-computed, static routes.

1. Select the next link along the source–destination route.
2. Use algorithm D_Order [63] to compute the worst-case delay at this link. Assign the channel the highest possible priority that does not violate guarantees of existing channels. Also compute and reserve adequate buffer and processing resources.
3. Check if the link delay is less than the end-to-end delay. Reduce the end-to-end deadline by the link delay.
4. Relay the channel establishment request to the next node with the reduced deadline.

Figure 3.3: Channel establishment procedure – forward phase

request, the local link delay bound, and the local buffer requirements.

Channel establishment occurs in two phases: the *forward phase*, which propagates the establishment request towards the destination, and the *reverse phase*, which propagates the establishment reply back to the source to commit or release resources at intermediate nodes, in case the establishment request succeeds or fails, respectively. Figure 3.3 outlines the forward phase of the channel establishment procedure. Given a particular source-destination route, channel establishment is performed using a fixed-priority scheme (algorithm D_Order [63]). We consider only static routes for real-time channels since it is very difficult to provide any message-delivery delay guarantees for a channel based on dynamic routing. Channel teardown is triggered by an rtc_close call. The NMP sends a teardown request containing the channel identifier along the path of the channel. At each node along the path, all reserved resources are freed and made available for other channels.

3.2.3 Data Transfer

Once a real-time channel is successfully established, the application triggers data transfer on the channel by sending a message using rtc_send. Data transfer occurs only from the source to sink since real-time channels are unidirectional in nature. Moreover, the unreliable-datagram semantics of real-time channels imply that data is transferred without retransmissions and acknowledgments. After the initial marshaling of call parameters on the AP, control transfers to the NP which performs the protocol processing and subsequent transmission of the packets belonging to this message. The transmitted packets are relayed by each intermediate node into the network. Upon arrival at the destination node, the NP reconstructs the message by reassembling the packets and deposits it into the appropriate receive queue on the destination AP. The receiving task subsequently invokes `rtc_recv` to retrieve the message.

The HARTOS API on the NP initiates transmission protocol processing by firing up a protocol thread to shepherd the message down the protocol stack to the network. The protocol thread is scheduled for execution by the *x*-kernel thread scheduler and runs non-preemptively until completion of protocol processing. Protocol processing of messages includes assignment of deadlines and encapsulation by the NMP, packetization by the FRAG protocol, and network-level encapsulation by the HNET protocol. Each packet is then scheduled for transmission by the link scheduler. The message manipulation routines in *x*-kernel are modified to associate the original message deadline with each packet, enabling the link scheduler to correctly order the transmission of packets onto the network. The sending task is blocked until the data has been successfully transmitted by the CIM. The transmission protocol thread exits after handing the last packet of the message to the link scheduler. The AP unblocks the sending task when the NP indicates that the data has been transmitted into the network.

Protocol processing is initiated by a protocol thread on the destination NP when the CIM announces receipt of a packet. The received packet is shepherded upwards through the protocol stack by a protocol thread after stripping the CIM header. All but the last packet of a message traverse up to the FRAG layer, which strips the FRAG headers and queues the received packets for reassembly when the last packet arrives. The thread shepherding the last packet continues non-preemptively through FRAG, NMP, and the HARTOS API before delivering the message to the correct receive queue on the destination AP.

3.2.4 Run-time Link Scheduler

Traffic management at each node involves *run-time scheduling* to order packet transmissions such that the guarantees made to all established real-time channels passing through that node may be met. *Traffic enforcement* is also a responsibility of run-time traffic management, which must take appropriate action when a real-time channel violates its traffic specification. In general, the link scheduler must (a) *maintain guarantees*, i.e., ensure all real-time packets meet their deadline as long as they do not violate their input specification, (b) *perform traffic policing*, i.e., prevent channels that violate their traffic specifications from affecting the performance of well-behaved channels, and (c) *ensure fairness* in the delay and throughput delivered to best-effort traffic. The run-time link scheduler controls access to the outgoing link and determines the order in which packets depart from the node. At the source NP, the transmission protocol thread deposits the packets of the outgoing message into link scheduler queues and exits, as explained earlier. At intermediate nodes, the reception protocol thread relays the incoming packet to the link scheduler at the HNET layer[6]. At destination nodes, received packets bypass the link scheduler completely.

The link scheduler is implemented as a special `scheduler` thread that is created at system startup and runs at the highest possible priority in the x-kernel; each link has its own `scheduler` thread. The link scheduler is invoked in two situations: (i) new packets are deposited into the scheduler queues, and (ii) packets that had arrived early are now current. Situation (i) is handled by controlling the execution of each `scheduler` with a *scheduler semaphore*. Protocol threads depositing new packets in the scheduler queues perform a `V` operation on this (counting) semaphore to trigger the execution of the `scheduler`. Since it has the highest priority, the `scheduler` runs as soon as the currently executing protocol thread either completes execution or blocks. Situation (ii) is handled by registering an event with the x-kernel to wake up the `scheduler` at the correct time.

The link scheduler maintains three queues (Queue 1, Queue 2, and Queue 3) in which outbound packets are inserted by protocol threads [63]. Queue 1 contains *current* real-time packets (whose logical arrival time is less than the current clock time). Best-effort packets are inserted in Queue 2. Queue 3 contains real-time packets which have arrived early, either because of bursty message generation or because they encountered smaller delays at upstream nodes. Packets in Queue 3 are transfered to Queue 1 as they become current. The scheduling algorithm improves best-effort performance by giving Queue 2 priority over Queue 3. Protocol threads delivering real-time packets to the `scheduler` insert the packets into Queue 1 if they are current and signal the `scheduler` before exiting. If the packets are early, they are inserted into Queue 3 and an event is registered with the x-kernel to signal the `scheduler` when the packet at the head of Queue 3 is eligible for transmission. Not signaling the `scheduler` immediately saves unnecessary context switches. Protocol threads delivering best-effort traffic simply deposit the packets into Queue 2 before signaling the `scheduler`. The scheduler semaphore's count is incremented by one each time a packet is

[6] Since this is a multicomputer platform, each node must also handle traffic passing through it. In general, though, intermediate traffic would be handled by network gateways and/or switches.

inserted into the scheduler queues. Queue 1 and Queue 3 are implemented as priority heaps, with Queue 1 ordered by packet link deadlines and Queue 3 ordered by the logical arrival time. Queue 2, on the other hand, is implemented as a FIFO queue so that best-effort packets are transmitted in order of their arrival. Packets violating traffic specifications can be buffered at the source node (effectively delaying them), or forwarded with their deadlines relaxed so that they will be buffered longer at downstream nodes, or simply dropped.

On creation, the `scheduler` blocks on its associated semaphore, pending further packet insertions. If it has packets to transmit, it continues execution and does a `P` operation on the link's *write semaphore* to obtain access to the link and initiate packet transmission. The write semaphore is a counting semaphore associated with each link and limits the number of outstanding packet transmissions (it is initialized to 2, as will be concluded in Section 3.3.1). The `scheduler` blocks again if there are two or more outstanding packets awaiting transmission on the CIM. Once it obtains access to the link, the `scheduler` first examines Queue 3 and transfers all packets that have become current to Queue 1. It transmits the packet at the head of Queue 1 if it is non-empty, else, it transmits the packet at the head of Queue 2. If Queue 1 and Queue 2 are both empty, and the packet at the head of Queue 3 has a logical arrival time beyond the *link horizon*[7], the `scheduler` releases the link's write semaphore and registers an event with the x-kernel indicating that it be woken up when the head of Queue 3 is eligible for transmission.

3.3 Influence of Network Adapter Characteristics

Transmission/reception performance is significantly affected by the design features and interface characteristics of the network adapter. The characteristics of the interface exported by the adapter directly determine the efficiency and flexibility with which data transfer to/from the network can be initiated and coordinated. Interface characteristics, therefore, have a significant impact on supporting real-time communication. An adapter's interface characteristics are partially determined by design features such as support for DMA and provision of large on-board memory. Using the CIM as an example, we discuss the effect of network adapter design features and interface characteristics on data transfer performance, medium access latency, and packet handling on reception. Based on these insights, we

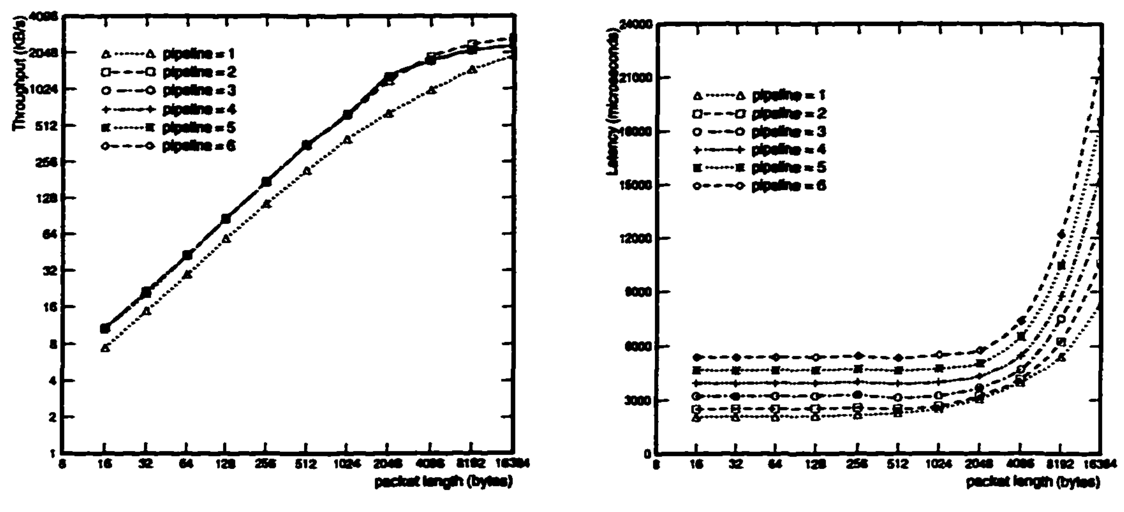
[7]The link horizon [63] is a parameter that controls the extent to which the scheduler is work conserving.

highlight some desirable design features for adapters to support real-time communication.

3.3.1 CIM Performance Characteristics and Implications

Several experiments were performed to determine the performance characteristics of the CIM, namely, the factors that affect data throughput and medium access latency (delay to access and use the network link). Packet size for transmission/reception and the number of outstanding packet transmissions (referred to as the *pipeline depth*) are two factors that significantly affect the performance of the CIM. In the experiments performed, a test application running directly above the HNET layer on one NP sends over 12,000 packets via the CIM to a peer application on another NP as fast as possible, while limiting the number of outstanding packet transmissions. The experiment is repeated for different packet sizes. Figure 3.4(a) plots the achieved throughput, and Figure 3.4(b) plots the medium access latency, as a function of packet size and pipeline depth. The medium access latency measures the time between initiation of packet transmission by the application to the completion of transmission.

As seen in Figure 3.4(a), with a single outstanding packet transmission (a pipeline depth of 1), throughput increases almost linearly with packet size. For a pipeline depth of 2, the throughput is always higher than before (by about 50%) but starts to saturate beyond a packet size of 2 KB. Pipeline depths of more than 2 do not provide any further increase in throughput; instead, the saturation in throughput is more severe than before. These results can be explained by considering the characteristics of the interface exported by the CIM. The interface between the NP and the CIM is in the form of a command/response FIFO. Packet transmission/reception involves a complex, non-atomic sequence of five or more commands and responses. The CIM polls the interface for commands from the NP, while, in the interrupt mode of operation, each response from the CIM generates an interrupt on the NP (the NP can also be configured to poll the interface for responses from the CIM). The handshake overhead of setting up transmission/reception therefore degrades performance, resulting in poor utilization of the link when the pipeline depth is 1. When two packet transmissions are pipelined, the commands/responses corresponding to different packets can be interleaved and overlapped, achieving higher utilization of the link. Beyond a pipeline depth of 2, though, queuing delays inside the CIM begin to dominate, eliminating any gains in throughput. For larger packets, the time to DMA the packets to the CIM



(a) Throughput

(b) Medium access latency

Figure 3.4: Performance of the Ancor CIM 250 network adapter. Note that in (a), both axes have log scales, and in (b), only the x-axis has a log scale.

begins to dominate and hence the increase in throughput diminishes, ultimately being limited completely by the CIM's DMA transfer bandwidth. Moreover, the transmission time also increases with packet size. Note that the CIM achieves a rather low utilization of the available DMA bandwidth on the VMEbus (see Table 3.1 in Section 3.1.3)[8].

Referring to Figure 3.4(b), the medium access latency using the CIM remains roughly constant for packet sizes up to 2 KB. For larger packets, the DMA overhead and the transmission time both rise, increasing the medium access latency rapidly beyond a packet size of 4 KB. The latency increases monotonically with pipeline depth since packets awaiting transmission experience higher queuing delays in the CIM.

A better understanding of the behavior of the CIM was obtained by tracing all the command/response interactions between the NP and the CIM and measuring the individual components of the medium access latency. The results (not shown here) confirmed the trends observed in Figure 3.4 but also revealed substantial unpredictability in the medium access latency. More specifically, the delay between initiating a DMA on the CIM and

[8] In all fairness to the manufacturer, we have learned that the new version of the adapter addresses some of these design weaknesses.

getting the transmission-complete interrupt becomes highly unpredictable for packet sizes larger than 4 KB and pipeline depths greater than 2. Both these effects are a direct consequence of FIFO queuing inside the CIM. Once a packet's transfer to CIM memory has been initiated, its transmission cannot be preempted or "stalled" to allow a more urgent packet to go through. If the adapter decouples packet transfer to the adapter memory from transmission into the network, then the NP can exercise fine-grain control over the order of packet transmissions. This also helps bound the medium access latency.

An unrestricted pipeline can introduce unacceptable delay jitter by introducing traffic-dependent variations in medium access latency. Since real-time communication necessitates low, predictable medium access latency, the pipeline depth and packet size on the CIM must be limited for real-time traffic as well as best-effort traffic. For example, an upper bound on pipeline depth is essential for jitter-sensitive applications like clock synchronization [31] and real-time audio/video. Since the CIM does not distinguish between best-effort and real-time traffic, the same pipeline depth and packet size must be used for both. In order to achieve the highest possible throughput while keeping the medium access latency under reasonable bounds, the pipeline depth was fixed at 2 and the packet size at 2 KB. This provides good performance for a mix of real-time and best-effort traffic.

3.3.2 Desirable Adapter Features

Several aspects of adapter design affect performance on packet reception. The received packet could be in error or it may have violated its deadline. Deadline violations could occur under high communication load in statistical real-time channels which employ resource overbooking to improve utilization. Similarly, the received packet may have to be dropped because of potential buffer overflow. The network adapter can facilitate intelligent packet handling by allowing the NP to inspect packet headers efficiently and manage on-board packet buffers. For example, limited lookahead could be used to receive the packets in order of importance (real-time over best-effort, earliest deadline first, etc.). The provision of large on-board memory on an adapter and the ability to consume packets in a non-FIFO order determined by the NP allows less-urgent inbound data to be temporarily staged on the NP while it consumes more-urgent data. Under deadline and/or buffer space violations, the NP may choose to drop the received packet. However, a policy to discard packets cannot be implemented without consuming additional host resources if the network adapter does not facilitate selective reception of packets and/or efficient reuse of its on-board memory. By forcing the NP to consume each received packet, even if it will be dropped later, the CIM design does not facilitate optimizations in which packets can be dropped by the adapter without wasting bus bandwidth and processing resources in the host. Additionally, since the NP processes packet headers while data can move directly to the destination devices, efficient examination of packet headers can improve packet reception performance. With the CIM, the only way the NP can examine packet headers without consuming the packet is by reading a sufficiently large number of bytes at the beginning of the packet and carefully handling any data bytes read. This incurs substantial overhead, both in reading the header bytes correctly and setting up the data transfer to the destination device. Note that many of these problems arise because the CIM is on the system I/O bus, and the NP does not directly control any of its operations. If the network adapter were on the NP's memory bus, it could potentially perform these functions efficiently. This has been done in the Nectar project [29] by moving protocol processing operations to the network adapter.

Accordingly, the features we consider desirable for real-time communication using I/O-bus-based adapters include:

- support for efficient network data transfer through simplified device interfaces, provision of large on-board memory to temporarily stage inbound (outbound) data, and support for transferring data via DMA to (from) the processors and other sources (sinks),
- full access to adapter memory for intelligent buffer allocation to inbound/outbound data, efficient header inspection, and selective packet discard,
- enhanced predictability through bounded medium access latency and insulation between real-time and best-effort traffic, and
- enhanced preemptibility by decoupling data transfer to the adapter from the initiation of transmission.

3.4 Overhead of Protocol Processing

The overhead involved in protocol processing on the NP significantly impacts the implementation of real-time channels and the ability to support real-time communication in general. We consider two components of this overhead: the intervening copies of data as it moves to/from the network and the execution of protocols that shepherd data between the application and the network.

3.4.1 Data-transfer Optimizations

The need to improve the delivered application-level throughput, especially in high-speed networking environments, has made transmission/reception path optimizations indispensable. These optimizations have received significant attention in recent years [39–42, 72, 95]. The primary focus of these efforts has been to eliminate unnecessary copies of data as it moves between the application's address space and the network through the OS kernel. The unreliable nature of data transfer on real-time channels obviates the need for error detection (checksumming) and recovery (retransmissions) mechanisms, making it possible to avoid unnecessary data copying. In order to optimize data transfer on real-time channels, however, it is necessary to consider other aspects besides improving the throughput delivered on each real-time channel. Since several real-time channels may be active at a given time, the data transfer during transmission should be optimized such that (a) node bus bandwidth is consumed as late as possible on the transmission path and only when absolutely necessary, and (b) node bus bandwidth is consumed by outgoing packets in an interleaved fashion, in the order of packet deadlines. Thus, it is essential to optimize data transfer, not only to minimize the incurred overhead for each real-time channel, but also to control the interference amongst different real-time channels.

Copying the AP-resident data to the NP across the HARTOS API for protocol processing and subsequent transmission results in FIFO consumption of bus bandwidth, overhead due to an extra copy, and reduced bandwidth for other processors contending for the bus. This degrades the performance of a given real-time channel and introduces interference between real-time channels. The extra copy can be avoided by moving the entire data directly to the CIM, but this cannot be done before protocol processing has been performed on the message and it has been fragmented into packets. Besides, this approach still suffers from FIFO consumption of bus and adapter/link resources. The absence of priority-based arbitration on the VMEbus necessitates alternative mechanisms to ensure that bus bandwidth is consumed in the global transmission order determined by the link scheduler. We achieve this by having the protocol stack maintain *remote references* to the data being transmitted. Data transfer to the CIM via DMA is initiated in the device driver using these remote references. Since the link scheduler determines the order in which packets are transmitted, data moves directly from the APs (or other devices) to the CIM without any intervening data copies, and in the global transmission order determined by the link scheduler. Data transfer is thus decoupled from the associated control, which occurs between the AP and NP through the HARTOS API on the one hand, and between the NP and the CIM on the other.

3.4.2 Software Overhead and Protocol Thread Scheduling

With no data movement costs incurred during protocol processing on the NP, the overhead of fragmentation by FRAG, encapsulation by HNET, and processing by the link scheduler and CIM device driver becomes important. The fragmentation (and reassembly) overhead is incurred only at the source and destination nodes while the HNET, scheduler and CIM driver overhead is incurred at all the nodes along the route. For a given fragmentation size and with no data copy, the software overhead is directly proportional to the number of fragments and is therefore higher for larger messages. With several real-time channels and best-effort traffic competing for NP's processing bandwidth, scheduling of protocol processing to consistently maintain QoS guarantees is critical.

3.4.2.1 Fragmentation and Link Scheduling Overhead

Figure 3.5 shows the latency of protocol processing and link scheduling as a function of message length without the CIM, i.e., the device driver simply drops all packets, and there is no data transmission. A test application running directly above the FRAG or HNET layer (as applicable) on one NP sends a total of over 12,000 packets down the protocol stack, under limitation of the pipeline depth. The pipeline depth is fixed at 2 and only transmission-side overhead is presented; the fragment size for fragmentation is 2 KB.

With no fragmentation, the throughput and latency are independent of message size since no data is copied within the protocol stack. In this case the message is processed as a single packet. The per-packet processing time of the HNET layer, including insertion in the scheduler queues, is about 100 μs (curve labeled `without scheduler (no frag)`). With the scheduler and CIM driver in the path, the per-packet processing time increases to 250 μs (curve labeled `with scheduler (no frag)`). Of the additional overhead of 150 μs, about 60 μs is attributed to instruction cache misses due to context switching to the scheduler.

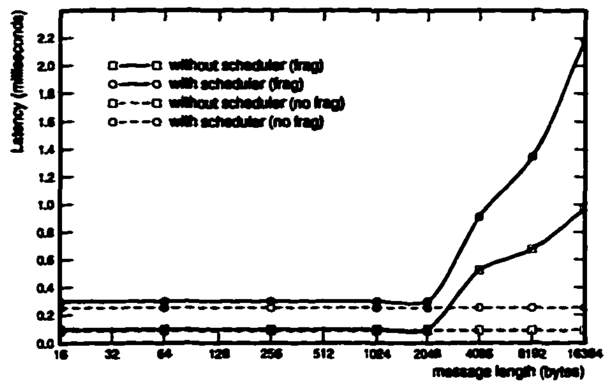


Figure 3.5: Protocol processing performance with and without fragmentation (no CIM).

The instructions comprising the test application's send loop remain in the cache when the scheduler is not invoked. The actual penalty incurred during protocol processing will be lower since the (instruction) cache will improve performance when processing multiple fragments between invocations of the scheduler. Since the scheduler always runs immediately after the currently executing protocol thread, some cache misses will surely occur if and when the thread resumes execution. The remaining difference is attributed to two context switches, one timer read, the processing of packet queues by the scheduler, and transmit processing in the CIM driver, including traversal of *x*-kernel's message structure to correctly set up commands to the CIM. Note that no transmission actually occurs on the CIM. The latency measured with the scheduler and driver roughly corresponds to the processing overhead of an outbound packet at an intermediate node, after it has been transferred to NP memory via DMA, and the corresponding protocol thread has been scheduled for execution. This information can be used in the delay computation during channel establishment.

With FRAG included in the transmission path (curve labeled `without scheduler (frag)`), the latency remains constant up to a message size of 2 KB since the fragment size is 2 KB and data is not copied within the protocol stack. The FRAG protocol provides a fast path for short (1-fragment) messages and a separate, relatively slow path for multi-fragment messages. A 4 KB message triggers fragmentation, resulting in a significant jump in latency. The extra cost of traversing the slow path ($\approx$ 400 μs) dominates the incremental cost of generating additional fragments ($\approx$ 90 μs). This explains the sudden drop and subsequent slow climb in throughput. Performance again degrades with the scheduler in the transmission path (curve labeled `with scheduler (frag)`), with a per-fragment scheduler processing overhead of $\approx 200\ \mu s$. The degradation increases with the number of fragments because of higher additional cost of fragmentation as well as an increase in the processing done by the scheduler and the CIM driver.

3.4.2.2 Scheduling Protocol Threads

With several real-time channels and best-effort traffic active simultaneously, it is critical that protocol threads be scheduled to consistently maintain QoS guarantees. The protocol-processing bandwidth must be consumed in a global order consistent with the traffic parameters of the active channels. Straightforward FIFO scheduling of protocol threads can introduce significant queuing delays, especially for large messages, as is evident from Figure 3.5. Bursts of long messages on individual channels and sudden rise in activity on multiple real-time channels only tend to exacerbate these delays. Early message arrivals due to bursts or violation of traffic specification should be prevented from consuming processing bandwidth if the generated packets would be dropped later in the link scheduler. This could be caused by insufficient packet buffer "slots," where the number of buffer slots available to a channel is determined by the maximum message size S_{max} (and the fragmentation size). The relative importance of protocol-processing overhead increases with reduction in the medium access latency. The latency to obtain the CPU for protocol processing must be bounded while utilizing the CPU as much as possible.

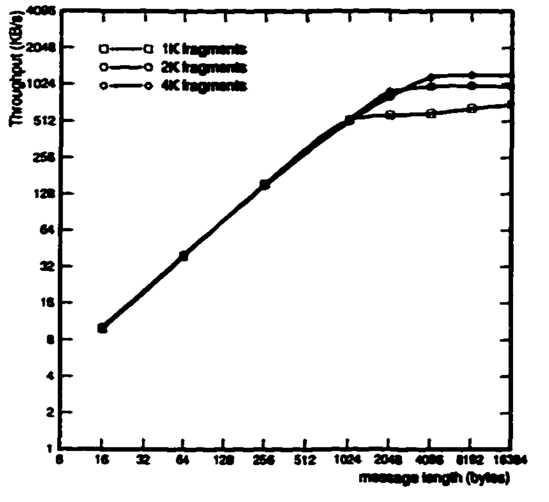
On HARTS, the CPU speed is relatively much faster than the network, and hence, the CPU is not a critical component that needs to be scheduled carefully. In our experiments, we found that it was sufficient to give a higher priority to real-time protocol processing than to best-effort protocol processing, to ensure that the RTCs got adequate CPU bandwidth even under very heavy best-effort traffic load. While this does not prevent interference between protocol processing for RTCs, on this platform, the impact of this interference was not enough to make a significant difference in real-time performance. We revisit this issue in Chapter 6 and demonstrate that much more sophisticated CPU scheduling techniques are necessary to maintain QoS guarantees in faster networks, where CPU capacity could be a bottleneck.

3.4.3 End-to-end Performance

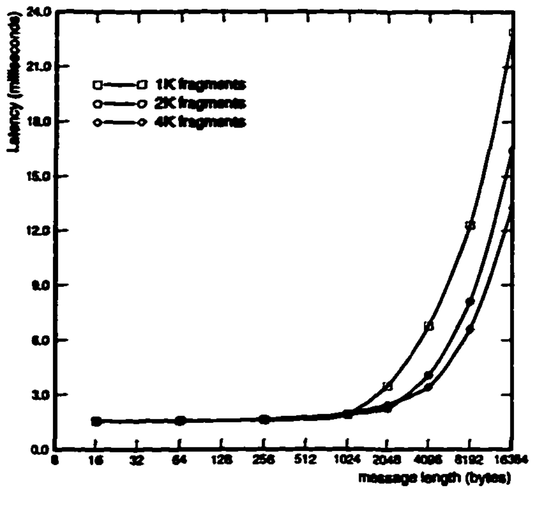
Figure 3.6 plots the end-to-end throughput and latency of message transfer between the NPs on two nodes using the CIM, with fragmentation and link scheduling. Figure 3.6 can be compared directly to Figure 3.4 to study the effect of fragmentation and link scheduling on end-to-end performance. The effect of fragment size (and hence the number of fragments/packets) is considered for different message sizes. For a given fragment size, the achieved throughput remains roughly independent of message size once fragmentation has set in. As the message size (and hence the number of fragments) increases, the latency to send the message also increases due to higher processing and transmission delays, since each fragment has to be transmitted as a separate packet. The incremental gain in throughput (or reduction in latency) reduces as the fragment size increases from 1 KB to 2 KB, and from 2 KB to 4 KB. With larger fragments, a smaller number of fragments need to be created and transmitted, reducing fragmentation and transmission overhead and increasing the throughput. However, as fragments become larger, the transmission throughput is increasingly dominated by the DMA bandwidth available to transfer the data to/from the CIM. Comparing Figures 3.4(a) and 3.6(a), for large messages (16 KB) fragmentation using 4 KB fragments reduces the achieved throughput from approximately 2.7 MB/second to 1.2 MB/second.

3.5 Effectiveness of Link Access Scheduling

In this section, we evaluate the efficacy of the link scheduler in insulating real-time traffic from best-effort traffic, and preventing ill-behaved channels (which violate their traffic specification) from affecting the delay guarantees made to well-behaved channels. The experiments evaluate the effect of traffic load (real-time and best-effort) on packet and message latencies, slot occupancy (queuing delays), and packet loss rate. The performance of best-effort traffic is measured by message latency and throughput, while that for real-time traffic is determined by whether or not all messages complete transmission by their deadlines. The deadlines and latencies of real-time traffic are measured with respect to the logical arrival times of messages. For best-effort traffic, latencies are measured with respect to the actual arrival time. The slot occupancy (or queuing delay) measures the actual time that an outgoing packet occupied a packet slot (equivalent to a buffer) in the link scheduler.



(a) Throughput

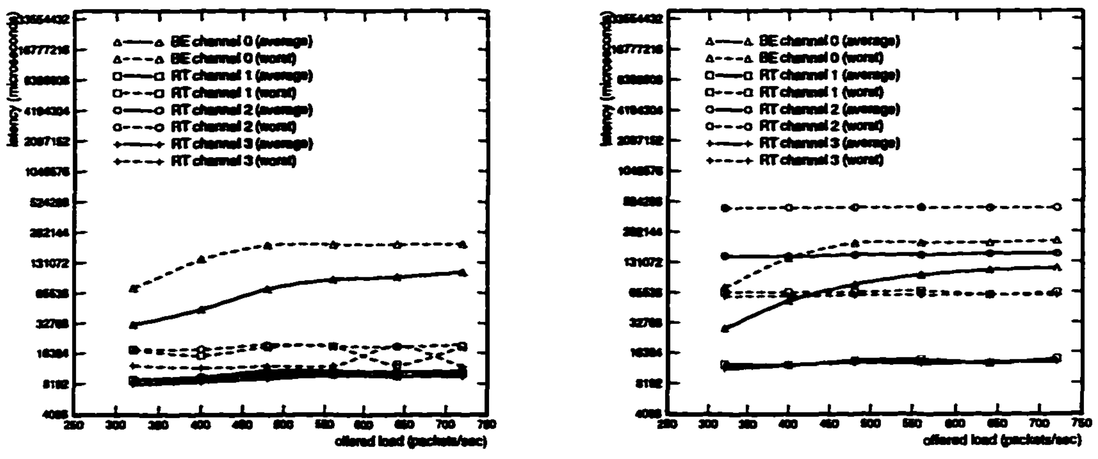


(b) Latency

Figure 3.6: Protocol processing performance with fragmentation (with CIM)

3.5.1 Outline of Experiments

The communication traffic is generated by four sources: a bursty best-effort "channel", a bursty real-time channel, and two periodic real-time channels. Tasks generating real-time and best-effort traffic execute on different APs. On the NP, protocol processing for real-time traffic is performed at a higher priority than that of best-effort traffic. This ensures that under high best-effort load conditions, real-time traffic gets sufficient protocol processing bandwidth. Besides keeping the experiments simple, this set up also helps appreciate the need for CPU scheduling mechanisms discussed in Section 3.4.2. Note that, since all real-time channels are given the same protocol processing priority, bursty or misbehaving channels are expected to interfere with other well-behaved channels. Each real-time channel generates 80 packets per second. The load generated by the best-effort source is varied from 80 to 480 packets per second (pps) in steps of 80. All the experiments were performed with a packet size of 2 KB and a pipeline depth of 2, while the message length was fixed at 8 KB, i.e., messages consist of 4 packets. The deadline for each real-time channel is set at 50 *ms* and the link horizon is set at 0 *ms*, i.e., transmission is non-work-conserving. Figure 3.6(a) shows that with a fragment size of 2 KB, the throughput for a single unconstrained source saturates at 1 MB/second, or 500 pps. Each traffic source is allowed a maximum of 50 slots (packets) in the scheduler queues at any time. Packets overflowing the scheduler queues



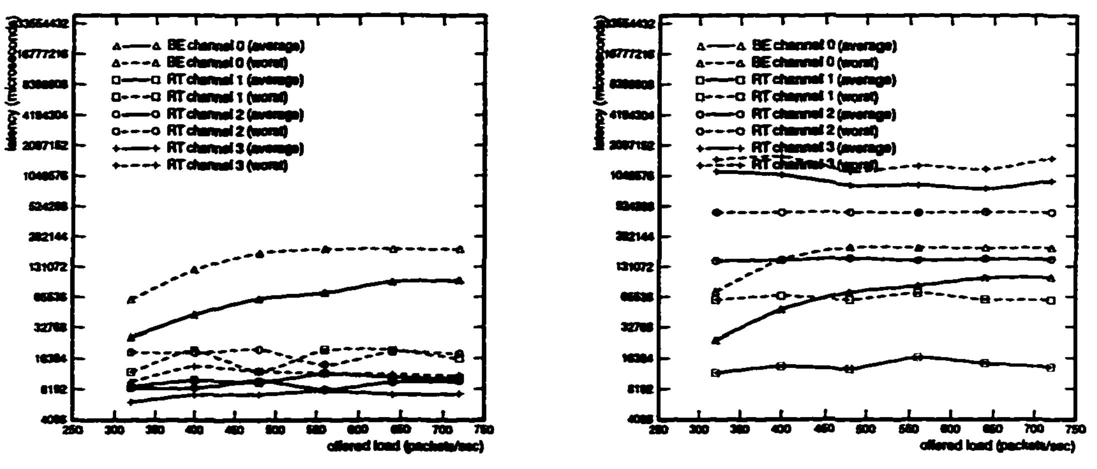
(a) Message latency (b) Slot occupancy (queuing delay)

Figure 3.7: Well-behaved real-time channels with variable best-effort load

are dropped. If a packet from any message is dropped, the remaining packets in the message are dropped as well. However, packets already inserted in the scheduler queues do get transmitted.

3.5.2 Effect of Best-effort Traffic Load on Real-time Traffic

Figure 3.7 shows the performance of well-behaved real-time channels under increasing best-effort load. Real-time channels 1 and 3 carry periodic traffic while real-time channel 2 is bursty. Each real-time channel generates the same total amount of traffic. Channel 0 is best-effort with a bursty source which increases its packet generation rate from 80 pps to 480 pps. Figure 3.7(a) shows that the periodic and bursty real-time channels have very similar average and worst-case performance that is independent of the total offered load, and all real-time messages are transmitted and no real-time packet is dropped. Best-effort throughput increases with offered load until the system capacity is reached, after which most additional messages are dropped. Latencies also increase gradually with load, until the system reaches saturation. Figure 3.7(b) shows how the behavior of bursty and periodic real-time sources differs. Messages from periodic sources typically arrive near their logical arrival times, and are eligible for transmission soon after they arrive. However, for the bursty real-time source on Channel 2, many messages arrive much earlier and are not transmitted before their logical arrival times. This ensures that real-time traffic arriving early does not



(a) Message latency (b) Slot occupancy (queuing delay)

Figure 3.8: Ill-behaved real-time channels with variable best-effort load

adversely affect best-effort performance.

The experiment in Figure 3.8 is very similar to the previous one, except that a periodic real-time channel (Channel 3) generates traffic at twice its specified rate. While Figure 3.8(a) looks almost identical to Figure 3.7(a), the excess packets on Channel 3 are dropped once the buffers available to Channel 3 are exhausted. Note that, though real-time traffic is assigned a higher priority than best-effort traffic, increasing the real-time load in this manner does not significantly affect the performance of best-effort or real-time traffic. In addition, the packets of Channel 3 that are delivered at all, are all delivered by their deadlines. A comparison of Figure 3.8(b) with Figure 3.7(b) shows that queuing delays do not increase for best-effort traffic, or the well-behaved channels. However, queuing delays shoot up for Channel 3. Results of the same experiment with a bursty misbehaving source are similar to the ones reported here.

3.5.3 Effect of Burstiness and Message Size on Delay Guarantees

As seen from the results so far, bursty sources have a greater slot occupancy time (larger queuing delay) than periodic sources. We repeated the experiments with message sizes from 4 KB–32 KB, while retaining the same total loads. Since each slot in the scheduling queue corresponds to a packet, longer packet bursts are obtained with bursts of longer messages. The probability of overflow in the scheduler queues increases with an increase in the burstiness. Even though the average traffic generation rate did not exceed the traffic specification, we observed some loss of real-time packets. The loss rate depended only on the behavior of the bursty source and the effect of increase in total system load was minimal. However, since real-time messages are processed at a higher priority, early real-time messages can use CPU bandwidth out of turn. The high medium access latency of the CIM masks out some of this effect. However, the degradation will be more pronounced with adapters providing relatively fast access to the network. The delay jitter reduces with a reduction in the burstiness of the sources, highlighting the need for the CPU scheduling mechanisms discussed in Section 3.4.2.

3.6 Conclusions

In this chapter, we explored the design tradeoffs involved in supporting real-time communication on bus-based multiprocessor hosts, which are increasingly being employed as multimedia servers and workstations. As the vehicle for this study, we implemented and evaluated real-time channels on the HARTS experimentation platform.

The main contributions are summarized as follows. A hardware and software architecture is presented that features a dedicated protocol processor, a split-architecture for the application programming interface used to access real-time communication services, and decoupling of data transfer and control in the communication protocol stack. We have highlighted the implications of network adapter characteristics for real-time communication. Since many commercial network adapters have features similar to the one we studied, these implications are applicable in general. For adapter designs ill-suited for real-time communication, techniques were presented to handle undesirable features such as unrestricted FIFO queuing in order to bound the medium access latency. To circumvent the lack of hardware support for priority-based access to resources, data transfer optimizations, CPU- and link-scheduling mechanisms to limit interference between different real-time channels were presented . These techniques together ensure that shared host resources such as bus bandwidth, protocol processing bandwidth, and link bandwidth are consumed in a global order determined by the traffic characteristics of the active channels. Finally, the performance and effectiveness of our real-time channel implementation through several experiments under varying traffic characteristics was demonstrated.

It should be noted that the CIM's throughput is much less than the capacity of the network link or the bus bandwidth. This is partly due to its poor integration with the host operating system software. For example, the overhead due to its complex command/response interface has a significant impact on throughput. In Chapter 5, we revisit the design of the CIM, and describe how network adapter emulation may be used as a design and analysis tool to improve the design, and hence performance, of the CIM.

Relatively simple CPU scheduling was adequate to maintain QoS guarantees using the CIM, since the network is very slow, and the CPU is not a bottleneck. In Chapter 6, we show that for faster networks, more sophisticated scheduling is needed to provide QoS guarantees, and examine various kinds of QoS support on the host and network adapter, and study their impact in various (faster) network configurations.

CHAPTER 4

THE *END*: A NETWORK ADAPTER DESIGN TOOL

In Chapter 3, we studied an Ancor CIM 250 based network, and demonstrated how real-time channels were implemented on that platform. Our experiments revealed several performance bottlenecks due to poor architecture and interface design. In this chapter, we describe *END*, a network adapter design tool, that may be used to design and analyze network adapters before they are built, thus helping avoid costly design errors.

4.1 Introduction

The experiments in Chapter 3 demonstrated that 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 and network device driver, 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 overhead 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 Section 2.3.1, we proposed *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. We now present the *E*mulated *N*etwork *D*evice (*END*), a network adapter design tool that interfaces to a real communication protocol stack on a host via the system I/O bus. Since most network interfaces are on the system's I/O bus instead of the private memory bus [83], this configuration allows *END* to generate the same overhead 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 experiments can be performed directly on the target host platform, with *END* running concurrently on its own processor, thus accounting for overhead and host architectural features that influence communication performance. *END* thus permits *hardware/software codesign*, where integration of hardware and software, and 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. This is not necessarily a problem because desktops equipped with at least two processors are now readily available. More importantly, the overhead 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.

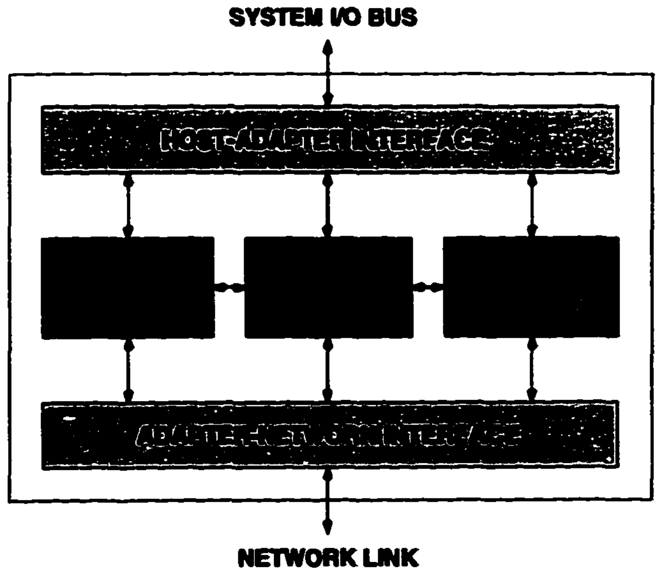


Figure 4.1: Generic network adapter architecture

Overall, the ability to accurately capture both the network and the host behavior, and run experiments in real time, outweigh the disadvantages.

The rest of the chapter is organized as follows. Section 4.2 describes the architecture of a generic network adapter, and highlights some issues in network adapter design. The architectural framework and our implementation of *END* is presented in Section 4.3. Section 4.4 considers how the implementation platform may impose restrictions on the emulator, and how some of these constraints may be overcome. Section 4.5 briefly describes the software structure of *END*, and if and how each software module may be ported to other platforms or modified to model different adapters. Section 4.6 discusses related work, and Section 4.7 concludes this chapter.

4.2 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. In particular, our goal is to determine how the various design options can affect QoS. The architectural framework of *END* is based on these architectural components.

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 dissertation. Within the end hosts, communication performance is determined largely by the capacity of the software and hardware components to move data between applications and the network. 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, scheduling and synchronization 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.

The focus of this chapter is on the design of the network adapter and how it interacts with the host communication software through a device driver. Figure 4.1 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 [83]. 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 may employ one or more general-purpose microprocessors (e.g., one for transmission and one for reception) and some custom hardware under control of the adapter firmware.

Provision of QoS guarantees may add significant complexity and overhead in the management of incoming and outgoing time-sensitive data. The design enhancements required depend to a great extent on the particular performance parameters (such as delay, bandwidth or loss) being targeted for provision of QoS. For example, packet transmission/reception delay is significantly affected by the efficiency of the host-adapter interface, including the cost of data-transfer and host-adapter handshake. Similarly, the raw data-transmission bandwidth available is affected significantly by the efficiency of the available data-transfer mechanism and the degree to which operations may execute concurrently on the adapter. The QoS delivered to individual connections is dictated by the nature and extent to which adapter resources are shared. Using shared buffers across all connections may, for example, result in higher packet loss in the presence of a greedy connection, i.e., one generating excess traffic. In the discussion below, we describe each of the adapter components mentioned earlier, with special emphasis on QoS support.

4.2.1 Host–Adapter Interface

The host-adapter interface exports various operations to the host, including initiation of packet transmission and reception, examining the status of pending transmissions, etc. It is typically implemented by a device driver running on the host which communicates with 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. Interrupts allow immediate synchronization, at the cost of increased overhead in handling an interrupt and its 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 overhead, the interface must be as simple as possible. At the same time, more complicated interfaces may be required for intelligent adapters with built-in programmability for enhanced flexibility. Depending on the degree of QoS support provided, the host-adapter interface may include commands that carry information regarding the type (e.g., class) of traffic or connection a packet belongs to, and the QoS level it requires for transmission. In the absence of such information, the adapter may be incapable of providing QoS-based service discrimination. In addition, multiple command/response mailboxes or queues may be required to minimize interference between interface operations of different classes of traffic. QoS support for reception may also need to perform host-controlled buffer management and priority-based packet input, and to limit the rate at which the adapter interrupts the host.

4.2.2 Adapter Internals

Referring to Figure 4.1, three modules provide the internal functionality of an 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, the packet is transferred between host memory and adapter buffers via DMA or programmed I/O (PIO). The choice between DMA and PIO is a function of such factors as the bandwidth of the system I/O bus and the size of the data transfer. With DMA, the 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. PIO, on the other hand, has no startup latency but requires CPU cycles for the entire duration of transfer, potentially limiting concurrency. A hybrid approach could employ PIO for small packets and DMA for larger ones.

The tradeoff between PIO and DMA applies just as well to time-sensitive traffic as to traditional best-effort traffic. However, with PIO, the host alone determines the ordering of packet data transfers to/from the adapter. With DMA, on the other hand, the adapter may have to perform data transfers to/from host memory according to the relative priorities of the individual packets. The nature of adapter support required is also determined by the capabilities of the system I/O bus such as priority-based arbitration and preemptable block transfer DMA.

Tx/Rx Queuing and Service Discipline: 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 queuing policy and mechanisms employed, and the service discipline employed to serve the packet queue(s), depend on the expected traffic mix. For example, for best-effort traffic it may suffice to provide simple first-in-first-out (FIFO) queuing of packets, with deep pipelining of operations on the adapter and a simple service discipline that selects and transmits packets in FIFO order. This delivers high throughput by keeping the overhead of queuing and packet selection low, and by exploiting the overlap between different operations, for instance, the DMA transfer of one packet and network transmission of another. More sophisticated queuing and scheduling algorithms may be required for real-time traffic in order to provide per-connection QoS guarantees.

Pipelining allows the host and adapter to operate independently, rather than in lock step. While this increases data-transfer throughput, it reduces host control over the order of packet transmissions on the adapter. This in turn makes it difficult to provide QoS guarantees to packets on a per-connection or per-flow basis, unless the adapter is designed to be QoS-sensitive, i.e., cognizant of QoS guarantees. Note that when the pipeline depth (i.e., the number of packets queued on the adapter) is 1, the host exercises complete control over packet transmission order [75].

The queuing and packet selection policy used by the adapter is relevant whenever the host generates data at a rate faster than the adapter can transmit. This could be either because of a relatively faster host CPU, or an *available* link bandwidth that makes the network link effectively slower than the host. At one extreme, the adapter could maintain a *single FIFO queue* for all outgoing packets sent by the host. Alternately, the adapter can provide superior service to real-time traffic by using *two FIFO queues*, one for real-time traffic and one for best-effort, and serving the real-time queue at a higher priority. However, even if the host sends packets to the adapter in a QoS-sensitive order, subsequent arrival of high-priority data may require transmission to be reordered on the adapter. The adapter can exercise fine-grain control over packet transmission order by using a (dynamically sorted) *priority queue* for real-time traffic, reaping the benefits of pipelining (i.e., higher throughput) without performance degradation or QoS violations. This, of course, requires that the adapter be made cognizant of the QoS requirements of individual packets. Since the adapter handles both incoming and outgoing data which share the same I/O bus for transfers to/from the host, it is necessary to correctly order these operations as well.

Buffer Management: Buffer management plays an important role in packet transmission as well as reception. In general, the adapter may need to provide buffers as a staging area for incoming and outgoing best-effort and real-time traffic; the buffers 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 (which may result in packet loss) correctly. Sharing buffers between connections, even with per-connection queues, simplifies buffer management at the risk of unpredictable packet loss under high traffic loads. The adapter may also exercise partial control over packet buffers on the host. For example, if buffer space on the adapter is low, it could potentially delay transfers between the host and adapter until just before packet transmission. Expensive copying overhead can be avoided if the adapter manages buffers such that packet headers are stored separately from packet data.

4.2.3 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 must 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 is issued after copying the packet to the adapter. Similarly, on packet reception from the network, the adapter must stage this data into on-board buffers and subsequently transfer the data to the host.

For transmission, the adapter-network interface can implement two distinct network models: a *point-to-point* network model and a *shared* network model. The point-to-point network model is characterized by a *continuously-available* network, such that the adapter can transmit on the link as soon as outgoing data is made available by the host. That is, there is no waiting time associated with accessing the attached network. Examples of point-to-point networks include various direct-connected topologies and switch-based networks such as Fiber Channel, ATM, and Switched Ethernet. The shared network model, on the other hand, is characterized by an *intermittently-available* network, implying that the adapter can transmit on the link only at certain epochs and for a limited duration. Various token-based networks, such as IEEE 802.5 and FDDI, with specified deterministic token rotation and holding times, fall into this category. Ethernet also falls into this category, except that the availability of the network is probabilistic in nature, as determined by the likelihood of collisions on the network.

4.3 *END* Emulation Architecture

In this section, we present the emulation framework for *END* in the light of the architectural components described in Section 4.2. We first define the functional interface that *END* must export to the host, followed by a description of the components constituting *END*. We then describe our implementation of *END* and how it operates in response to host commands, with an emphasis on network transmission. Finally, we discuss the *END* design and implementation issues related to modeling end-to-end communication, or stand-alone network reception.

4.3.1 Host View of the 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. 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.

In contrast, for data reception, *END* can generate incoming traffic using either stochastic models or traces of real network traffic. In this case, *END* is a source of network traffic, and the host is a sink. Alternatively, 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 by connecting two or more *END* nodes using the system I/O bus (see Section 4.3.5). Further

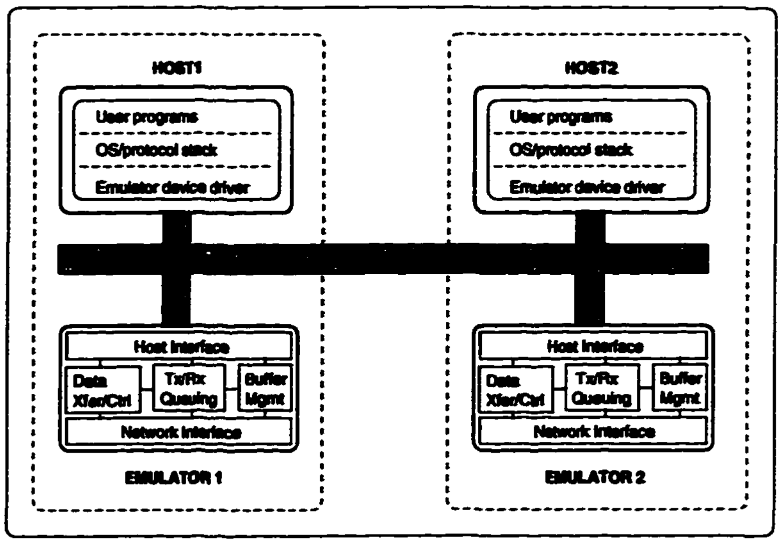


Figure 4.2: *END*-based device emulation architecture.

details regarding reception emulation and issues in data reception will be presented in Chapter 7.

4.3.2 Emulator Components

Figure 4.2 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 4.3.1, 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 helps ensure 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

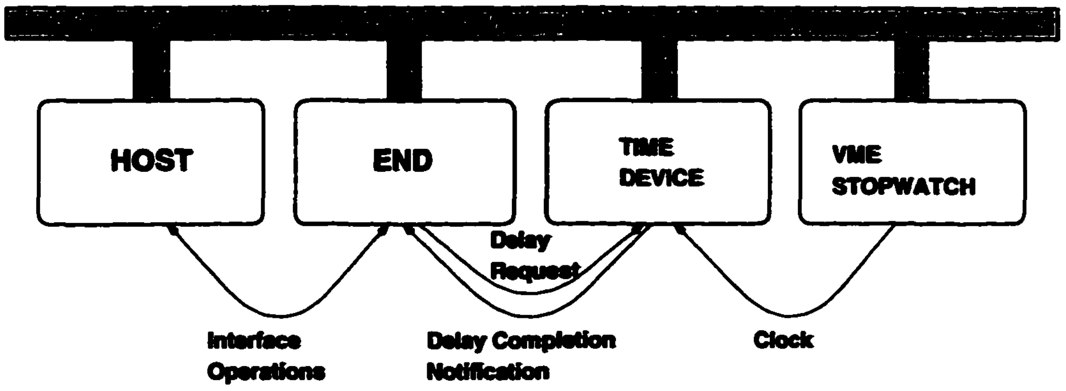


Figure 4.3: Emulator system configuration.

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 data is not actually 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, end-to-end communication*. For the purposes of studying adapter performance, the actual data content may be 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 delay expires, either with an interrupt or simply setting a completion flag that the adapter can poll. If an event occurs on more than one node, all nodes involved need to be notified simultaneously. An example is the completion of a transmission. This event is initiated by the transmitting node, but the reception node must also be notified that it has received a packet.

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
	programmed 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 4.1: Configurable components of *END*.

4.3.3 Implementation

Figure 4.3 depicts the configuration of our implementation with one emulator node (for each additional node, another host-*END* pair is required; the time device is shared by all the nodes). The processor boards of each emulated node communicate with each other via the VMEbus, as per the interactions shown in Figure 4.3. Emulation of transmission as well as reception has been implemented in *END*. The internal structure of *END* comprises an *emulation core* and several other components, summarized in Table 4.1. The emulation core is a minimal executive with support for threads, interrupt handling, and semaphores. It is essentially a cyclic executive that polls the host for commands, executes them, and notifies the host of completion of the commands. It has several sub-components that may be configured and parameterized to realize any desired adapter behavior. The components surrounding the emulation core are briefly described 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 either via DMA or PIO. In our prototype, there is no actual data transferred for pure transmission, although there is real data transfer of 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. Two queuing policies have been implemented, FIFO and Earliest Due Date (EDD), 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. Packet buffers on the host and/or adapter may also be reserved on a per-connection basis [75].

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 command/response FIFO. Each delay request has a unique identifier, a completion time, resources used, a priority level, and a flag indicating whether the emulator needs an interrupt in addition to the completion notification in its response FIFO. If two timed activities contend for a shared resource, the lower priority activity waits for the completion of the higher priority activity. If the higher priority event has preemptive priority, the lower priority event is preempted, and its 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 [53][1]. 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 time is measured correctly.

END models are implemented as multi-stage devices using the above components. Each stage corresponds to an activity like data movement (e.g. copying, transmission) or trans-

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

formation (e.g., encoding, segmentation), and requires certain resources which are typically shared (e.g., CPU, buffers, buses, network links). Messages on the device pass through one or more stages, with a corresponding 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 can be processed at that stage concurrently.

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.

4.3.4 Emulator Operation

Operation of the emulator is illustrated with the example of the transmission request procedure shown in Figure 4.4 (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, buses, buffers and data structures. It also captures contention

```
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 4.4: Transmission emulation for n-stage adapter operation.

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.

4.3.5 Two-way Communication Across the "Network"

For two-way communication, we use the system I/O bus as the "network", with data being transferred from one adapter node to another (see Figure 4.2). Since bus contention increases with the number of emulated nodes, the I/O bus bandwidth limits both the speed of the network being emulated, as well as the number of nodes in the experimental configuration. However, this approach is still 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 entire 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 wide range of network speeds. There is at least one other study that has evaluated protocol performance using the I/O bus of a multiprocessor host as a high-speed network [45].

In order to support such two-way communication, the *END* model of the network adapter was extended to support the commands for data reception, as well as for data transmission. Further, the time device provides time services for more than one *END* node, and synchronizes their activities when necessary. For example, when modeling data transmission from one node to another, the time device simultaneously sends a transmit completion notification to the source node, and a receive completion notification to the destination node. Further, if, as in the case of the CIM, network transmission has preemptive priority over DMA, it also ensures that DMA operations, if any, are preempted at *both* the source and destination nodes during transmission. It also ensures that the entire data (or simply the headers) is copied to the destination *END* node before sending a reception completion notification. All this is accomplished in *END* by associating resource utilization and priority tags with each delay request to determine which nodes are involved in the operation, which operations can occur simultaneously, and which need to be preempted.

4.3.6 Reception Modeling

It is often desirable to perform experiments where there is no traffic source node, but a destination node receives some kind of synthetic traffic. This could be either because we wish to focus purely on reception issues, or simply because the experimental platform does not have enough nodes to support end-to-end communication (in our implementation, a minimal end-to-end configuration has five CPU cards – the source and destination hosts and "adapters" (*END*), and the time device (Figures 4.2, 4.3)). In this case, *END* generates packets that are indistinguishable from those originating from a real source node. The data and headers for such packets may be generated either by replicating the source node's application and protocol stack on *END* and composing packets on the fly, or using stored packets. Depending on the applications and protocol stacks, it may or may not be necessary to have real data, but correct headers are required to traverse the destination host's protocol stack. Packet arrivals are triggered by "reception complete" notifications from the time device. The time device can be programmed to generate these arrivals based on any stochastic function, real traces, etc.

4.4 Platform Considerations

In principle, *END* may be used to model arbitrary network and adapter configurations by specifying the appropriate paradigms and performance parameters. However, implementing such a model effectively necessitates that the implementation platform (more specifically, the device emulator) have sufficient CPU capacity to model operations in an accurate and timely manner. For instance, we cannot specify emulator delays smaller than the shortest delay provided by the time services, or support data throughput at rates greater than what the host can generate. In this section, we evaluate our platform and assess its capabilities. We then describe how *END* may overcome some of the constraints placed on it by the limitations of the platform.

4.4.1 System Capacity Analysis

The host on our experimental platform uses an Ironics IV3207 card, with a 25 MHz Motorola MC68040 processor (SpecInt92 rating of 12.3). A detailed performance analysis of this platform has been presented in [74]. To summarize, using the protocol stack in Figure 3.2, this CPU can process about 2000–2750 best-effort packets per second, with the lower number corresponding to single-packet messages, and the higher number corresponding to 15-packet messages (Figure 1(a) in [74]). Since the maximum packet size is 4 KB, this corresponds to throughputs of 8000–11000 KB/second. Protocol processing costs and scheduling overhead are higher for real-time data, and with 15-packet messages, the host is able to process 1300 real-time and 675 best-effort packets each second (see Table 6.2 and Figure 6.2(a)). Hence, we estimate that the mean CPU processing costs per packet (including scheduling overhead) for best-effort and real-time traffic are about 364 μs and 580 μs, respectively.

The emulator and time device also use CPU cards identical to the host. When the emulator "transmits" a packet, it must dequeue it (from a FIFO queue, or a priority heap sorted by the packet deadline), issue a delay request to the time device, and handle the time device interrupt at the end of the delay. In addition to the actual transmission delay, these operations take about 66 μs and 122.5 μs for FIFOs and heaps, respectively. Assuming that the requested delay is long enough, we can fetch the next packet for transmission while the previous packet is being transmitted, thereby masking most of the cost of the dequeue operations. In this case, the overhead is about 48 μs and 53 μs, masking a delay of about 18 μs and 70 μs for FIFOs and heaps, respectively. The remaining overhead is the cost of issuing the delay request to the time device, and handling its response. This implies that the emulator has the CPU capacity to transmit approximately 15000 best-effort or 8000 real-time packets per second. In practice, we can achieve less than half of this, since the emulator must also use its processing capacity for other activities such as host-adapter interface operations, enqueuing/dequeuing packets for DMA, modeling delays for DMA, etc. In the next section, we see how *END* overcomes some of these limitations.

4.4.2 Overcoming the Limitations of the Platform

One of the key advantages of emulation is that it enables accurate experiments with real system components and overhead. However, it is also a disadvantage in that such modeling is restricted by the actual performance of the emulation platform, unlike simulations where arbitrary times may be assigned for various system activities. There are various techniques to get around some of these restrictions, in particular, to model networks that are faster than the platform's communication medium, and to model hosts that are faster than the platform's CPU. While these techniques (described below) permit us to model a wider range of systems, they do distort the results since they cannot selectively scale the relative speeds of different system components by different factors. For example, the technique for modeling faster CPUs automatically scales the speed of the CPU processing and of memory access by the same factor, which, in general, will not be the case. In spite of these limitations, these are useful techniques, but the results should be interpreted with caution.

Modeling fast networks: Hosts in an emulation environment may communicate with each other either via a real network (say, an Ethernet), or even pretend that the system bus is the the network (see Section 4.3.5). The speed of the communication medium and the contention for access to this medium limits the network speeds we can emulate. However, if the actual data transmitted is not important (which is true for many performance evaluation experiments), it suffices to transfer only the packet headers, and use arbitrary data at the destination node. If the headers are small compared to the data, this not only reduces contention, but allows us to model much faster networks. Let $T_h(h)$ and $T_m(m)$ be the times required to transfer a header of length h on the given communicating medium, and a message of length m bytes on the target network, respectively. As long as $T_h(h) \leq T_m(m)$, we can model such a network by inserting a delay of $T_m(m) - T_h(h)$ while emulating the data transfer operation (since a delay of $T_h(h)$ will really be experienced while transferring the header, we only need to account for the remaining amount of time).

Modeling fast CPUs: In addition to fast networks, we often need to model systems with significantly higher CPU capacities. One solution is to port *END* to a faster platform but this may not always be practical for reasons of cost and time. A simple solution arises from the observation that it is not the CPU's processing bandwidth alone that matters, but how it compares with network bandwidth. If we wish to model a network with a given bandwidth, and have a given workload, by slowing down both the network and the rate at which data is generated by the same factor, the network utilization remains unchanged. However, since the CPU itself is unchanged, it executes at the same speed in both cases, making the CPU relatively faster than the network in the latter case. Thus, to give the illusion of a faster CPU, it suffices to *slow down* the network. If the *CPU speedup factor* is C_{sf}, we need to slow down all system delays by the same factor. This implies increasing time intervals such as message interarrival times, real-time message deadlines, and synthetic delays[2] by a factor of C_{sf}. Consider modeling an operation that takes c units of CPU time on our host, and has a synthetic delay of d, for a total delay of $c + d$. If we increase the delay by a factor of C_{sf}, the processing cost of the operation remains unchanged at c, and the total delay is now $c + dC_{sf}$, making the CPU overhead relatively smaller by a factor of C_{sf}. Considering the effective system capacity utilization of this operation, we have:

$$\text{Utilization} = \frac{d}{c+d} \tag{4.1}$$

After slowing down all delays by C_{sf}, we have:

$$\begin{aligned}\text{Utilization} &= \frac{dC_{sf}}{c+dC_{sf}} \\ &= \frac{d}{\frac{c}{C_{sf}}+d}\end{aligned} \tag{4.2}$$

It follows that the experiments now take C_{sf} times as long to run, but we have effectively built an environment where the CPU appears to be much faster. This feature must be implemented carefully, since delays may be specified in various places in the operating system and emulator. In our implementation, we have been able to localize these changes so that applications need not be modified in any way. All programs used to analyze data from the experiments also use C_{sf} to normalize the results.

4.5 *END* Software Implementation

The implementation of *END* follows the structure in Figures 4.1 and 4.3. *END* is written almost entirely in C (about 3000 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 readily available on almost any platform. 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. Further, the interrupt

[2]Synthetic delays are the delays used to model host and network activities that do not really take place on *END*. For example, since data packets are not really transmitted, a synthetic delay is inserted into the *END* model to account for the time taken by this operation.

Module	Modifications required due to change of:		Approximate size for CIM model (lines of C code)
	Platform	Target adapter	
Host-adapter interface: device driver (on host)	minimal	extensive	2000
Host-adapter interface: host driver (on *END*)	minimal	extensive	1500
Adapter-network interface	minimal	minimal	300
Libraries	minimal to none	moderate	variable, depends on feature set
Time device	minimal	minimal to none	1500

Table 4.2: Portability of *END*.

wrappers and interrupt generation code is localized, and does not need to be changed in multiple locations. We now describe the various software modules, keeping in mind platform dependencies and portability. Software modules may need to be modified either because of changes in the platform, or changes in the adapter being modeled. A summary of the nature and extent of changes required in the various modules is presented in Table 4.2. It is clear from the table that while *END* can easily be ported to another platform, substantial parts of the *END* model must be changed when we wish to model a different adapter.

Host-adapter interface: This is completely determined by the adapter being modeled and would need to be rewritten for each adapter. It comprises the device driver on the host, and the host driver on *END*. The *END* model for a particular adapter can be ported from one platform to another with little or no modification. Device drivers (and their corresponding host drivers) are typically structured as actions triggered by particular commands. These commands typically indicate initiation or completion of data movement across the I/O bus, or completion of transmission/reception. It is possible that even dissimilar devices could have similarly structured device drivers, perhaps permitting some code reuse.

Adapter-network interface: The adapter-network interface allows for polling and/or interrupts as the synchronization mechanism. It has a clean interface with the adapter model. It is invoked by the adapter when a packet is ready for transmission, and it may call adapter functions when packet transmission or reception completes. Other than these entry points, no changes should be needed due to a change of platform or adapter model.

Libraries: Buffer management and queuing mechanisms are completely platform independent and have been implemented as separate libraries. At present, adapter buffers are pre-allocated, fixed-size, shared buffer pools. Queuing support has been implemented for FIFO and deadline-based priority heaps. Other buffer management policies, queuing mechanisms, or other services can readily be added to these libraries. Data-transfer control provides for modeling PIO and DMA. This is supported by the Time Device, which allows for synchronous and asynchronous delays, which are necessary to model PIO and DMA, respectively. Libraries would need to be extended to model new adapter features (e.g., SAR, protocol translation). However, these libraries would be usable by other adapters that have similar features.

Time Device: For time services, a high resolution clock is necessary; while we have used the VME StopWatch, such clocks are now available on many modern microprocessors. The time device is a separate module that uses no operating system support. It may be ported to another platform by changing the clock source and its interrupt management mechanism. It supports multiple *END* nodes, synchronizes with them using polling and/or interrupts, and does not change at all from one adapter model to another.

4.6 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 [33,41,83,95], and communication subsystems in general [38,89]. 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 overhead [15,102]. Other efforts have focused on protocol-level simulation with the ability to run the actual protocol stack during simulation [20], and network-level simulation with a focus on routing and end-to-end protocol performance [64,73]. 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 could 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 [17]. 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 the adapter design and its interactions with the protocol stack.

I/O device modeling: The issues involved in modeling disks have been studied by other researchers [103]. 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.

4.7 Conclusions

In this chapter, 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 described how *END* may be used in various configurations – with true end-to-end communication between two nodes, or with only a source node sending packets to a sink, or only a destination node with packets arriving from a synthetic source. We also discussed issues related to how emulation is affected by limitations of the platform, and how these may be overcome.

In the next three chapters, we demonstrate the versatility of *END* by using it to study various problems in communication subsystem design. Chapter 5 uses *END* to construct a representative model of a real network adapter, and to show how the adapter may be redesigned to improve its performance. Chapter 6 describes how *END* is used to model different network configurations and study QoS issues in the network adapter and the host operating system and Chapter 7 uses *END* to study various host and adapter based solutions to the receive livelock problem.

CHAPTER 5

ADAPTER DESIGN USING *END*: A CASE STUDY

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 this chapter, we use *END* to study adapter design by conducting a case study for an existing device, the Ancor VME CIM 250 [3] (CIM) adapter, described in Chapter 3. 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 for this device.

The rest of this chapter is organized as follows. Section 5.1 presents the case study of the CIM network adapter and its emulation using *END*. Section 5.2 studies the performance of the CIM to identify performance bottlenecks and uses *END* to evaluate design improvements for data transmission. Based on this experience, the limitations of emulation and possible improvements are discussed in Section 5.3. Section 5.4 concludes this chapter.

5.1 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*. Our experiments with the CIM in Chapter 3 revealed several performance bottlenecks and potential for improvement, making the CIM a suitable candidate for this study. The performance characteristics of the CIM have also been studied in detail in [70]; some of those results have been used 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 the *END* model. 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. Since we focus on data transmission, the experiments described in the rest of this chapter use *END* in a "transmit-only" configuration, i.e., data transmission is modeled as a delay, and the transmitted data is not delivered to any destination node. However, we did perform the same experiments in an end-to-end configuration using two-way communication described in Section 4.3.5, and obtained nearly identical results. We now describe the CIM, its data transmit and receive paths, and how we emulate its behavior using *END*.

5.1.1 Ancor CIM 250

The Ancor VME CIM 250 [3] Communications Interface Module is a network adapter for ANSI Fiber Channel 3.0 [2] networks. Besides communication interface hardware (an IBM OLC266 Fiber Optic link card), the CIM has an NEC 32 MHz 70236 I/O processor, 8 MB 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 device driver, writes responses to the host via the response FIFO, and then issues an interrupt to notify the host of the response. Note, in the descriptions below, that each packet transmission or reception involves at least five commands – three from the adapter to the host (possibly with an interrupt with each command) and two from the host to the adapter.

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 into local memory
			DMA data for list into 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 5.1: Host–adapter interaction for data transmission.

Transmission: A simplified sequence of the events involved in CIM transmission is shown in Figure 5.1. 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` ack** to the host on completion. The CIM interrupts the host each time it sends a response. 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 in parallel. We call the number of incomplete **`write`** operations the *pipeline depth* on the CIM.

Reception: A simplified sequence of the events involved in CIM reception is shown in Figure 5.2. When a message arrives at the adapter, it notifies the host with a **`read`** command. The host initializes its data structures and allocates buffers to receive the packet and responds with a **`read` ack** command that completes the handshake by giving the adapter the host's transaction identifier for this message. The host then sends an **`address`** command with the packets destination address (or a scatter-gather address list of the buffers in an **`address list`** command for packets longer than the kernel page size). When the data has been copied to the host, the adapter notifies it with an **`address` ack** command, and the host sends more **`address`** commands, if needed. After the last of the data has been transferred to the host memory, the adapter sends a **`read` end** command.

Phase	Network Adapter		Device Driver
Initialization	store packet in buffer		
	initialize data structures		
	send `read`	⟶	
	interrupt host	⟶	initialize data structures
		⟵	complete host handshake
		⟵	send `read ack`
DMA		⟵	send `address`
	DMA packet to host	⟶	
	send `address ack`	⟶	
	interrupt host	⟶	send `address` command, if needed
Completion	send `read end`	⟶	
	interrupt host	⟶	demux packet to protocol stack

Figure 5.2: Host–adapter interaction for data reception.

5.1.2 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 5.3 shows an outline of the *END* model for the transmit operations of the CIM (Figure 5.1) using the structure presented in Figure 4.4. Functions `stage`$_i$`_do()` of *END* are triggered by events such as a command from the host or the completion of the previous stage. `stage`$_i$`_done()` represents the action taken after the completion of the time delay (if any) representing the action of that stage. Similarly, Figure 5.4 shows an outline of the *END* model for the receive operations of the CIM shown in Figure 5.2. A key difference is that a `write` sequence is triggered by the host, while a `read` sequence is triggered by a synthetic packet arrival (in the case of a "receive-only" configuration of *END*), or a transmission from another *END* node (in the case of true two-way communication). These functions are adequate to completely define the host–*END* interface, i.e., this model interacts correctly

Stage$_i$	Trigger	Stage$_i$_do	Stage$_i$_done
Initialization	`write`	Init data structures send `write ack` interrupt host	

DMA	`address list`	read address list
		enqueue segments for DMA
		`StartDmaXfer(){`
		dequeue segment;
		async delay for DMA}
			if (last segment copied)
			enqueue message for xmit
			`Xmit_do()`
			send `address ack`
			interrupt host
			`StartDmaXfer()`

Xmit (Fiber Channel)	`DMA_done`	dequeue message async delay for transmit	send `transmit ack` `Xmit_do()`

Figure 5.3: Outline of *END* model for transmission on the CIM.

Stage$_i$	Trigger	Stage$_i$_do	Stage$_i$_done
Initialization	message arrival from network	Initialize data structures send `read` interrupt host	
	`read ack`	complete host handshake	

DMA	`address`	enqueue packet for DMA
		`StartDmaXfer(){`
		dequeue segment;
		async delay for DMA}
			send `address ack`
			interrupt host
			`StartDmaXfer()`

Completion	last buffer transferred	send `read end` interrupt host	

Figure 5.4: Outline of *END* model for reception on the CIM.

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

Table 5.1: Important CIM parameters

with the host device driver. To complete the model, it is necessary to ensure that it executes with the same delays as the CIM.

5.1.3 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 [70], our own measurements [56], and conversations with staff at Ancor Communications, Inc. Lin *et al.* [70] performed extensive experiments on the CIM (using VMEbus 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 [70] were made on a completely different platform with a different CPU, memory, protocol stack and operating system. Accordingly, we used 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 below.

5.1.3.1 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 [70], $\mathcal{T}_{DMA}$ must be computed via measurements since it is platform-dependent. $\mathcal{T}_{FC}$ is given by

$$\mathcal{T}_{FC}(\mathcal{S}) = \mathcal{F}_{overhead} + \lceil \frac{\mathcal{S}}{\mathcal{F}_{framesize}} \rceil \times \mathcal{F}_{framedelay}, \tag{5.1}$$

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 5.1, $\mathcal{T}_{FC}$ can be computed for a given message of size $\mathcal{S}$.

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:

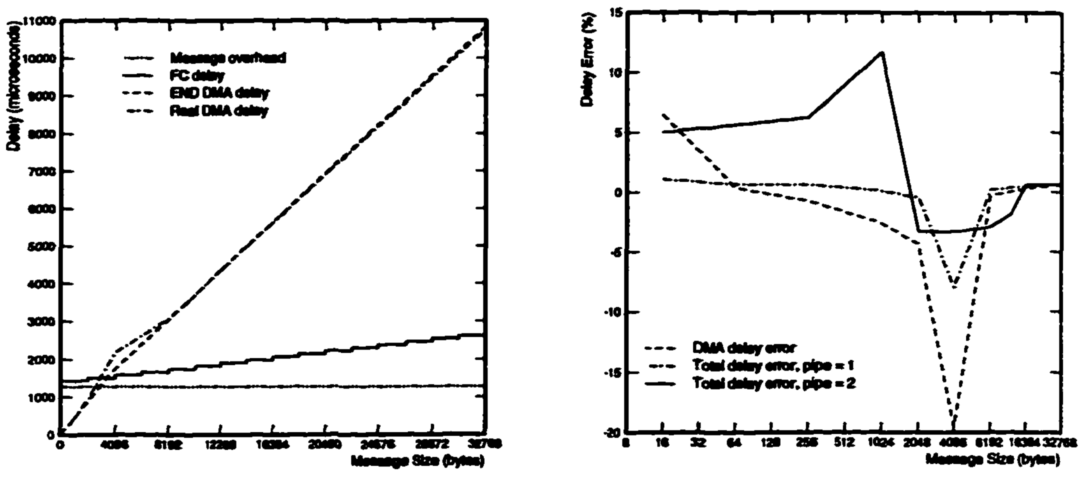
$$\mathcal{T}_{m(pipe=1)}(\mathcal{S}) = \mathcal{M}_{overhead} + \mathcal{T}_{DMA}(\mathcal{S}) + \mathcal{T}_{FC}(\mathcal{S}), \tag{5.2}$$

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 12000 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 5.1 lists the measured value for $\mathcal{M}_{overhead}$ as 1280 μs. 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 μs, which yields an $\mathcal{M}_{overhead}$ value of approximately 1220 μs after accounting for the DMA delay.

$\mathcal{T}_{DMA}$ can now be computed as the only unknown in Eq. 5.2. 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 5.1), and lower for larger messages (see Figure 5.5(a), also expanded in Figure 5.7(a)). As an approximation, we use a delay function that is the combination of two linear functions (Figure 5.5(a)):

$$\mathcal{T}_{DMA}(\mathcal{S}) = \begin{cases} \lceil \frac{\mathcal{S}}{\mathcal{D}_{segsize}} \rceil \times \mathcal{D}_{overhead} + \mathcal{S} \times \mathcal{D}^{small}_{byte} & \text{if } \mathcal{S} < \mathcal{S}_{small} \\ \lceil \frac{\mathcal{S}}{\mathcal{D}_{segsize}} \rceil \times \mathcal{D}_{overhead} + \mathcal{S}_{small} \times \mathcal{D}^{small}_{byte} & \\ \qquad + (\mathcal{S} - \mathcal{S}_{small}) \times \mathcal{D}^{large}_{byte} & \text{otherwise,} \end{cases} \tag{5.3}$$

where $\mathcal{D}_{overhead}$ is the fixed overhead per DMA segment, $\mathcal{D}_{segsize}$ is the DMA segment size used by the CIM, and $\mathcal{D}^{small}_{byte}$ and $\mathcal{D}^{large}_{byte}$ are the incremental costs per byte obtained



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

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

Figure 5.5: Sources of delay in the model and their accuracy.

empirically. The values corresponding to these parameters are listed in Table 5.1. As can be seen from Figure 5.5(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.

5.1.3.2 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:

$$\mathcal{T}_{m(pipe=2)}(\mathcal{S}) = \mathcal{T}_{DMA}(\mathcal{S}) + \mathcal{T}_{FC}(\mathcal{S}) \qquad (5.4)$$

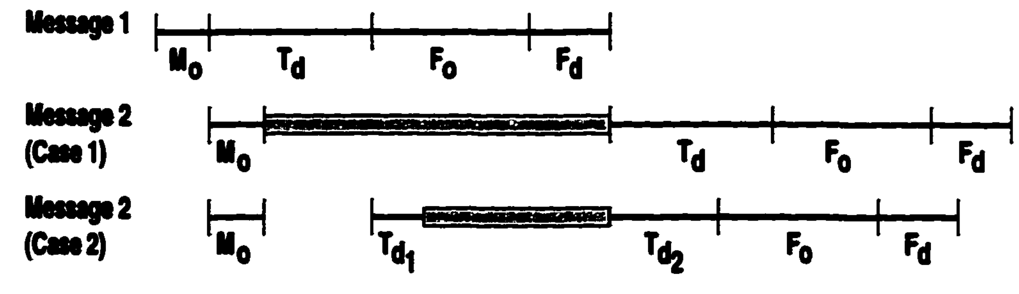


Figure 5.6: Inter-message overlap during CIM transmissions.

This is illustrated in Figure 5.6, 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. 5.2 corresponds to the time line for Message 1, while Eq. 5.4 corresponds to Message 2 (case 1).

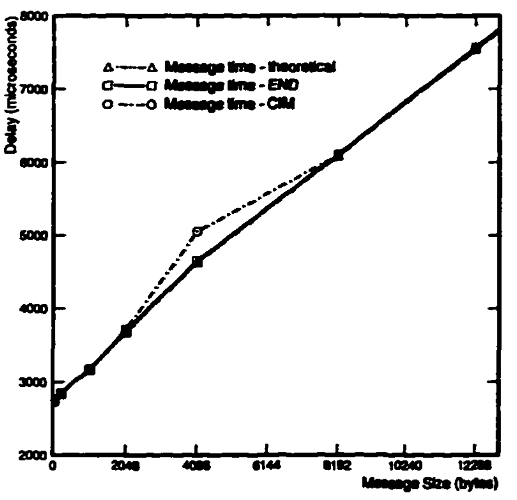
Note that the above analysis should always be optimistic with respect to the corresponding *END* model (Figure 5.7(b)). This is because, in practice, perfect overlap between $\mathcal{M}_{overhead}$ and other operations is unlikely and one must account for the cost of other operations on *END*. The measured 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.

This 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:

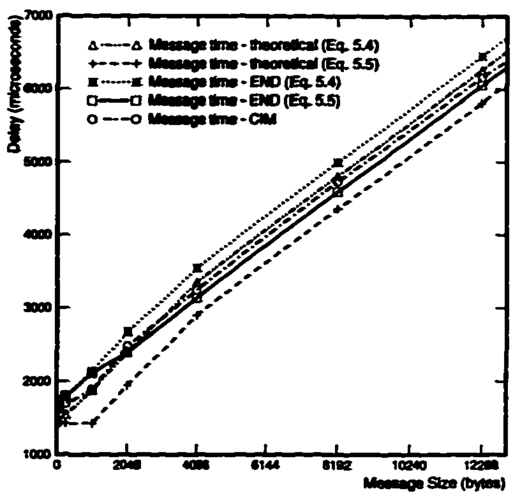
$$\begin{aligned}\mathcal{T}_{m(pipe=2)}(\mathcal{S}) &= \max(\mathcal{T}_{DMA}(\mathcal{S}), \frac{1}{3}\mathcal{F}_{overhead}) + \frac{2}{3}\mathcal{F}_{overhead} \\ &\quad + \lceil\frac{\mathcal{S}}{\mathcal{F}_{framesize}}\rceil \times \mathcal{F}_{framedelay} \qquad (5.5)\end{aligned}$$

This corresponds to Message 2 (case 2) in Figure 5.6, 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. This is because FC operations have higher priority than DMA. The remaining part, T_{d_2}, resumes when Message 1 completes transmission.

Discussion: Figure 5.7(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 5.7(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



(a) Pipeline depth = 1



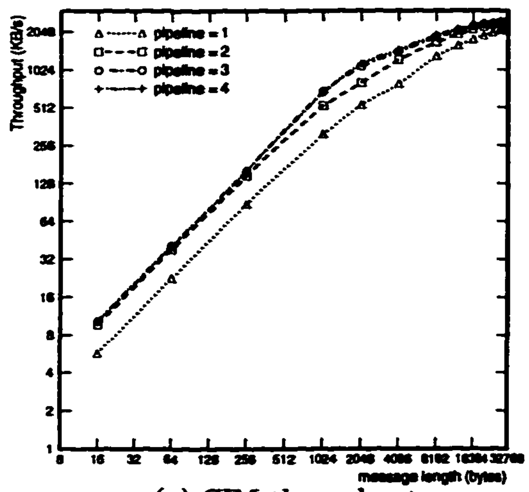
(b) Pipeline depth = 2

Figure 5.7: Theoretical and measured mean message times.

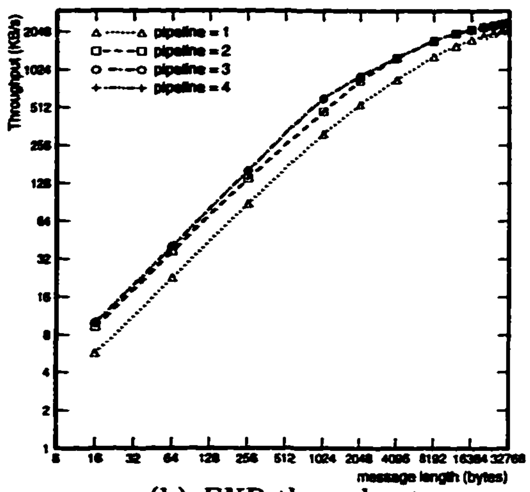
earlier. The *END* model (with Eq. 5.5) shows good agreement with the real CIM, though not as good as with pipeline depth of 1. Note that we are able to construct reasonably accurate theoretical models because the system has homogeneous traffic and is otherwise idle. Under more complex traffic conditions, or with greater pipeline depth, or with 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.

5.1.3.3 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 5.8 shows the mean throughput for the CIM and *END* (note that Figure 5.8(a) is the same as Figure 3.4(a)). 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



(a) CIM throughput



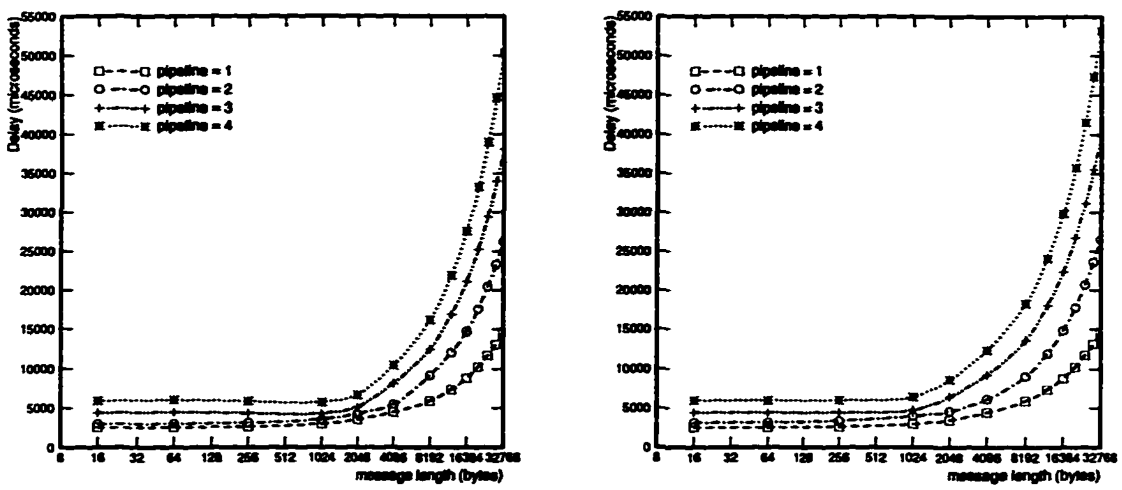
(b) *END* throughput

Figure 5.8: 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.

the number of messages increases. Figure 5.7 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 5.9 shows the mean message delays (note that Figure 5.9(a) is the same as Figure 3.4(b)). 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.



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

Figure 5.9: Comparison of CIM and *END*'s transmission delay.

5.2 CIM Transmission Analysis and Optimization

In this section we demonstrate the use of *END* as a design tool to explore and evaluate design alternatives to improve the handling of outgoing traffic. In particular, 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 modifications to improve performance:

(i) simplification of the host-adapter interface to reduce overhead (Section 5.2.1), and

(ii) architectural changes to increase concurrency in CIM operations (Section 5.2.2).

The performance improvements due to these changes are described in Section 5.2.3.

5.2.1 Reducing the Host-Adapter Interface Overhead

Figure 5.1 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 to be sent after either or both of these phases, as determined by the higher software layers. Note that the equations for delay in this case will be almost identical to Eq. 5.2 and Eq. 5.5; only the value of $\mathcal{M}_{overhead}$ will reduce.

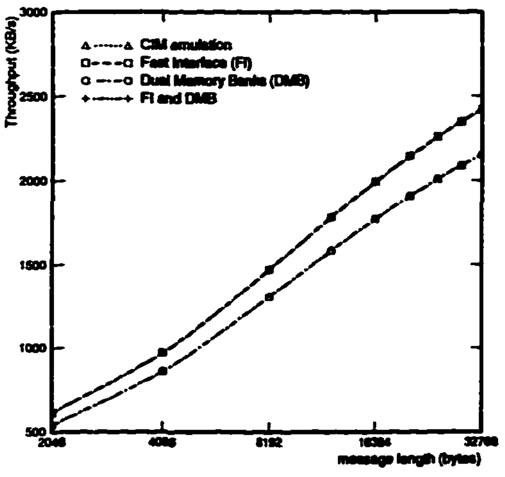
5.2.2 Exploiting Increased Concurrency

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 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 [87]. 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 a pipeline depth of 1, there is no benefit to having two buses. When the pipeline depth is 2, however, the DMA and FC phases can proceed concurrently. As in Eq. 5.5, 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:

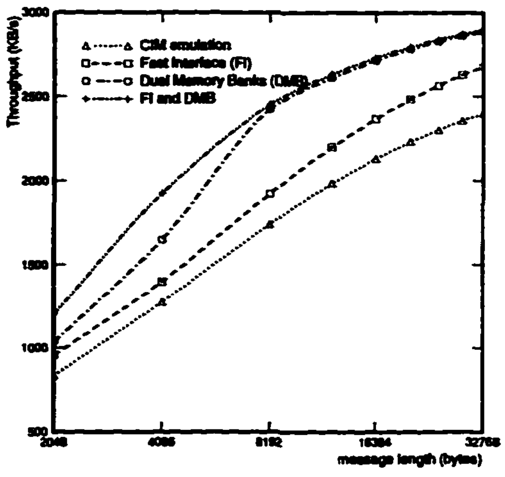
$$\mathcal{T}_{m(twobus,pipe=2)}(\mathcal{N}) = \max(\mathcal{T}_{DMA}, \mathcal{T}_{FC}) \qquad (5.6)$$

5.2.3 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



(a) Pipeline depth = 1



(b) Pipeline depth = 2

Figure 5.10: Transmission throughput for improved CIM models.

so that the DMA and FC phases proceed in parallel, i.e., the FC phase no longer preempts the DMA phase.

Figure 5.10(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 overhead was more significant compared to the cost of transmission[1].

Figure 5.10(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

[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.

number of interactions with the host, and the resulting unpredictability. This effect is more pronounced for small messages since the delay at the interface is a significant part of the total transmission delay.

5.3 Discussion

In Section 5.1 we established that *END* could accurately emulate the behavior of a real network adapter and in Section 5.2 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*.

***END* as a Prototype:** As seen in Section 5.1, 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 (*END* is used in this manner in Chapter 7). 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.

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 overhead of the host operating system and applications is 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 overhead 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 4.4.1, 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. Also, faster CPUs may be modeled by slowing down other activities, so that the CPU is now relatively faster. Scalability and contention limitations may also be addressed by using WAN emulation in the style suggested by *hitbox* [1].

5.4 Conclusions

In this chapter, we used *END* to accurately model the behavior of a real network adapter, the Ancor VME CIM 250, and modify both the hardware and software in its original design to remove some performance bottlenecks. These design improvements yielded throughput increases of 15–50%, depending on the hardware and software modifications and on the traffic load. While we do not claim that the proposed modifications to the CIM are necessary or even desirable, *END* helps a designer evaluate the various alternatives and their corresponding price/performance tradeoffs. Further, this evaluation may be made early in the design cycle, without actually building prototypes, and before it becomes too expensive to make significant modifications. We demonstrate that the the code for the device driver for the *END* model is almost identical to that of the device driver for the real network adapter. Further, we claim that the structure of software in the *END* model itself would be very similar to that of the network adapter firmware. Both these result in reuse of code, thereby minimizing wasted effort in the development of the *END* model.

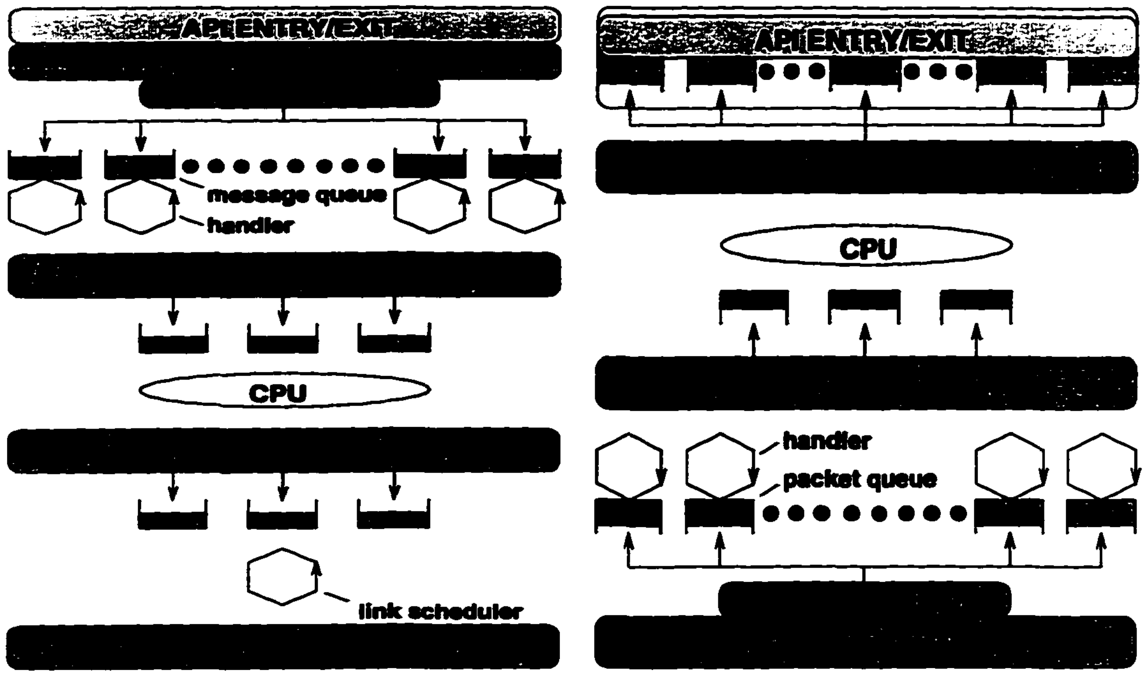
CHAPTER 6

QUALITY-OF-SERVICE ISSUES IN ADAPTER DESIGN

In Chapter 3, we demonstrated how real-time channels were implemented using commercial, off-the-shelf, network components that did not provide any explicit support for QoS. This was achieved by limiting the number of packets queued on the network adapter, and performing deadline-based link scheduling on the host. Other than link scheduling, the host operating system had little QoS support, and simply assigned a higher priority to protocol processing for real-time traffic, than for best-effort traffic.

While the admission control and scheduling policies described so far mainly consider link scheduling as a critical requirement to guarantee deadlines, further analysis [74] revealed that other system overhead could also critically affect system throughput and admissibility of real-time traffic. However, in Chapter 3, even though we did not explicitly consider this overhead, the relatively simple CPU scheduling was quite effective. The reason this worked was that the network link was much slower than the CPU, ensuring that the protocol processing completed before the previous packet transmission, thus making more sophisticated CPU scheduling irrelevant. However, with faster networks, the CPU overhead can become comparable to the network delays, or even become the bottleneck. In such a case, it is necessary that not only the link, but the CPU resources as well, are consumed in an order determined by the QoS requirements of individual applications.

In this chapter, we briefly describe a QoS-sensitive communication subsystem that integrates CPU and link scheduling [74,75]. We consider various configurations of the host operating system, networks and network adapters with differing levels of QoS support. Using *END*, we examine how delivered QoS is affected by the level of QoS support provided by the different components of the end-host communication subsystem. We consider two



(a) Source host (b) Destination host

Figure 6.1: QoS-sensitive communication architecture.

different network models: a point-to-point network and a token-based shared network. Our evaluation results show that adapter QoS support is essential for shared networks while host QoS support may suffice for point-to-point networks, if the number of packets queued on the adapter can be bounded.

The rest of this chapter is organized as follows. Section 6.1 presents a summary of our QoS-sensitive communication software architecture implemented on HARTS [75]. Given this architecture, and the QoS issues in adapter design discussed in Section 4.2, Section 6.2 outlines our research goals and approach. Sections 6.3 and 6.4 present results from performance evaluation for point-to-point networks and shared networks, respectively, and also compare and contrast the two. Finally, Section 6.5 concludes this chapter.

6.1 A QoS-sensitive Communication Subsystem

Figure 6.1 provides an overview of our QoS-sensitive communication architecture (see [75] for details). This architecture provides a *process-per-channel* model of protocol processing adapted from the *process-per-message* model provided by x-kernel. In this model, a unique process, called a *channel handler*, is associated with each real-time channel to perform protocol processing for all messages generated on the channel. An important feature of this architecture is that scheduling may be provided at two stages – CPU scheduling for protocol processing, and link scheduling for packets queued at the network interface. Considering transmission (Figure 6.1(a)), messages transmitted by applications are queued at their channel handler's message queue. The channel handler runs in an infinite loop (for the lifetime of the channel), accepts these messages, performs protocol processing (including packetization), and inserts the packets into a packet queue for its outgoing link. Before a message is enqueued at its handler's queue, the API may perform some traffic enforcement, and assign the message a priority. The handler inherits this priority, and based on the priority and the type of the message, the handler is assigned to the appropriate CPU queue. Similarly, packets may be subject to further policing, and are assigned a priority which determines their link queue. Reception is similar, but with events occurring in the reverse order. In this architecture, note that the policing, priority assignment, and scheduling mechanisms for the CPU and the link queues may be configured independently.

In the implementation of real-time channels described in Chapter 3, policing and scheduling support is provided for packets queued at the link. Policing of packets is performed by assigning a logical arrival time and computing the corresponding packet deadlines, and using a multi-class EDD scheduling algorithm based on these deadlines (Section 3.2). This helps ensure that packets are transmitted in a QoS-sensitive order. This scheme is extended in this QoS-sensitive communication architecture to ensure that all system resources, not just link bandwidth, are consumed in a QoS-sensitive order. Channel handlers inherit message deadlines, and are scheduled for execution using a similar multi-class EDD scheme. The process scheduler is layered above the *x*-kernel scheduler (which provides fixed-priority non-preemptive scheduling with 32 priority levels). Since all channel handlers execute within a single (kernel) address space, the preemption model employed for handler execution is that of *cooperative preemption*. That is, the currently executing handler yields the CPU to a waiting higher-priority handler after processing up to a certain (configurable) number of packets (the preemption granularity). Besides bounding CPU access latency, this allows us to study the influence of preemption granularity and overhead on channel admissibility [74].

Link scheduling decisions are made by calling a scheduling function either by the channel handler, or by the transmission-complete interrupt service routine (Option 1 in [74]).

In order to support real-time communication, network adapters must provide a bounded, predictable transmission time for a packet of a given size. Since network adapters are typically best-effort in nature, their designs are usually optimized for throughput and may be unsuitable for real-time communication, even with a bounded and predictable packet transmission time. Even when explicit support for real-time communication is provided, on-board buffer space limitations may necessitate staging of outgoing traffic in host memory, for subsequent transfer to the adapter. To support real-time communication on these adapters, link scheduling must be provided in software on the host processor. In our implementation, packets created by channel handlers may be scheduled for transmission by a non-preemptive multi-class EDD link scheduler, or, in case the adapter provides adequate QoS support, may simply use FIFO scheduling.

The architecture above has been demonstrated to provide overload protection and fairness, and maintain QoS guarantees [75]. Overload protection is provided by per channel traffic enforcement, both for the CPU scheduling and link scheduling. It is also fair, since early real-time traffic is delayed till its logical arrival time, and does not take away resources from best-effort traffic. When combined with channel admission control [74], since the order of CPU processing and link transmission is determined by message deadlines, this also guarantees that all messages meet their deadlines.

6.2 Research Goals and Approach

In this chapter, we focus on some of the QoS issues highlighted in Section 4.2, namely, the necessity and effectiveness of QoS support on the adapter, given various levels of QoS support on the host. That is, we want to explore the need for, and the type of, QoS support on the network adapter if the host provides each connection with its associated QoS for protocol processing. Our goals are to demonstrate that: (i) some form of adapter QoS support improves performance, either by eliminating FIFO queuing (and hence, priority inversion) on the network adapter, or by reducing host CPU load by offloading some functionality from the host to the network adapter, or some combination of these factors; (ii) QoS support in adapters interfacing to shared networks is necessary in order to keep link utilization high and prevent QoS violations; and (iii) *END*-based device emulation provides a reasonably accurate and rich framework to study these tradeoffs.

Our approach is to evaluate different adapter configurations using *END*, with the QoS support provided ranging from a single FIFO queue to a deadline-based priority queue. For all these configurations, *END* implements suitably QoS-enhanced host-adapter and adapter-network interfaces with shared packet buffers. The host QoS support we consider is for real-time channels [63], in the form of QoS-sensitive protocol processing via deadline-based CPU scheduling of channel handlers, as described in the previous section. In some of the experiments, some of the CPU support may be relaxed. The workload used for the evaluation comprises a mix of real-time channels, multiple channels generating traffic based on MPEG traces [86], and best-effort traffic.

6.3 Point-to-point Network Model

Point-to-point networks such as switch-based and mesh-based topologies are characterized by dedicated links between source and destination hosts such that a source host exercises complete control over access to its attached link. There is no interference due to contention from any other host. This implies that data transmission can begin on the link as soon as the host generates a packet and submits it to the adapter. Even if the host sorts packets in an order determined by their QoS requirements, there is potential for priority inversion (i.e., an urgent packet arriving late is queued behind less urgent packets) if there is QoS-insensitive queuing of packets on the network adapter. We use *END* to understand to what extent such queuing or provision of QoS support affects the delivered QoS guarantees.

At one extreme, the host may treat the adapter as a black box, exercising complete control over transmission order by initiating transmission of a packet only after the previous one has completed. At the other extreme, the host does not limit the number of packets it may queue on the adapter, and the adapter is responsible for data transmission order, and sorts packets by a deadline supplied by the host. In between these two extremes, the host exercises partial control over transmission order by sending a finite (small) number of packets to the adapter for transmission. These packets are sorted by their deadlines before being sent to the adapter, but, since the adapter transmits these packets in FIFO order, a packet that arrives later, and with a tighter deadline, may suffer some priority inversion.

6.3.1 Experimental Configuration

Transmission is modeled as a two-step process – a DMA transfer from the host to *END* followed by data transmission from *END* to the network medium. At any given time, the "DMA transfer" and "network transmission" can occur simultaneously, but for different packets (i.e., the network adapter has dual ported memory and dual buses to permit such simultaneous operations). No more than one operation of each kind can occur at a time, since each requires a shared resource like a system bus or network link. When link scheduling is performed on *END*, the DMA transfers are ordered by packet deadlines and the transmission is performed in FIFO order, i.e., the order of completion of DMA. If link scheduling is performed on the host, *END* simply uses FIFO ordering.

We performed four sets of experiments, as shown in Table 6.1. Stage 1 represents DMA of packets from host to adapter memory, and Stage 2 represents network transmission. Each data point in the graphs corresponds to behavior measured over the transmission of over 30,000 packets. Experiments were repeated over several runs to confirm their validity, and no significant differences were observed from one run to another. In the first experiment, link scheduling is provided on the emulator, i.e., as soon as the host completes a packet's protocol processing, it issues a transmit command to *END*. In effect, the host does not bound the maximum queue length on the emulator (in practical terms, it is bounded by the relative speeds of the host and the "network" and by the buffer sizes on the emulator). In the next three experiments, the host sorts the packets according to their deadlines, and *END* transmits them in the order received. To bound potential priority inversion due to FIFO queuing on *END*, the host limits the number of unacknowledged packets queued on the emulator.

In these experiments, the DMA has a delay of 50 ns/byte, and the network transmission requires 40 ns/byte. Since the worst case delay for any stage is 50 ns/byte, given the overlap of the two stages, we should expect the effective data transfer rate to correspond to a delay of 50 ns/byte, in addition to per packet overhead for queuing operations, network overhead, interrupts etc. Note that similar experiments were performed for a large range of DMA and network delays, and performance was determined largely by the greater of the two delays. This is to be expected, since the slowest resource will always be the performance bottleneck.

Experiment Number	Link Scheduling	Queuing Stages (Type, Delay (ns/byte))		Max. queue length on emulator
		Stage 1	**Stage 2**	
1	**On the Emulator**	**EDD, 50**	**FIFO, 40**	–
2	**On the Host**	**FIFO, 50**	**FIFO, 40**	2
3		**FIFO, 50**	**FIFO, 40**	3
4		**FIFO, 50**	**FIFO, 40**	6

Table 6.1: Experimental configurations and parameters

6.3.2 Evaluation Workload

Table 6.2 lists the test workload comprising of traffic carried on a set of real-time channels specified by their maximum message size, S_{max}, and the minimum interarrival time between messages, I_{min}. The long-term average data rate does not exceed R_{max} ($= S_{max}/I_{min}$), but short-term bursts of up to B_{max} messages may occur. The real-time channels were established using the analysis and techniques presented in [74]. Using EDD scheduling for protocol processing (with packets between preemption set to 4) and EDD link scheduling, all real-time traffic is guaranteed to meet its deadline. All messages are 60 KB, and the maximum packet size is configured as 4 KB. Traffic on channels 0 and 1 is bursty, while channel 2 has strictly periodic real-time (RT) traffic. Channel 3 carries best-effort (BE) traffic and its offered load varies from about 235 KB/s to 7500 KB/s. We study the performance of the BE and RT traffic as a function of offered BE load.

6.3.3 Best-effort Performance

Figure 6.2 shows the throughput and mean delays achieved by BE Channel 3 as a function of the offered load. Since the link stage is faster than the DMA stage, we would expect link transmission time to be completely hidden by the DMA time of the next packet. Considering DMA and network delays, and additional per packet overhead (Section 4.4.1), the network capacity is limited to approximately 15 MB/s. We are unable to achieve this level because the host CPU cannot generate data at that rate, i.e., the CPU is the bottleneck. For each experiment, the delivered throughput rises with the offered load and eventually saturates, at which point the average packet latencies rise rapidly. Saturation occurs significantly later when link scheduling is performed on *END* (Experiment 1). Since protocol processing

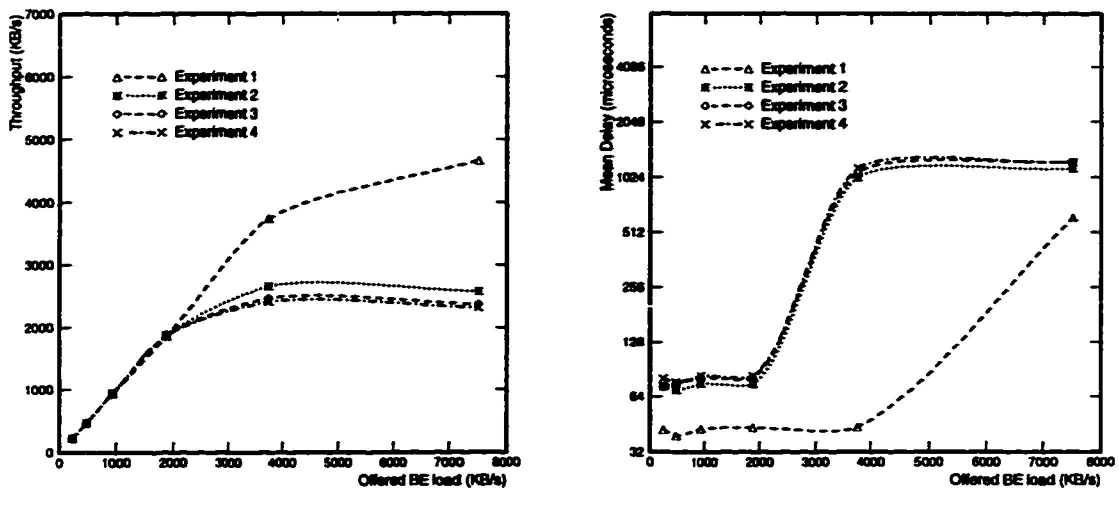
Channel	Type	Traffic Specification				Deadline (ms)
		S_{max} (KB)	B_{max} (messages)	R_{max} (KB/s)	I_{min} (ms)	
0	real-time (RT)	60	12	1200	50	40
1	real-time (RT)	60	8	2000	30	25
2	real-time (RT)	60	1	2000	30	30
3	best-effort (BE)	60	10	variable	–	–

Table 6.2: Workload used for evaluating the point-to-point network.

is expensive, and the CPU is the bottleneck in this configuration, reducing the CPU load simply by moving some of the scheduling operations to *END* has significant benefits since, in this case, *END* performs relatively simple functions and has spare CPU capacity. Since admission control for the real time channels included the cost of link scheduling, using adapter based QoS reduces the CPU requirements and increases the admissibility of real-time traffic. Having a faster CPU on the host may not be adequate because applications will always compete with communication processing for the CPU. Since real-time traffic has a higher priority, it continues to meet its QoS requirements independent of the BE load (Figure 6.3). Hence, BE traffic only gets the remaining portion of the processing capacity, and gets substantially affected by any reduction in it.

In Experiments 2–4, link scheduling is performed on the host and the maximum number of packets queued on the adapter is bounded to control priority inversion. To use the link capacity effectively, we need to ensure that at any given time, there are at least two packets being processed concurrently on the adapter, one in the DMA phase, and another in the transmission phase. Since *END* is not configured for more than two phases, there is no further gain in concurrency or performance in Experiments 3 and 4. There is a small increase in queue lengths on *END*, and hence, slightly higher delays. The graphs show a small decline in performance at this point.

The main conclusions from these experiments are that performance may be optimized by knowledge of the internal structure of the network adapter, and that the host operating system must configure the number of packets it may pipeline on the adapter in order to exploit the greatest possible concurrency between the host and the different stages of processing on the adapter. Any further pipelining of packets will not increase concurrency, and hence has little or no benefit, and may actually reduce system throughput while exacerbating priority inversion.



(a) BE Throughput (b) BE Delay

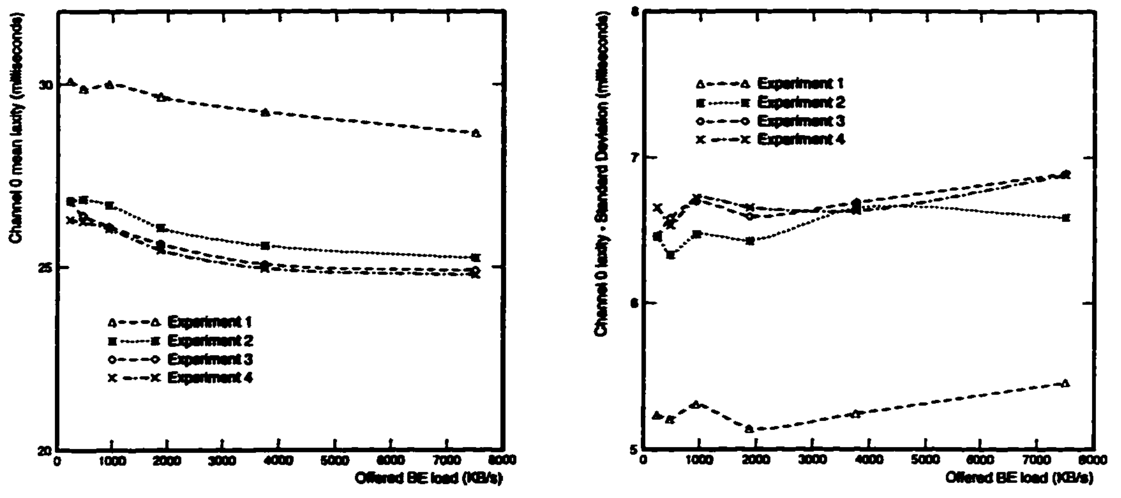
Figure 6.2: Best-effort performance.

6.3.4 Real-time Performance

Figure 6.3 shows the mean laxity and jitter in laxity of a real-time channel. In all the experiments, all real-time deadlines were met. Delay jitter was slightly lower (about 1 *ms*) for Experiment 1 compared to Experiments 2–4 since the link scheduling decisions were made closer to the transmission time, thereby reducing the chance of priority inversion. In Experiments 2–4, not only does message latency increase as we increase the maximum queue lengths on the adapter, but the jitter increases as well. However, at the configured speed, a 4 KB packet gets transmitted in about 160 μs (plus the overhead), which is small compared to the end-to-end delay bound and some other communication costs, and hence any increase in priority inversion is quite small.

6.3.5 Discussion

The CPU could optimize BE performance by matching the transmission request pipelining to the pipelining of different stages on the adapter. This worked very well in our example since all packets were the same size. Performance would almost certainly decline in the presence of bimodal traffic, with a mix of very small and maximum sized packets, since successive packets may no longer have the maximum possible overlap as the relative delays of different stages on the adapter would vary with the size of the packets they are processing.



(a) Mean real-time laxity (b) Standard deviations of real-time laxity

Figure 6.3: Real-time performance.

It would be desirable to have more packets queued on the adapter to maximize the chances of using all stages to the maximum extent.

One would expect that with deadlines computed on the host, and longer queues on the adapter, we would see greater delay jitter for real-time data due to the increased possibility of priority inversion. However, this did not happen for several reasons: (1) since the link was dedicated to a single CPU, there was no queue build up and delay due to contention for the link; (2) the CPU was not fast enough to build very long queues on the adapter; (3) transmission delay for an individual packet is small compared to its deadline, thus limiting the extent of worst case priority inversion; (4) the number of connections was quite small, reducing the chances of interference; and (5) all data in this experiment had maximum sized packets, and similar periods and deadlines.

If the CPU were fast enough to build up queues on the adapter, it might be argued that it derives no advantage from queuing packets on the adapter, and it could restrict queue lengths to a small number, and still keep all stages of the adapter as busy as possible, while exercising greater control over transmission order. On the other hand, if the CPU is slower than the link, it becomes the bottleneck and thus will not be able to build up any queues on the adapter. Either way, it would appear that the host can keep tight control over jitter without any special support from the adapter.

However, in networks without dedicated links (i.e., with shared links as in Ethernets, token buses, token rings, FDDI, etc.) multiple hosts contend for access to the link. While the link is unavailable, the host can build long queues of packets on the network adapter, which get drained in bursts whenever the adapter gets access to the network. In the next section, we consider an adapter for such a network, and determine the nature and efficacy of its QoS support.

6.4 Shared Network Model

Various schemes have been proposed to support real-time traffic over shared networks such as FDDI [26] and Ethernet [100]. Typically, such schemes involve passing a token between the nodes on the network, and permitting the host holding the token to transmit data. The duration the host may hold the token (called the *Token Holding Time* (*THT*)) is a measure of the bandwidth made available to the host each time it receives the token. If each host bounds its *THT*, it is possible to bound the time it takes for the token to rotate through the network (called the *Target Token Rotation Time* (*TTRT*)). Since each node has a minimum bandwidth allocation and a worst case latency to receive that bandwidth, it may use this to provide real-time guarantees to data connections originating on it.

Determining suitable values for *TTRT* and *THT* requires consideration of overhead and QoS constraints. For each cycle of the token, the overhead includes network latency (determined by the length of the network), token passing overhead, and the number of nodes. We need to ensure that this overhead is not too great a proportion of the *TTRT*. However, we cannot make *TTRT* too large either, since that would restrict the ability to meet tight deadlines. Once *TTRT* is established, individual nodes may reserve bandwidth for themselves by setting their value of *THT*.

As noted at the end of the previous section, when the network link is not available continuously, the host can build long queues of packets on the network adapter, which get drained in bursts whenever the adapter gets access to the network. In such a situation, even with careful scheduling of packets on the host, there is potential for significant queuing delays and priority inversion if the adapter does not provide any QoS support. In this section, we describe how *END* is used to model one such network, and the efficacy of different approaches to solve this problem.

6.4.1 Network Model

We use *END* to implement a model of a token-based shared network. This model is similar to the point-to-point model, but we need to add a token passing scheme, and different scheduling strategies. A token is created and its behavior is specified in terms of its *THT* and *TTRT*, whose values are specified at initialization time. An outline of the token passing and management is presented in Figure 6.4. On initialization, the token is released immediately (`ReleaseToken()`), and a request is issued to the time device that the function `TokenArrive()` be invoked after a delay of *TTRT*−*THT*. `TokenArrive()` sets a flag indicating that the token is at this node, requests that `TokenExpire()` be invoked after a delay of *THT*, and then invokes the function `Transmit()`. As long as the token is at the node, `Transmit()` transmits data, if available. If there is no data to transmit, it releases the token immediately so that the next node in the sequence may get a chance to transmit data. When `TokenExpire()` gets invoked (after a delay of *THT*), if it finds that the token has already been released (because there was no data to transmit), it does nothing. Otherwise, if the link is idle, it releases the token immediately, else it clears the token flag, so that `Transmit()` will release the token as soon as it completes transmitting the current packet. If the token expires during packet transmission, it is held till the completion of that packet transmission, i.e., it is held for a little longer than *THT*. This problem is overcome by noting the excess time the token was held, and reducing the *THT* by that amount the next time the token arrives. In this way, each node is guaranteed a fraction $\frac{THT}{TTRT}$ of the link bandwidth, and does not have to wait any longer than *TTRT* to receive the token.

6.4.2 Experimental Configuration

We used the above model of *END* to study the behavior of a mix of peak rate real-time traffic, MPEG traces and best effort traffic on a shared 1 Gigabit per second network. Values of *TTRT* and *THT* are chosen such that each node receives 10% of the link capacity, i.e., 100 Mbps or 12.5 MB/s. As noted in Section 4.4.2, due to the system overhead, we cannot use the full capacity made available to us. For example, the transmission time for a 4 KB packet is about 32 μs, but with an overhead of 48 μs per packet, only 40% of the link bandwidth is available to us. Since many MPEG frames are much smaller than 4 KB, the utilization is even lower. Further, the host CPU is also too slow to generate data fast

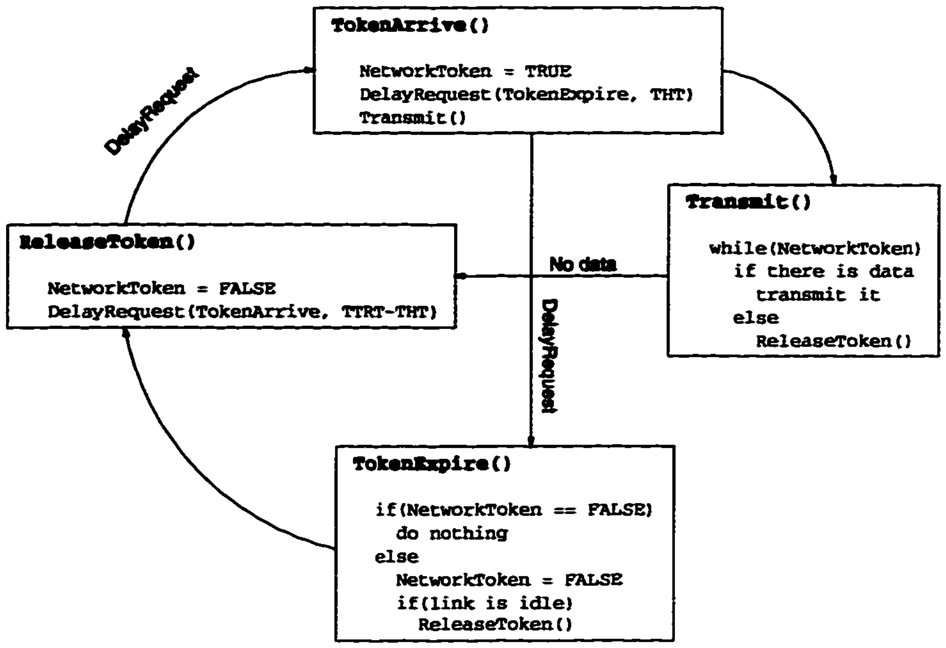


Figure 6.4: Token management on *END*.

enough to saturate this link. To ameliorate this problem, we used a CPU speedup factor (C_{sf}, see Section 4.4.2) of 12 to represent the behavior of a processor with a SpecInt92 rating of about 147.6 (roughly comparable to a 133 MHz Pentium, which has a SpecInt92 rating of 155). In this configuration, we found that we were able to utilize just over 89% of the network capacity for purely best-effort traffic (compare this with 88.9% predicted by Equations 4.1 and 4.2), and a little less for a mix of real-time and best-effort traffic (around 83% for our test workload). Also, most of our experiments used the optimizations described in Section 4.4.1 that performed queue operations concurrently with the transmissions. For some experiments, we contrast these results with those of the unoptimized system (marked *no opt* in the graphs).

The host configuration is identical to that described in Section 6.1. As in Section 6.3, the number of packets between preemption was set to 4, and the maximum packet size was 4 KB. However, instead of reserving capacity for real-time traffic using real-time channels and providing absolute deadline guarantees, we simply reserved bandwidth for each connection and ensured that it was provided its data rate using the run-time mechanisms described in [75]. For various configurations of the host and emulator, we evaluated the deadline compliance of the various schemes.

Channel Numbers	Channel type	Bandwidth Reservation (%)	Utilization (%)
0–5	Real time	35.2	35.2
6	Best effort	–	residual
7–74	MPEG	64.8	36.8

Table 6.3: Workload used for evaluating the shared network.

6.4.3 Evaluation Workload

Table 6.3 shows a summary of the test workload and observed throughput behavior. We used three distinct types of data sources: a best-effort channel with a variable load, 6 synthetic real-time channels, and 68 MPEG channels. We first reserved capacity for the real-time channels, and then reserved the remaining capacity for MPEG channels based on traces from 6 sources (Table 6.4). The traffic parameters for both the MPEG and the real-time channels were specified using the same parameters as were used for the real-time channels in Table 6.2. The synthetic real-time channels generate data at their peak rate. Since the MPEG sources are extremely bursty, we over-reserved capacity for them, to the extent of one standard deviation more than their mean data rate. This implies that even though we reserved 100% of the link (of which only 83% is usable) there was still unused bandwidth, which was absorbed by the best effort channel.

Many MPEG sources and their properties are described in detail in [86]. The frames are 384x288 pixels in a cycle of 12 with the frame sequence IBBPBBPBBPBB. We selected six different traces, two each of movies, sporting events, and television news and talk shows, and have summarized their properties in Table 6.4. We generated multiple instances of each channel, and generated frames of the sizes given in the traces every 33 *ms*. To avoid problems of correlations of frame sizes with multiple instances of the same trace, we staggered the starting time within the trace by one minute for each instance. Since all the MPEG channels have the same period, we randomized their starting times to reduce the probability of all of them generating frames for transmission in rapid succession (when we did not randomize the starting times, we observed significantly worse behavior for those channels that consistently generated data just after two others).

	Name	Type	Frame Size (bits)			
			min	max	mean	std. dev.
1	dino	movie	880	119632	13078	14749
2	lambs	movie	288	134224	7311	11195
3	atp	sports	280	190856	21889	20408
4	race	sports	4192	202416	30749	21167
5	news	news/talk	16	194416	20664	25992
6	talk	news/talk	2080	106768	14536	16519

Table 6.4: Traffic characterization of sample MPEG traces from [86].

6.4.4 Host and Emulator QoS Support

The host generates data from the 75 channels as described above and issues transmit commands to the emulator. When the emulator does not have the network token, it performs any necessary DMA and interface operations. However, when the token arrives at the network, it dedicates its entire capacity to transmit any packets queued on it. With a TTRT of 10 *ms* and a THT of 1 *ms*, we observed that during the THT, between 26 and 65 packets were transmitted, depending on the sizes of the packets queued on the adapter. Clearly, in order to ensure that the link is kept busy at all times, the emulator must ensure that it has at least 65 packets queued for transmission. Without appropriate support, there is potential for plenty of interference between packets of different real-time connections, and between real-time and best-effort packets resulting in priority inversion and missed deadlines.

As in Section 6.3, the host used QoS-sensitive CPU scheduling for all protocol processing. We studied three different approaches for determining the network transmission order, identified in the graphs in Figure 6.5 using the labels below:

1. *Emulator QoS (EQoS):* The host issues transmit commands to the emulator as soon as it completes protocol processing. The emulator enqueues BE packets in a FIFO queue, and RT packets in a priority heap sorted by their deadline, and performs transfers giving higher priority to RT packets. On completion of DMA, the RT packets are inserted into another priority queue sorted by their deadlines, and the BE packets are inserted into another FIFO queue, and transmitted using the same priorities as for the DMA stage. *EQoS, no opt* is a similar configuration, but without the optimizations described in Section 4.4.1.

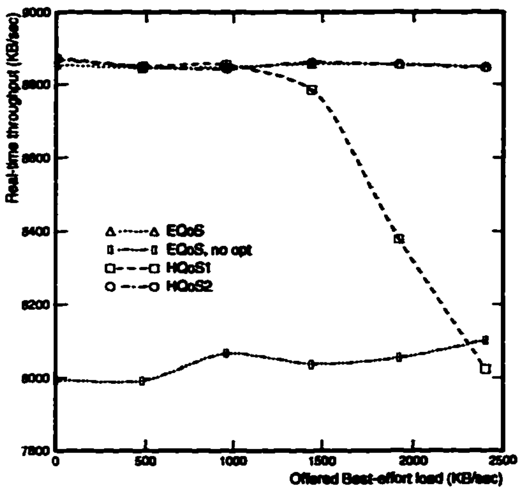
2. *Host QoS, 1 queue (HQoS1):* The host determines the transmission order, and the

emulator uses a single FIFO for all the BE and RT packets (separate FIFOs for the DMA and transmission).

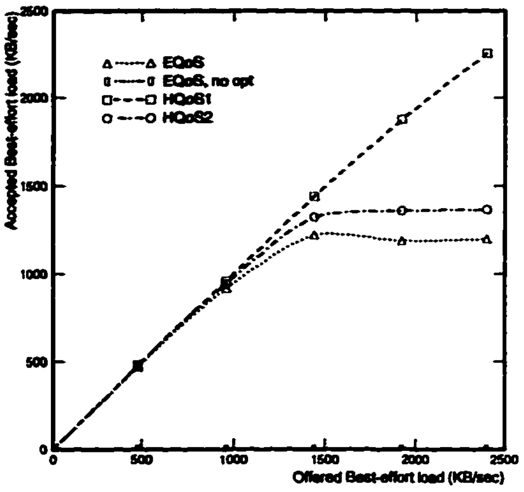
3. *Host QoS, 2 queues (HQoS2):* The host determines the transmission order, and the emulator uses two FIFOs for each operation to separate the BE and RT traffic. RT packets are given higher priority than BE packets (separate FIFOs for the DMA and transmission).

The performance of these configurations is summarized in Figure 6.5. Figure 6.5(a) shows the throughput for real-time traffic as the offered BE load rises. In the case of *EQoS* and *HQoS2*, the RT throughput is not affected by the increasing BE load since RT packets always have a higher priority than BE packets. However, in the case of *HQoS1*, BE and RT traffic share the same queues on the emulator, and the RT throughput starts to decline as the system saturates. Expectedly, Figure 6.5(b) shows the opposite behavior for BE traffic. As the system saturates, the BE throughput saturates for *EQoS* and *HQoS2*, but continues to rise for *HQoS1*. Figure 6.5(c) shows the total system throughput in all three cases. While the differences are small, they are readily apparent. The reason the throughput is higher for *HQoS1* than *EQoS* is that the overhead for heap operations is 53 μs, compared to 48 μs for FIFO operations. We also see that using the unoptimized emulator (*EQoS, no opt*), throughputs are significantly lower. Since the system capacity is lower than the offered RT load, the BE throughput drops to zero for this set of experiments.

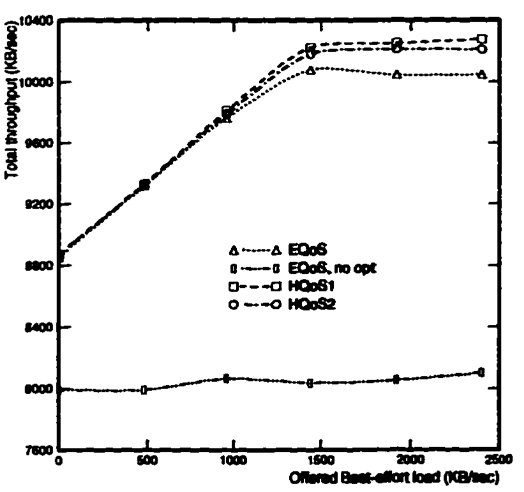
Figure 6.5(d) provides an overview of the delivered QoS, as measured by deadline misses for RT packets and drops of BE packets. For *EQoS*, over 99% of the packets meet their deadlines, and *HQoS2* is almost as good, with just 3–4% of the packets missing their deadlines. Note that the number of deadline misses does not increase with increasing BE load. This is because of the isolation provided by having two queues on the adapter. All deadline misses are due to priority inversion caused by interference between packets of different RT channels. This effect gets magnified by having traffic with different periods and deadlines. Repeating these experiments with only MPEG streams for RT traffic reduced the deadline misses to almost zero, since now all packets have identical periods and deadlines. For *HQoS1*, RT packets are not protected from BE packets, and as the BE load rises, almost all the RT packets miss their deadlines. This happened because we did not bound the queue lengths at all on the emulator. As in Section 6.3, we now bounded the pipeline



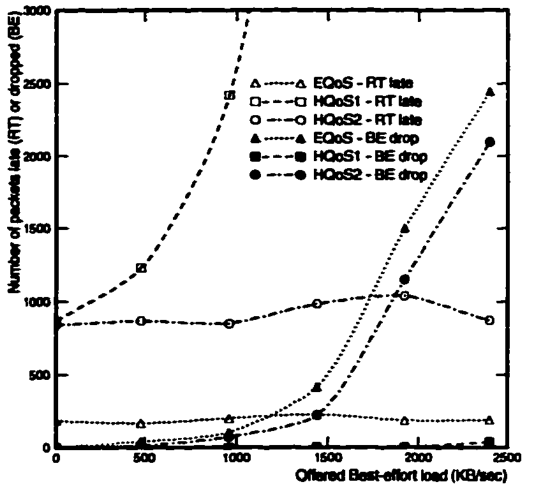
(a) Real-time throughput



(b) Best-effort throughput



(c) Total throughput



(d) QoS summary

Figure 6.5: Performance summaries: Each data point corresponds to approximately 26,000 RT packets. The BE load varies.

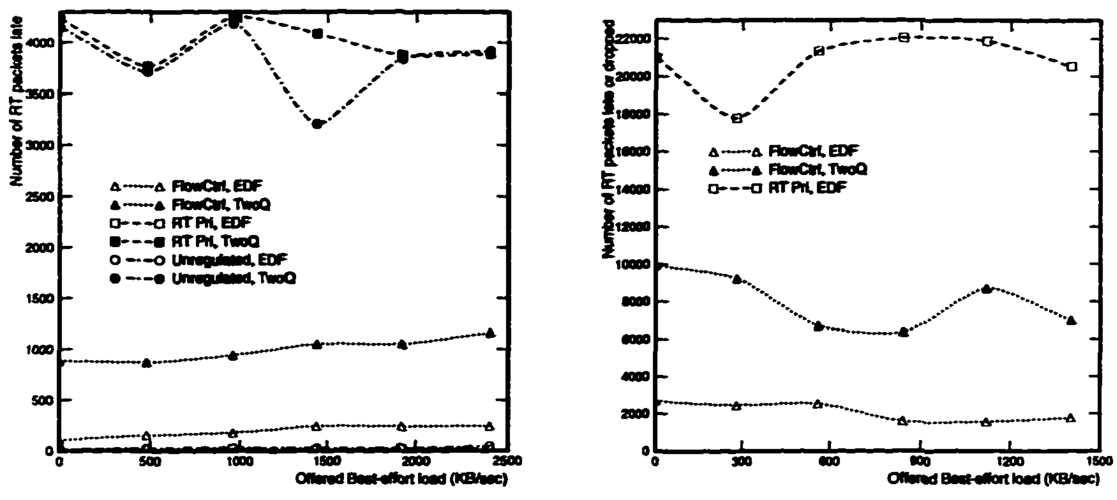
depth to 60, and the RT performance improved dramatically for *HQoS1*, with only 15% of the packets missing deadlines even under heavy BE load. To further improve the RT performance, we reduced the *TTRT* and *THT* to 1 *ms* and 0.1 *ms*, respectively, and fixed the pipeline depth at 6. This reduced the maximum priority inversion to the order of a few (1–3) milliseconds, rather than a few tens of milliseconds seen earlier, and a negligible proportion (0.07%) of real-time packets missed their deadlines. However, due to increased token processing overhead, the total system throughput declined slightly from 10.3 MB/s to 9.8 MB/s. Though this looks like a simple and attractive solution, reducing the values of *TTRT* and *THT* to very low levels may not be practical in real systems.

All the trends observed for the aggregate of real-time traffic were observed in individual channels as well. Both MPEG and other real-time channels had similar laxities and delay jitters. However, the tighter the deadline of a real-time channel, the more likely it was to have packets missing deadlines.

6.4.5 Effect of CPU Capacity and Scheduling on QoS

The above results demonstrate that queuing delays and priority inversion on the adapter impact provision of QoS more in shared networks than in point-to-point networks. While reasonable QoS behavior was observed simply by isolating RT from BE traffic, even better results were observed by isolating individual channels from one another via deadline-based scheduling of packets on the network adapter. The three queuing policies that we considered differ significantly in their level of complexity, cost (i.e., overhead), and performance (i.e., ability to maintain QoS).

While comparing these alternatives, we note that their efficacy is strongly influenced by the QoS support provided on the host [75]. Since the host sends (generates) packets in order of their deadlines (due to QoS-sensitive CPU scheduling of channel handlers), reordering of RT packets is required only for urgent packets that arrive subsequently. Since, in our workload, the traffic was largely periodic with similar deadlines, simply isolating RT traffic from BE (the *HQoS2* configuration) was very effective. *HQoS1* also performed reasonably well once maximum pipeline depth was enforced, since, even though RT traffic is not completely isolated from it, the impact of BE traffic is limited. Further, the host also policed and shaped traffic to ensure that no RT connection used more than the capacity reserved, at the expense of other RT traffic. In the absence of such policing and shaping, even



(a) $C_{sf} = 12$ (b) $C_{sf} = 7$

Figure 6.6: QoS summaries: no link scheduling on the host. Each data point corresponds to approximately 26,000 RT packets. The BE load varies.

a single misbehaving RT connection could cause other RT packets to miss their deadlines. Under such circumstances, even *EQoS* would not be adequate unless it incorporates policing and shaping functions. If the host could work ahead in a work-conserving fashion (i.e., generate packets not necessarily in strict order of deadlines), we would expect *EQoS* to be much more effective than *HQoS2*.

In order to evaluate the impact of CPU support, additional experiments were performed with reduced OS support for QoS. In all these experiments, link scheduling was removed from the host, but execution order of channel handlers continued to be determined by packet deadlines. The policing of channel handlers was configured in the following three ways:

1. *FlowCtrl*: The OS provides all policing and scheduling as in the *EQoS* configuration. Early RT packets are not processed until their conformance times.

2. *RT Pri*: Protocol processing is enabled for early real-time traffic, and has a higher priority than BE traffic. Cooperative preemption remains enabled.

3. *Unregulated*: Cooperative preemption is disabled. A RT channel handler now drains the entire queue before allowing other channels to run.

In each experiment, an OS configuration was selected on the host, and either EDF or two queues (*TwoQ*) on the emulator, as shown in Figure 6.6. The experiments were repeated for two different host speeds, C_{sf} of 12 and 7. Note that *FlowCtrl, EDF* is the same as *EQoS*. In Figure 6.6(a), with *FlowCtrl, TwoQ*, the behavior is the same as that of *HQoS2* in Figure 6.5(d), i.e., host CPU scheduling seems to be adequate, and removal of link scheduling on the host does not seem to matter. With reduced OS policing (*RT Pri* and *Unregulated*) and EDF on the emulator, the number of late RT packets is negligible. This is due to the fact that the RT handlers have a greater priority than BE handlers, and this improved performance comes at the cost of BE throughput. However, with just *TwoQ* on the emulator, the number of RT packets that are late increases significantly. Still, just approximately 15% of the packets are late, which is much better performance than we expected, based on our earlier experience [75]. The reason that QoS does not suffer very much even with relatively weak CPU support is that the CPU is not a bottleneck, i.e., there is enough CPU capacity to complete all the protocol processing on time.

To verify that CPU scheduling is needed, we repeated these experiments with a slower CPU configuration, $C_{sf} = 7$. Now, CPU capacity is barely adequate to process all the packets, and we begin to see significant packet losses. Figure 6.6 shows that when both the CPU and the link are close to saturation, packet transmission order becomes much more important, and both host and adapter support are essential for QoS. With *FlowCtrl, EDF*, only 10% of the RT packets are dropped or miss their deadlines, while as many as 80% of the RT packets are late or dropped when we reduce the policing on the host even slightly. With a slower CPU ($C_{sf} = 6$), almost all packets missed their deadlines, regardless of the CPU and adapter policies, and with a faster CPU ($C_{sf} = 8$), performance was similar to that seen in Figure 6.6(a). This demonstrates that the adapter and CPU scheduling policies are critical as the system approaches saturation. We observed similar results with $C_{sf} = 12$, by running a high priority application that reduced the CPU capacity available for protocol processing. Hence, for any kind of QoS guarantees, it is important that there be adequate link and CPU capacity for communication, and suitable scheduling and policing be employed. We note, further, that there were no ill-behaved real-time connections (those that generate data in excess of their reserved capacity) in our experiments, and CPU policing is even more important to protect well-behaved connections from such ill-behaved connections.

From the above discussion, it is clear that the nature of adapter QoS support required depends not only on application requirements and network configuration, but also on the extent to which QoS support is provided on the host. While adapter QoS support helps provide better QoS compliance, as in the case of point-to-point networks, it also has a secondary benefit of reducing the load on the host, allowing the latter greater capacity for other tasks. Our approach also illustrates the flexibility *END* offers in exploring the most effective partitioning of QoS support between the host and the adapter, taking into account the actual cost-performance tradeoffs on the target platform.

6.5 Conclusions

In this chapter, we explored QoS support in network adapters for provision of QoS guarantees. This study was performed on *END*, an emulated network device that we designed and implemented to explore various QoS issues in network adapter design. *END* is capable of representing the functionality of a network adapter at different levels of detail, with varying degrees of support for QoS guarantees. Since it interfaces with a real host, *END* operates in real-time and can be quite effective in evaluating network adapter and host-adapter interface designs early in the design cycle. Moreover, unlike simulation or mathematical modeling, *END* provides the ability to accurately account for low-level system overhead and concurrency, while retaining the flexibility of simulation and avoiding the complexity of analytic models. The hardware/software codesign facilitated by *END* can help design network adapters that integrate well with the host architecture and operating system.

Using *END*, we evaluated the necessity and effectiveness of QoS support on the adapter, given sufficient QoS support on the host, for two distinct network access models, namely, point-to-point networks and shared networks. Using a mix of synthetic and realistic real-time traffic, we demonstrated that providing QoS support on the adapter using priority-based packet ordering improves performance relative to the case when such ordering is determined on the host. The efficacy of the QoS support provided is not as apparent for point-to-point networks when the network bandwidth matches or exceeds host processing capacity, since the host is unable to generate sufficient traffic to cause appreciable queuing on the adapter. If the host processing capacity is significantly higher than network bandwidth, the bottleneck will shift to the adapter, resulting in queue buildup, and hence the need for QoS on the adapter. This effect is readily observed in the case of shared networks, where a given host may utilize only a fraction of the available network bandwidth. For a fixed host processing capacity, the effective network bandwidth is thus significantly lower, necessitating provision of QoS-sensitive queuing and packet selection policies on the adapter.

END has provided us with a flexible and accurate environment to study adapter design tradeoffs while capturing the effects of subtle interactions between the host communication software and the network adapter. Accomplishing the same via simulation or modeling would have been significantly more difficult, if not impossible, without an extensive analysis of the host architecture, the operating system, the network adapter and system overhead.

CHAPTER 7

ELIMINATING RECEIVE LIVELOCK

The previous three chapters presented the overall architecture of *END* and demonstrated how it could be used to model the data transmission and reception behavior of a real network adapter. It was also used to explore QoS issues in network transmission for different operating system configurations and network architectures. Since a host can completely regulate data transmission, but does not have similar control over incoming data, the issues surrounding data reception can be quite different from those pertaining to data transmission. This chapter shows how *END* may be used to study the impact of reception issues on network adapter design. In particular, it describes how *receive livelock* occurs under heavy network load, and the kind of modifications to the operating system and network interface that prevent its occurrence. Some of these solutions have been implemented and evaluated using *END*. We propose a novel scheme that makes simple modifications to the network adapter and its device driver and completely avoids receive livelock without requiring any changes in the operating system kernel. This solution is general enough to simultaneously handle multiple network interfaces and guarantee minimum bandwidths to each interface even under extreme overload conditions.

The rest of this chapter is organized as follows. Section 7.1 describes the receive livelock problem and how it affects system throughput. Section 7.2 shows how the host and adapter interrupt and scheduling policies may be modified to avoid receive livelock. It describes various solutions to receive livelock, and how they are implemented using *END*. Section 7.3 proposes an analytic framework to predict the efficacy of these solutions when there is one network interface, and presents experimental results that confirm the analysis. Section 7.4 discusses how these solutions would need to be modified when the host interfaces to multiple interrupt sources, and demonstrates how the adapter-based scheme remains effective without any modifications. Section 7.5 concludes this chapter.

7.1 Receive Livelock

A brief description of the receive livelock problem (summarized from [80,83]) is presented below. Packets received at a host must either be forwarded to other hosts (as in the case of a router), or to application programs where they are consumed. The delivered system throughput is a measure of the rate at which such packets are processed successfully. Figure 7.1 (adapted from [83]) demonstrates the possible behaviors of delivered throughput versus offered input load. Ideally, every packet received is processed, no matter what the packet arrival rate is. However, all practical systems have a finite capacity, and cannot receive and process packets beyond a maximum rate (determined by their processing capacity, and the cost of receiving and processing the packet), called the *Maximum Loss-Free Receive Rate* (*$\mathcal{MLFRR}$*) [83]. In poorly-designed communication subsystems, under network input overload, a host can be swamped with receiving arriving packets to the extent that the effective system throughput falls to zero. Such a situation, where a host has not crashed but is unable to perform useful functions such as processing received packets or running other ready processes, is known as *receive livelock*. Similarly, under receive livelock, a router may be unable to forward packets to the outgoing interface, resulting in transmit starvation [83].

The primary reason for receive livelock is that most traditional network adapters interrupt the attached host for every arriving packet, and these hardware interrupts are handled at a very high priority, e.g., higher than software interrupts (in BSD Unix-derived systems) or input threads that process the packet further up the protocol stack. At lower packet arrival rates, this design allows the host to consume a packet almost immediately on arrival, freeing up (limited) buffer space on the adapter for future packets. However, this implies that the host must (at least) accept and process all incoming packets, regardless of whether the host has sufficient processing capacity available to process them completely. As a consequence, the host is driven into saturation with no useful work being performed; the arriving packets get dropped due to queue overflows and other CPU-bound tasks do not get to run due to the high priority assigned to packet arrival interrupts.

The problem of avoiding receive livelock has been addressed at length in [80]. The

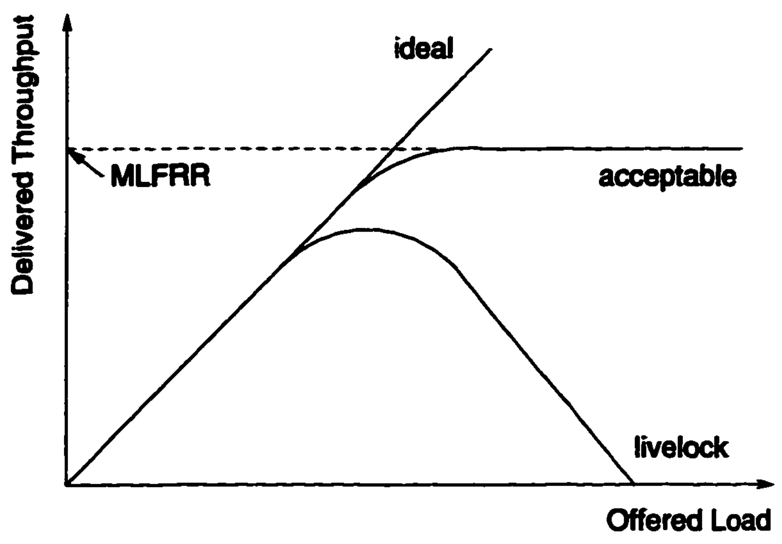


Figure 7.1: Range of possible behaviors of delivered throughput as a function of offered load.

authors' goals in designing packet reception mechanisms and policies were to guarantee acceptable ***system throughput***, reasonable ***latency*** and ***jitter***, ***fair*** allocation of resources, and overall system ***stability***. In particular, they used a combination of techniques to carefully schedule packet processing to avoid receive livelock and keep the system throughput near the $\mathcal{MLFRR}$ even under high loads. These techniques include limiting interrupt arrival rates to shed overload, polling the transmit and receive interfaces to provide fairness, processing received packets to completion, and explicitly regulating CPU usage for packet processing. These techniques constitute significant kernel enhancements and may also need to be customized based on the organization of the operating system, protocol stack and communicating applications. In addition to the goals presented above, it is also desirable that any techniques must be as ***general*** as possible and must ***minimize implementation complexity***.

A general solution is one that is applicable to a variety of operating systems and network interfaces, and must not depend on the target application. It must also be easily adaptable or parameterizable if the system configuration or goals are changed. For instance, the mechanisms and policies must be able to accommodate additional network interfaces, or optimize a particular aspect of performance (e.g., latency, jitter or throughput). Minimizing implementation complexity implies that modifications to the operating system must be minimized and localized and, as far as possible, applications should not be changed at all.

Further, reception mechanisms must be made as efficient as possible to lower the probability of livelock and to increase the $\mathcal{MLFRR}$.

Another solution to receive livelock is *lazy receiver processing (LRP)* [39], which has been implemented on Unix platforms with UDP/IP and TCP/IP based protocol stacks. As seen in the discussion above, receive livelock occurs since reception protocol processing is performed by the OS kernel at a very high priority. In the LRP paradigm, the network interface demultiplexes incoming packets to their destination socket queue (as opposed to a shared IP queue). In case of queue overflow, packets are simply discarded. Protocol processing for arriving packets is performed at the priority of the receiving application, rather than at interrupt priority. Now, the currently executing process is not interrupted, and protocol processing occurs only when the receiving application requests data and is scheduled to run. Since there are no unnecessary interrupts, and excess incoming data is discarded without consuming host CPU resources, receive livelock is avoided. LRP is also fair, since the CPU is allocated to various tasks based on the priorities computed by the scheduler. Note that this solution assumes that the network adapter has adequate intelligence to demultiplex packets directly to their destination sockets, i.e., it must have a general-purpose processor, and the adapter firmware must be designed to specifically interface with the host processor's protocol stack. If that is not possible, a similar, but less effective, solution maybe implemented in software. In this case, the packets are assigned to their socket queues in the receive interrupt service routine (ISR), and the rest of the protocol processing proceeds as before. This solution is not entirely free of receive livelock, since, under conditions of overload, the host must do some processing (handling the interrupt, and demultiplexing the packet) even if the packet is discarded in case of socket queue overflow. However, since the wasted CPU capacity is much less than in the case where the entire protocol processing is performed in the ISR, the probability of receive livelock is greatly reduced.

Note that prevention of receive livelock is facilitated by allowing the host to exercise control over the packet arrival interrupts. This can be done either by eliminating device interrupts entirely in favor of polling [99], or using a hybrid scheme [80] to limit the input arrival rate. Pure polling imposes significant CPU overhead and tends to exacerbate the average latencies seen by incoming packets, with the additional disadvantage that it is difficult to choose the proper polling frequency. In the hybrid scheme of [80], interrupts are used only to initiate polling and are enabled only when the polling thread has finished handling all the packets pending at an interface. Input handling is disabled (i.e., interrupts are turned off and the polling thread not run) if the total CPU cycles spent processing packets in the current polling period exceeds a certain threshold.

However, there are several problems with this hybrid scheme. Once the polling thread starts to run, it handles all packets pending at an interface (each time servicing only *quota* number of packets), until there are no pending packets or the time spent processing packets exceeds the threshold. Under a burst of incoming packets, it is likely that the polling thread may continue running until the capacity threshold is reached for that polling period. As the authors of [80] also point out, this introduces additional latency for applications that require received packets to be queued for processing by another thread. Further, there is also the question of selecting the polling timeout to re-enable interrupts on the attached interface.

The above problems can be solved by extending the host-adapter interface to allow the host to either directly specify the rate at which the adapter can generate interrupts (based on the CPU bandwidth the host is willing to allocate to receive packets), or provide the adapter with appropriate information so that it can adjust its interrupt rate to the host's load. In the following sections, we describe our implementations of variants of the schemes described in the literature [80,83,99]. We then propose and implement a novel adapter-based solution to avoid receive livelock and demonstrate how it achieves our goals of performance, generality and simplicity. We realize these extensions using *END*, and perform experiments to evaluate the relative efficacy of the various host- and adapter-based schemes. In addition to demonstrating the flexibility provided by *END* to explore design changes in a realistic setting, these experiments also illustrate that input traffic can be generated locally on *END* without requiring another machine and a sufficiently fast attached network.

7.2 Avoiding Receive Livelock

In this section, we present an overview of our implementation of the reception subsystem and describe the modifications to the host and adapter scheduling and interrupt policies required to avoid or reduce the effects of receive livelock.

Phase	Network Adapter		Device Driver
Initialization	store packet in buffer		
	initialize data structures		
	send read	$\longrightarrow$	
	interrupt host	$\longrightarrow$	initialize data structures
		$\longleftarrow$	complete host handshake
		$\longleftarrow$	send read ack
DMA		$\longleftarrow$	send address
	DMA packet to host	$\longrightarrow$	
	send read end	$\longrightarrow$	
	interrupt host	$\longrightarrow$	demux packet to protocol stack

Figure 7.2: Host–adapter interaction for packet reception.

7.2.1 Implementation Overview

The sequence of events following the arrival of a packet from the network are shown in Figure 7.2. The same sequence of events is captured in an *END* model shown in Figure 7.3. This is similar to, but simpler than, the CIM model described in Figures 5.2 and 5.4[1].

When a message arrives at the adapter, it buffers the packet, initializes some data structures, and notifies the host with a **read** command. The host then initializes its data structures and allocates buffers to receive the packet and responds with a **read ack** command. This completes the initialization handshake by giving the adapter the host's transaction identifier for this message. The host then sends an **address** command with the packets destination address. When the packet has been copied to the host, the adapter notifies it with a **read end** command. Each packet reception involves four commands – two from the adapter to the host (possibly with an interrupt with each command) and two from the host to the adapter.

Several changes were made to the host and to *END* to address the receive livelock problem. Though all the implementations and experiments were performed using *END*, in the rest of this discussion, we shall refer to adapters, unless we wish to highlight a feature of *END* that might not be necessary/possible to implement on a real adapter. The host-adapter device driver was extended to add a control interface (see Table 7.1). While the control interface uses the same command interface as the regular device commands (like **read**, **read ack**, etc.), these commands are not directly involved in receiving or transmitting data. All the commands are issued by the host to the adapter, and cause some action to

[1] The *END* model and device driver for this adapter were derived from the *END* model for the CIM.

$Stage_i$	Trigger	$Stage_i$_do	$Stage_i$_done
Initialization	message arrival from network	Initialize data structures send **read** interrupt host	
	read ack	complete host handshake	

DMA	**address**	enqueue packet for DMA
		`StartDmaXfer(){`
		dequeue segment;
		async delay for DMA}
			send **read end**
			interrupt host
			`StartDmaXfer()`

Figure 7.3: Outline of *END* model for reception for a generic adapter.

be performed by *END*.

The *Traffic Specification Commands* are used to make *END* create and destroy synthetic incoming traffic streams with specified parameters (distribution of arrival rate and packet sizes). These arrival patterns may either be computed at run-time using pseudo-random number generators, or by replaying pre-computed random processes. The advantage of using pre-computed arrival patterns is that it reduces the computational load on *END*. However, this approach may be limited by the availability of memory on *END* to store these traces, and also does not easily allow for random events that depend on system behavior that can the determined only at run time. When an incoming traffic stream is created, *END* associates with it a handler function that generates the specified type of message. It then passes on the arrival distribution parameters to the time device. The time device computes the packet arrival times, and notifies *END* of each "arrival". *END* then triggers the handler function which creates the incoming message, and then proceeds in a manner identical to the arrival of a real message. `CTRL_CREATE_PERIODIC` creates a periodic packet arrival process, and `CTRL_CLEAR_PERIODIC` terminates the process. The *Adapter Mode Control Commands* determine the interrupt policy across the host-adapter interface, i.e., when, if at all, the adapter may issue interrupts to the host. The various policies and their respective commands are discussed in detail below.

7.2.2 Host-based Policies

In host-based policies, the host operating system is modified to exert explicit control over the adapter interrupts and the host software is modified to carefully schedule the sequence of operations involved in network reception. The techniques presented here are partly derived from solutions proposed in [80,99].

Traffic Specification Commands	
CTRL_CREATE_PERIODIC	Create a periodic packet generation process
CTRL_CLEAR_PERIODIC	Terminate a periodic packet generation process
Adapter Mode Control Commands	
CTRL_MODE_POLL	Poll mode – adapter does not interrupt
CTRL_MODE_INTR	Interrupt mode (default)
CTRL_HOST_INTR_CTRL_ON	Host explicitly enables and disables adapter interrupts
CTRL_HOST_INTR_CTRL_OFF	Adapter decides when to interrupt (default)
CTRL_BACKOFF_ENABLE	Enable adapter interrupt backoff under overload
CTRL_BACKOFF_DISABLE	Disable adapter interrupt backoff

Table 7.1: Adapter control commands.

Interrupt-based Reception: This is the default operating system kernel mode where the adapter interrupts the host with every command (CTRL_MODE_INTR). These interrupts are high priority and preempt all other host processing. Clearly, in this mode, the kernel is susceptible to receive livelock. In the initial implementation, messages traversed the entire protocol stack in the `read end` interrupt service routine, and would get dropped if the receiving application had too much of a backlog and ran out of buffers, thereby wasting CPU capacity by processing messages that could not be consumed by the application. An *early drop* policy checks for availability of application buffers before accepting a packet. If a buffer is not available, it sends an error notification (ERR_NOBUFS) in the `read ack` command so that the adapter drops the packet without wasting any more host capacity.

Continuous Polling: In this scheme, the host instructs the adapter not to interrupt it (CTRL_MODE_POLL), and continuously polls the adapter interface for incoming packets. A variable, `poll_quota`, determines the maximum number of messages that may be accepted from the interface consecutively before the host starts processing the packets. While this scheme will perfectly balance the incoming packet rate and the packet processing rate, it is not a realistic option for most OSs since this implies that the OS knows all system activities *a priori* and statically schedules them. However, it is useful for comparative evaluation as a baseline strategy since it should provide optimal performance, and all other schemes may be compared with this to measure their efficacy.

Timed Polling: This is another polling scheme and the adapter does not interrupt the host (`CTRL_MODE_POLL`). However, the host does not poll the adapter interface continuously, but a polling process does so periodically, typically triggered by a clock interrupt. A timed polling scheme is specified by the period of the clock interrupt and the value of `poll_quota`. If the polling process executes for the entire period, it misses the next interval, thus ensuring that it cannot absorb all of the CPU capacity on an extended basis. It shares some of the advantages of continuous polling since it can regulate the maximum amount of time spent receiving messages. Further, it can use its normal scheduler to handle all other CPU tasks ensuring greater flexibility. However, selection of the period and poll quota can be difficult, and significantly impacts performance. A larger period can make the packet processing latency large, and making the period too small may impose an unacceptably high interrupt processing overhead. Further, the quota and period in effect fix the maximum CPU capacity that may be used for receiving packets. If this bound is too low, under low application load, and high data reception rates, the host will not be able to increase the reception rate even if there is spare CPU capacity.

Explicit Interrupt Management: In this scheme, the host explicitly controls when the adapter may interrupt it (`CTRL_HOST_INTR_CTRL_ON`). The adapter interrupts the host with its first command, and then disables the host interrupts. The host then polls the interface for up to `poll_quota` commands, and then allows the receiving application to execute. The host re-enables interrupts either after a delay of `Interval` milliseconds, or when the host becomes idle, whichever comes first. With a lightly-loaded system, this has the effect of enabling the interrupts as soon as possible, making it responsive. With a more heavily-loaded system, the delay is likely to complete before the host can re-enable interrupts, resulting in behavior similar to clocked interrupts.

7.2.3 Adapter-based Policies

Intelligent (programmable) adapters may also be used help avoid receive livelock. In LRP [39], the adapter could determine the destination of a packet, and since it had access to socket queues on the host, it could drop packets without interrupting the host. Similarly, in DEFTA [24], the device driver ensures that the packets are dropped in the adapter when the host is starved of resources to receive subsequent packets. The adapter may be tightly integrated with the host OS and protocol stack, as in LRP, but, in general, in contrast to host-based policies, it is possible to implement systems with little or no modification of the OS when the adapter is responsible for avoiding receive livelock. The host needs to provide some guidelines (either at initialization time, or during run time) limiting the maximum rate at which the adapter is permitted to interrupt the host. If the host limits the number of packets or commands processed in each ISR, it will bound the maximum CPU utilization due to packet reception, thereby ensuring that packet processing and other applications can progress. Conceptually, this is very similar to timed polling, but with the CPU utilization for packet reception limited by the adapter, rather than the host. If the adapter's maximum interrupt rate is fixed, it suffers the same shortcomings as timed polling, discussed in the previous section. We now propose an adaptive scheme that addresses these limitations while providing a solution that results in minimal changes to the OS.

Adaptive Backoff: In this scheme, the host provides some feedback to the adapter of its current load, and the adapter decides its interrupt rate based on this information. Under low loads, this scheme behaves like the normal interrupt-based adapter, i.e., it interrupts the host on every command. When the host indicates to the adapter that its load is increasing (and there is a backlog in the processing of incoming data or executing other applications), the adapter reduces its interrupt rate. As the load on the host decreases, the adapter increases its interrupt rate until it reverts to the normal interrupt mode. The CTRL_BACKOFF_ENABLE command has four parameters: *minimum backoff period, maximum backoff period, backoff factor* and *restore factor*. The first time the host indicates overload, the adapter waits for at least the *minimum backoff period* before interrupting again. If the adapter receives further indication of overload, it increases the backoff period by the *backoff factor*, a real number greater than 1. In no case does it allow the backoff factor to exceed the *maximum backoff period*. Each time the host indicates a lower load, the adapter reduces the backoff period by the *restore factor*, a real number less than 1. When the backoff period falls below the minimum backoff period, the adapter reverts to the normal interrupt mode. This scheme is similar to the algorithms for setting the window size for congestion control in TCP to the extent that it adapts the interrupt rate in response to overload. However, there are many differences, like the absence of acknowledgments and retransmissions and the fine grain of time, that make it quite distinct.

The parameters of CTRL_BACKOFF_ENABLE may be selected based on performance requirements. The minimum backoff period sets an upper bound on the CPU capacity used for handling receive interrupts from that interface. The maximum backoff period determines the worst case latency for the host to respond to the interface. The backoff and restore factors bias an interface towards the maximum and minimum backoff periods, respectively. A large backoff factor ensures that the interface sheds load rapidly due to overload, and a small restore factor ensures that the interface reduces latency quickly as the offered load declines. As we will show in Section 7.4, the maximum backoff period may be used in conjunction with the interrupt quota to divide the hosts capacity amongst multiple interfaces in a fair manner.

While the discussion above lays out the basic principles and mechanisms of the adaptive backoff scheme, it does not specify how "low load" or "overload" is defined. This can be done in various ways. In general, the load of the system may be measured in terms of the progress of the host's applications – if they do not receive enough of the CPU capacity to complete their tasks (including processing incoming packets), the host could be considered to be overloaded. Our measure of load was simply the buffer utilization – if the host runs out of buffers to receive packets, clearly, the receiving application is not progressing fast enough to process and free buffers. In our implementation, there are 128 receive buffers, and the host sends a backoff indication when the buffer availability drops below 25%. This value was chosen empirically by lowering the threshold until the host stopped dropping packets. This depends on the number of available buffers, and the maximum packet arrival and processing rate, and would need to be tuned for other platforms. The main requirements were that the host could absorb temporary bursts of packets and avoid wasting resources. If too few packets are permitted to be queued on the host, packets may be dropped unnecessarily on the adapter, and if too many packets are queued before making the adapter back off, there could be buffer overruns on the host. In addition, as soon as the host signals overload, it exits the receive ISR, even if there are pending commands, and allows the application to be scheduled.

7.3 Evaluation

We now propose an analytic framework to evaluate the various schemes proposed in the last section, and compare our experimental results with the analytic predictions. Experiments were performed for each of the configurations described in Section 7.2. *END* generates

C_{sw}	context switch time	20 μs
C_{cm}	cache miss penalty (worst case)	90 μs
C_{intr}	interrupt overhead	28 μs
C_{ri}	receive processing time	400 μs
C_{pp}	packet processing time "Load = 1" "Load = 1000"	 73 μs 347 μs
$\mathcal{A}$	packet arrival rate	variable
$\mathcal{P}$	packet processing rate	variable
$\mathcal{Q}$	maximum ISR quota	variable
$\mathcal{I}$	clock interrupt interval	variable

Table 7.2: Important system parameters.

packets arriving periodically at rates ranging from 1000 to 5500 packets per second. Each arriving packet is "processed" by a test application and then discarded. This application simply performs a dummy computation, either just increment the count of packets processed, or additionally, execute an empty loop with 1000 iterations. These applications are designated as "Load = 1" and "Load = 1000", respectively. Figure 7.4 shows the performance of both configurations, and all the remaining experiments were performed only with the application "Load = 1000". In each experiment, the number of packets processed is measured, as well as the mean and standard deviations of the delay from the time of arrival of the packet till it is processed and discarded.

7.3.1 Analysis of Receive Processing Mechanisms

Table 7.2 shows the measured values of important parameters of the reception subsystem on HARTS. When the host receives a packet, it takes C_{ri} time units to handle the reception protocol processing, and C_{pp} time units for the application to process the packet. In addition, depending on the scheme used, there could be other costs like the overhead for interrupts, context switches and cache misses. While parameters like C_{sw}, C_{cm}, C_{intr} are determined by the platform, C_{ri} is determined by the protocol stack and C_{pp} depends on the application processing the incoming data.

7.3.2 Interrupt-based Reception

As described in Section 7.2.1, each incoming packet generates two interrupts from the adapter. Ignoring cache behavior, the cost of processing each packet is approximately

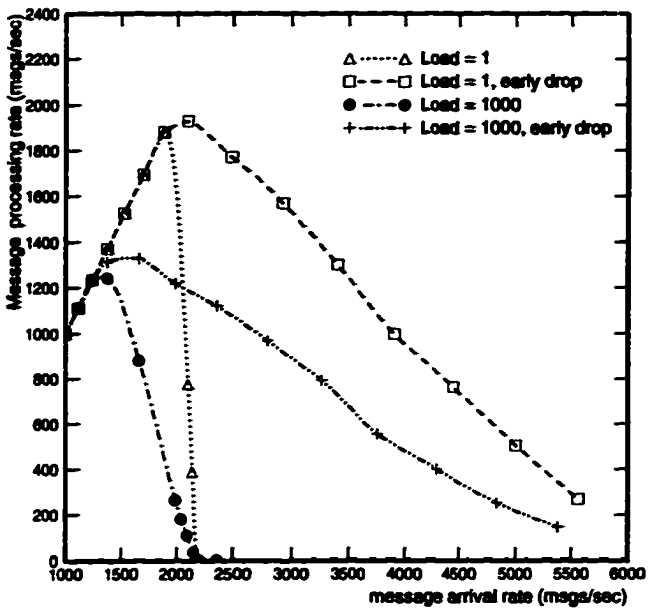


Figure 7.4: Interrupt based packet reception behavior.

$2 \times \mathcal{C}_{intr} + \mathcal{C}_{ri} + \mathcal{C}_{pp}$. Hence, the maximum rate at which the system can accept packets is given by:

$$\mathcal{MLFRR} = \frac{1}{2 \times \mathcal{C}_{intr} + \mathcal{C}_{ri} + \mathcal{C}_{pp}} \tag{7.1}$$

As the arrival rate of packets rises, more and more of the capacity is used simply to handle packet reception, and only the residual capacity is used to process packets, as shown below:

$$\mathcal{P} = \frac{1 - \mathcal{A} \times (2 \times \mathcal{C}_{intr} + \mathcal{C}_{ri})}{\mathcal{C}_{pp}} \tag{7.2}$$

Further, it may be noted that livelock will set in when packets can no longer be processed, i.e., when $\mathcal{P} = 0$, which happens when $\mathcal{A} = 1/(2 \times \mathcal{C}_{intr} + \mathcal{C}_{ri})$.

This behavior is shown in Figure 7.4. With loads 1 and 1000, the $\mathcal{MLFRR}$s are about 1881 and 1242 packets/second, respectively, and livelock sets in when the arrival rate exceeds 2183 packets/second. Note that at livelock, no packets are being processed at all, and hence, the $\mathcal{MLFRR}$ is independent of the value of $\mathcal{C}_{pp}$. With the *early drop* policy, if, on receiving a **read** command, the host has no buffers, it tells the adapter to drop the packet, avoiding the cost of the **read end** command, and of executing the receive protocol stack. While this helps delay the onset of livelock, it does not prevent it, and neither does

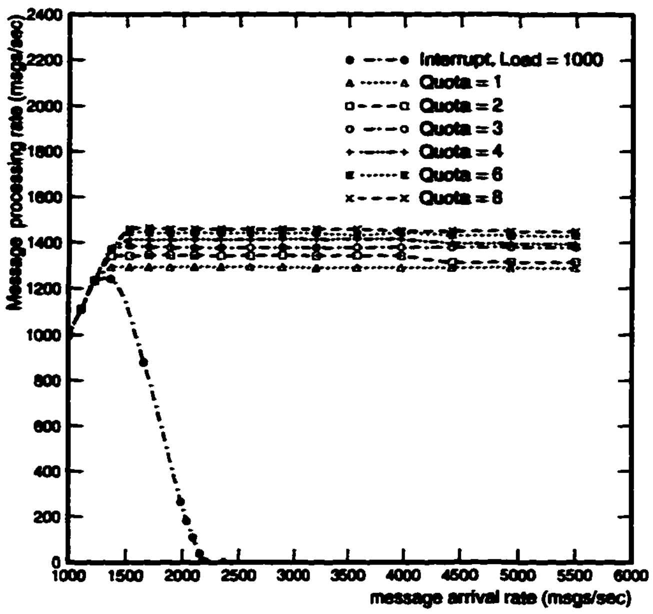


Figure 7.5: Continuous polling based packet reception behavior.

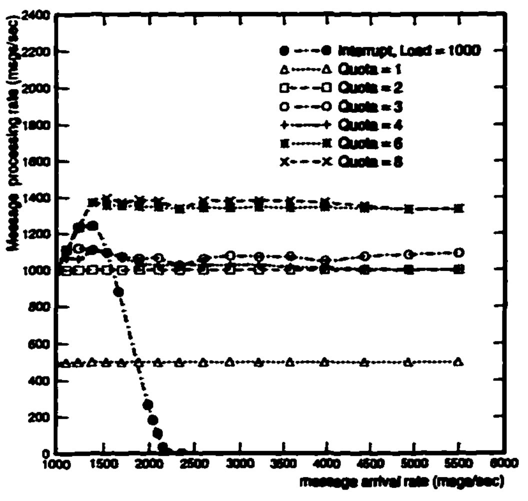
it raise the value of $\mathcal{MLFRR}$. This is similar to what was observed using the early discard policy in [39].

7.3.3 Continuous Polling

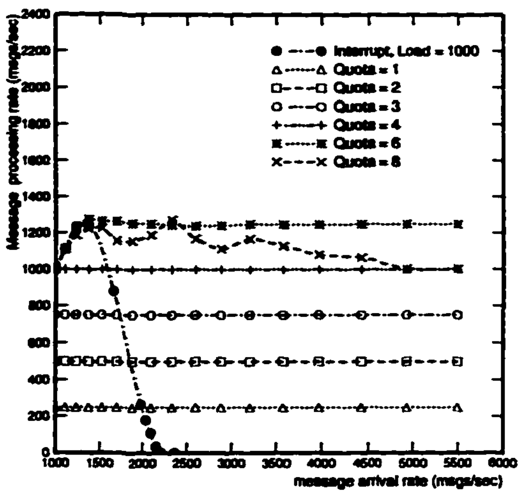
By polling the network interfaces (instead of using interrupts), the OS can explicitly schedule all activities (reception processing, packet processing and other applications), and thereby control their CPU utilization. It is also possible to trade off some capacity with delay jitter by varying the quota of the maximum number of adapter commands processed in each iteration ($\mathcal{Q}$). At $\mathcal{Q} = 1$, the delay per packet is very periodic, whereas increasing $\mathcal{Q}$ increases the system capacity by reducing the number of context switches, as well as potentially improving cache behavior, as estimated below:

$$\mathcal{MLFRR} = \frac{1}{C_{ri} + C_{pp} + C_{sw}/\mathcal{Q}} \tag{7.3}$$

This implies that $\mathcal{MLFRR}$ varies between 1303 and 1334 packets per second as $\mathcal{Q}$ is increased from 1 to 8. The experimental observations in Figure 7.5 show values of 1294 to 1447 packets per second. While the values agree very closely to the theoretical predictions at lower values of $\mathcal{Q}$, the observed behavior is much better for higher values. This is



(a) $\mathcal{I} = 1\ ms$



(b) $\mathcal{I} = 2\ ms$

Figure 7.6: Timed polling based packet reception behavior.

primarily because we do not estimate the increase in CPU capacity due to improvement in cache behavior. Further, as expected, continuous polling completely eliminates the receive livelock condition.

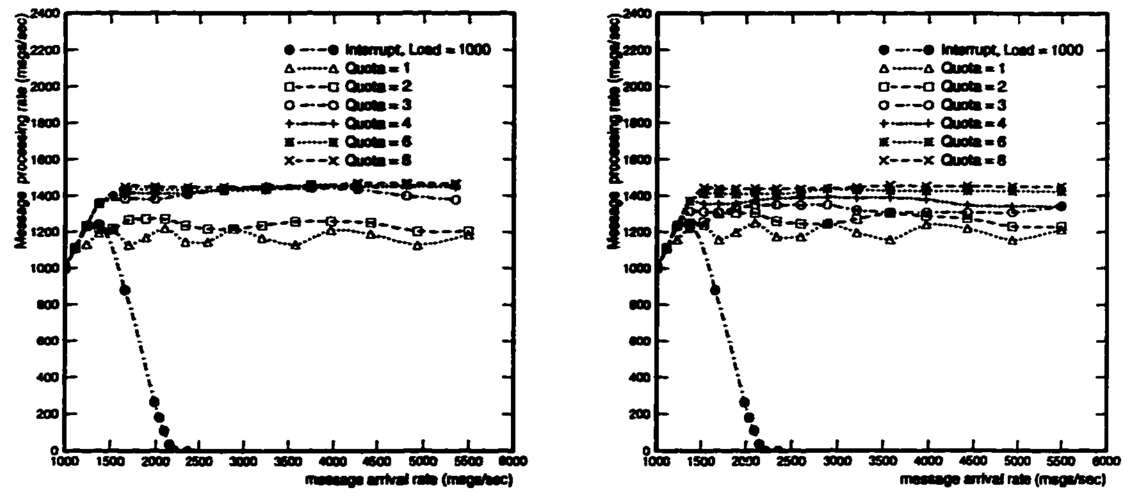
7.3.4 Timed Polling

In timed polling, a periodic timer interrupt every $\mathcal{I}$ time units triggers a polling process that handles up to $\mathcal{Q}$ commands. This represents the maximum capacity that may be used for reception processing in any such interval. Since this bounds the CPU capacity used by interrupts, it can also guarantee livelock-free behavior. Equation 7.4 below represents its behavior:

$$\mathcal{MLFRR} = \min\left(\frac{1}{\mathcal{I}} \times \frac{\mathcal{Q}}{2}, \frac{1 - \frac{\mathcal{C}_{intr} + \mathcal{C}_{sw}}{\mathcal{I}}}{\mathcal{C}_{ri} + \mathcal{C}_{pp}}\right) \tag{7.4}$$

In this scheme, each second there are $1/\mathcal{I}$ interrupts, and since each packet requires two commands, no more than $1/\mathcal{I} \times \mathcal{Q}/2$ packets may be received each second. Further, for each interrupt, capacity is used for the interrupt overhead and a context switch, leaving the residual capacity for reception and packet processing and other applications.

Figure 7.6 shows the timed polling packet-reception behavior for intervals of 1 *ms* and 2 *ms*, i.e., 1000 and 500 interrupts per second, respectively. At lower values of $\mathcal{Q}$ (1 and 2



(a) $\mathcal{I} = 1\ ms$ (b) $\mathcal{I} = 10\ ms$

Figure 7.7: Explicit interrupt management of packet reception behavior.

for $\mathcal{I} = 1\ ms$, and 1–5 for $\mathcal{I} = 2\ ms$), the CPU capacity is not fully utilized, and only a small number of packets can be processed (whatever the arrival rate) and the $\mathcal{MLFRR}$ is determined by the first term of Equation 7.4, as confirmed in the figure. For larger values of $\mathcal{Q}$, the second term should determine the $\mathcal{MLFRR}$, yielding values of about 1274 and 1306 packets per second for $\mathcal{I}$ of 1 ms and 2 ms, respectively. However, this is not exactly what we observe. One reason is that as $\mathcal{Q}$ increases, the CPU misses some of the timer interrupts and hence avoids some of the context switches and possibly also improves the cache behavior, resulting in better than expected throughput. However, when $\mathcal{I} = 1\ ms$ and $\mathcal{Q} = 4$, and when $\mathcal{I} = 2\ ms$ and $\mathcal{Q} = 8$, we see that throughput actually declines. This happens because the CPU frequently just misses the triggering of the timed polling process, and remains idle for the rest of the period, resulting in wasted capacity. This behavior is due to the relationship between the clock interrupt interval, the packet arrival rate and the cost of the various CPU operations, and in general, is hard to predict.

7.3.5 Explicit Interrupt Management

The explicit interrupt management is a hybrid scheme that adapts itself to the system load. It is similar to timed polling in that it processes $\mathcal{Q}$ commands, ensuring that other processes also get to run, and hence avoids receive livelock. However, when the CPU is otherwise idle,

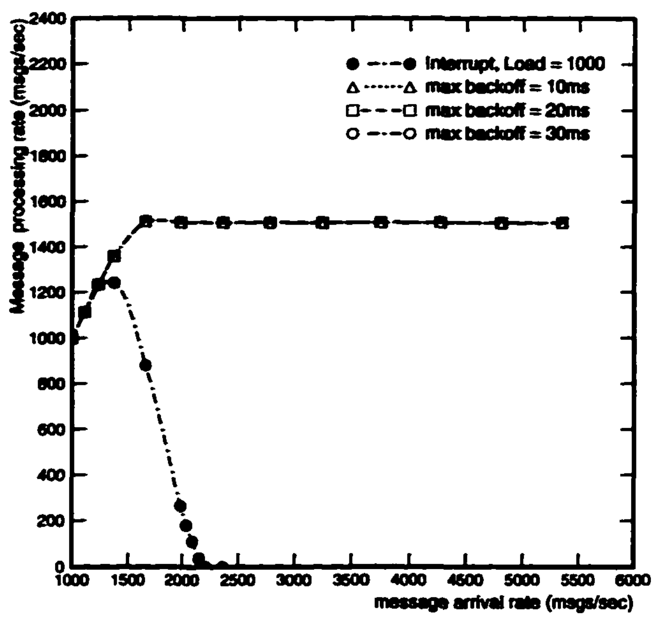


Figure 7.8: Adaptive backoff based packet reception behavior – backoff rate = 1.5, restore rate = 0.8.

it enables interrupts, and thus uses all available capacity. Due to its increased complexity, it can have slightly higher overhead, but, as shown in Figure 7.7, provides a fairly stable throughput. Moreover, it does not suffer low throughput on an otherwise idle system, as in the case of timed polling (Figure 7.6).

7.3.6 Adaptive Backoff

In adaptive backoff, the adapter continuously adapts its interrupt rate based on the hosts load indications. This ensures that it can keep the host continuously busy, and never overrun it. We performed experiments with a wide range of parameters, and found that the throughput always adapted to provide the $\mathcal{MLFRR}$ (Figure 7.8). In fact, the only time there was even a small amount of wasted capacity was when the maximum backoff was large (30 *ms*), and the backoff rate was large (2.0) and the recovery rate was slow (0.9), creating the situation where the host had drained its queue before the adapter's next interrupt. In addition, the overhead is relatively small, and the $\mathcal{MLFRR}$ of over 1500 packets per second was the highest we observed for any scheme.

7.3.7 Discussion

In Section 7.1, the goals of designing packet reception mechanisms and policies were identified to be to guarantee acceptable system throughput, reasonable latency and jitter, fair allocation of resources, and overall system stability [80]. Further, we required that these techniques be as general as possible and must minimize implementation complexity.

Host-based schemes are often very sensitive to parameter settings and the specific scheduling paradigm. Even with a single source of interrupts, they have to be debugged carefully and tuned for stability of performance. While this might be acceptable for special-purpose systems like the routers described in [80], a more general approach is desirable.

Adaptive backoff performs as well as, if not better than, the host-based solutions on all the performance criteria. While such a scheme does require some minor changes to the adapter firmware, changes to the host OS are minimal. All it requires is that the host have some measure of overload that it conveys to the adapter, and leaves the rest up to the adapter. This would typically involve minor modifications of the device driver, and the OS scheduler can remain unchanged.

Other than the purely interrupt-based OS, all the other schemes met the requirements of throughput, latency, jitter and stability. In fact, it is also possible to balance needs for throughput and jitter, by varying the polling or interrupt processing quota. Fairness will be discussed in Section 7.4, where we consider multiple network interfaces (or other sources of interrupts). The host-based solutions have the advantage of being purely software changes. The mechanisms described in Section 7.2.2 were implemented without directly modifying the OS scheduler, but by setting process priorities appropriately, and using cooperative preemption and controlling interrupts. This works on our *x*-kernel-based platform because it runs entirely in kernel mode, and the kernel threads are not preemptable. Further, all executing processes are known in advance, making it possible to schedule them in a controlled manner. Of these solutions, only timed polling is quite general, since all it requires is a periodic interrupt that triggers packet reception. The rest of the solutions are not particularly general. Each has to be hand-crafted to solve the problem at hand, and would need modifications based on any changes in the application mix.

A more general purpose OS (like Unix) would require complex changes to the OS scheduler, and other parts of the kernel, to meet the demands of the different executing tasks.

However, the adaptive backoff solution would work well on other platforms since it simply regulates the receive interrupt rate based on the progress of the processes that consume received packets, and consequently make available buffers to receive more data. In the next section, we demonstrate how adaptive backoff may also be used to regulate multiple network interfaces, and divide CPU resources fairly amongst them.

7.4 Multiple Interrupt Sources

Balancing the needs of multiple network interfaces further complicates implementation of livelock-free kernels. In such a case, not only must the kernel be livelock-free, but the host capacity must be divided amongst multiple network interfaces in a fair manner. Fairness implies that each interface is guaranteed a pre-specified, minimum portion of the host's capacity. In addition, it is desirable that any capacity that is not used by one interface is available to the other interfaces.

7.4.1 Host-based Solutions

Some of the host-based configurations for handling incoming data can be adapted to handle multiple network interfaces. Any interrupt-based kernel will continue to be susceptible to receive livelock. Explicit interrupt management is quite complicated with a single interface, and gets even more complicated as the number of interfaces increase. In general, it is impractical to turn interrupts on and off at the correct times when multiple interfaces are involved.

The two polling-based solutions can be used to share capacity amongst different interfaces, either by determining a polling sequence or by fixing the polling quota for each interface in each polling cycle. The polling sequence or the quotas for each interface determine how reception capacity is divided amongst the various sources. However, it is not easy to reallocate capacity from an idle interface to a busy one, since it would involve changing the polling sequence or polling quotas on the fly. If different processes handle data from different interfaces, the cost of rescheduling these processes may also be unacceptably high.

7.4.2 Adapter-based Solutions

Adaptive backoff requires little or no changes to handle multiple interfaces. Under conditions of overload, each interrupting source stays at its maximum backoff period, since the arrival rate is greater than the processing rate, causing buffer overflows and hence, adapter backoff. Note that each interface must have its own buffer pool, since, with a global buffer pool, a relatively idle interface may back off since a busy interface could occupy all available buffers. By setting a quota for the maximum number of packets the host may receive for each interrupt, the capacity an interface may use on the host CPU is bounded, allowing the OS to reserve any portion of its capacity for other applications. A minimum capacity may be guaranteed for each interface by setting its backoff parameters appropriately. The backoff parameters for different interfaces may be set independently, subject to total system capacity. In addition, due to the backoff/restore mechanism, the backoff period gets modified continuously, ensuring that a busy device can readily absorb the spare capacity of a temporarily idle device. Clearly, this is possible only if all the sources of interrupts comply with the backoff policy; since the host provides no explicit protection, even one misbehaving device can cause livelock.

Let each network interface be specified by the tuple $<\mathcal{I}_{min}, \mathcal{I}_{max}, b, r, \mathcal{Q}>$, where $\mathcal{I}_{min}$ and $\mathcal{I}_{max}$ are the minimum and maximum interrupt intervals, respectively, b is the backoff ratio, r is the restore ratio, and $\mathcal{Q}$ is the processing quota in each ISR. Even during overload, an interface interrupts the host at least once every $\mathcal{I}_{max}$ units, and can process up to $\mathcal{Q}$ interface commands in each ISR. Since two commands are required to receive a packet, it is guaranteed a packet reception rate of at least $\frac{\mathcal{Q}}{2 \times \mathcal{I}_{max}}$, as long as the total load offered to the host is less than its capacity, i.e., the condition specified in Equation 7.5 below is satisfied.

$$\mathcal{MLFRR} \geq \sum_{\text{all interfaces}} \frac{\mathcal{Q}}{2 \times \mathcal{I}_{max}} \qquad (7.5)$$

Note that this condition is independent of the values of b and r, since, during overload, in the steady state, each interface's interrupt interval will remain at $\mathcal{I}_{max}$. Large values of b ensure that the interrupt rate falls rapidly during overload, and large values of r ensure rapid restoration of the interrupt mode once the (transient) data burst ends.

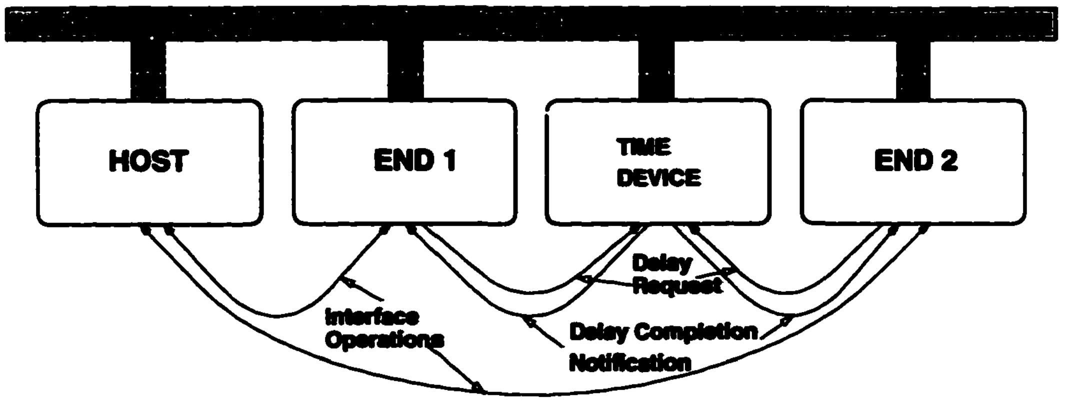


Figure 7.9: Emulator system configuration with one host and two network interfaces (VME StopWatch is not shown).

7.4.3 Experimental Evaluation

Figure 7.9 shows the emulator system configuration for experiments with two network interfaces. The *END* device driver on the host node was modified to interact with multiple interfaces. The *END* models only needed to be configured to interact with the appropriate host node, and did not require any other modifications. Each network interface is allocated 128 reception buffers, with the backoff threshold set at 25% of the capacity. Messages received from both interfaces are demultiplexed to the same application, which processes them in FIFO order.

As seen in Section 7.3.6, using adaptive backoff, the $\mathcal{MLFRR}$ is a little over 1500 packets/second. In each of the experiments below, *END* 2 generates traffic at a constant rate of 1000 packets/second, and *END* 1 gradually increases the rate at which it generates traffic, starting at about 300 packets/second. $\mathcal{I}_{min}$ and $\mathcal{I}_{max}$ are fixed at 1 *ms* and 20 *ms*, respectively, for both *END* nodes, and Q_1 and Q_2 are modified to change the reservation levels. The various experimental configurations are summarized in Table 7.3. In experiments 1–6, a total of 1500 messages/second are reserved, and in experiments 7–12, only 1100 packets/second are reserved, allowing the balance to be used by either interface. In Figures 7.10 and 7.11, the graphs to the left have $b = 2.00$ and $r = 0.99$, and the corresponding graphs to their right have the same load and quota parameters, but $b = 1.50$ and $r = 0.80$. Each data point represents 10,000 packet receptions.

Figure 7.10 represents experiments 1–6, where the entire reception capacity of the host (1500 packets/second) has been divided between the two interfaces. In each experiment,

Expt. Number	*END* 1			*END* 2		
	Quota (pkts/sec)	b	r	Quota (pkts/sec)	b	r
1	300	2.00	0.99	1200	2.00	0.99
2	700	2.00	0.99	800	2.00	0.99
3	1000	2.00	0.99	500	2.00	0.99
4	300	1.50	0.80	1200	1.50	0.80
5	700	1.50	0.80	800	1.50	0.80
6	1000	1.50	0.80	500	1.50	0.80
7	100	2.00	0.99	1000	2.00	0.99
8	500	2.00	0.99	600	2.00	0.99
9	800	2.00	0.99	300	2.00	0.99
10	100	1.50	0.80	1000	1.50	0.80
11	500	1.50	0.80	600	1.50	0.80
12	800	1.50	0.80	300	1.50	0.80

Table 7.3: Experimental configurations: $\mathcal{I}_{min} = 1\ ms$, $\mathcal{I}_{max} = 20\ ms$. Note that Quota = $\frac{Q}{2 \times \mathcal{I}_{max}}$.

since *END* 2 generates data at a constant rate of 1000 packets/second, and *END* 1 starts to generate data at 300 packets/second, initially, the offered load is less than the system capacity, and hence there is no backoff. As *END* 1's data rate increases, initially its throughput rises linearly, till the $\mathcal{MLFRR}$ is reached, and for some time beyond that as well. In fact, as can be seen in Figures 7.10(a,b) (Experiments 1 and 4), it takes away significant amounts of reserved capacity from *END* 2 for a small range of arrival rates. In Experiment 1, *END* 2 then recovers to 1000 packets/second, its packet generation rate (note that *END* 2's quota is greater than its arrival rate, and the unused capacity is used by *END* 1). This happens because, at low loads for *END* 1, its buffers do not fill up, and hence, it does not back off. At some point, the offered load from *END* 1 becomes high enough that it starts to back off, and then the capacity gets shared roughly as predicted. Since the backoff interval does not stay at $\mathcal{I}_{max}$ all the time, interfaces with lower reservations may get a larger share since their mean interrupt interval may be somewhat less than $\mathcal{I}_{max}$. This is particularly true when r is small (Figures 7.10(b,d,f)), since the interface rapidly reduces its interrupt interval. In fact, for small values of r, capacity tends to get divided roughly equally between the two interfaces. For $r \approx 1$ (Figures 7.10(a,c,e)), once an interface reaches $\mathcal{I}_{max}$, it stays close to that value until the offered load drops. In this case, capacity used by each interface is very close to the reserved value.

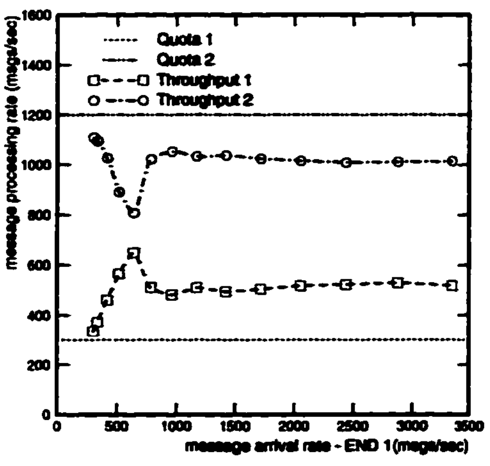
In contrast, Figure 7.11 represents experiments 7–12, where only part of the reception capacity of the host (1100 packets/second) has been reserved for the two interfaces. In most of these experiments, each interface gets its minimum reserved capacity (except when *END* 1 has not yet started to back off in experiments 7 and 10 (Figures 7.11(a,b)), for the same reasons as explained for experiments 1 and 4). As before, when $r \approx 1$ (Figures 7.11(a,c,e)), the excess capacity tends to be allocated roughly in proportion to the reservations of each interface. When r is small (Figures 7.11(b,d,f)), the total capacity tends to be divided roughly equally, typically, with a little more for the interface with the higher reservation.

7.4.4 Discussion

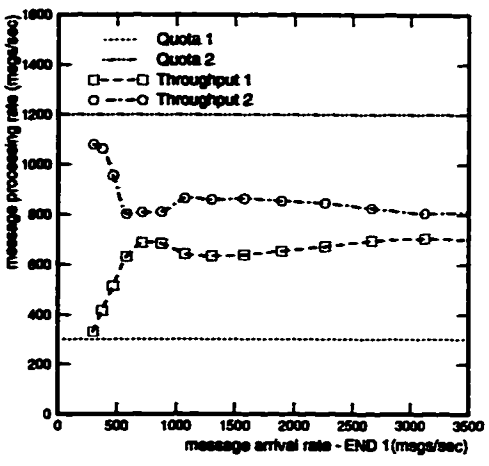
The experiments described above show that the host's capacity for receiving and processing network data can be divided amongst multiple interfaces in a pre-specified proportion while avoiding receive livelock. As in the case of a single interface, receive livelock was eliminated for a wide range of parameters without compromising throughput or flexibility. However, the exact proportion in which capacity was partitioned was very sensitive to various parameters, in particular, the restore factor[2], r.

Other factors would also affect the performance of this scheme. In this implementation, there is a single thread processing data from both interfaces, and so, any received data is handled at equal priority. However, if each interface (or even each network end-point (e.g., sockets)) had a distinct process associated with it, the load could be balanced better by appropriate CPU scheduling. Buffer allocation could also be used to control the interface. By allocating a fixed number of buffers for each interface based on its quota, and using a global buffer pool for unallocated capacity, it would be easier to ensure that one interface does not take more than its share of the CPU capacity. However, this also introduces new problems, like how buffers may be temporarily reallocated to a busy interface when another interface is idle. Analysis of these factors is, however, beyond the scope of the current work.

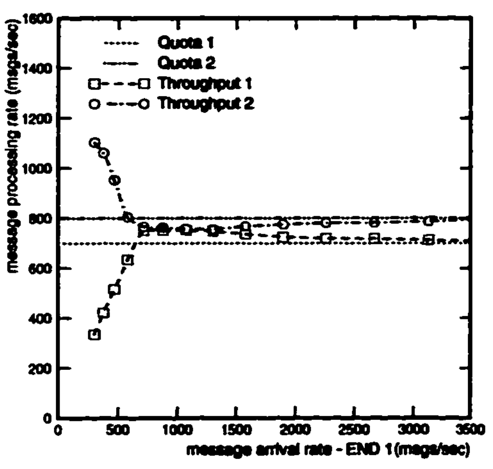
[2]Experiments modifying the backoff factor, b, showed that changing its value did not matter very much since the interface reaches its maximum backoff interval, $\mathcal{I}_{max}$, quite rapidly, even for fairly small values of b (unless $b \approx 1$).



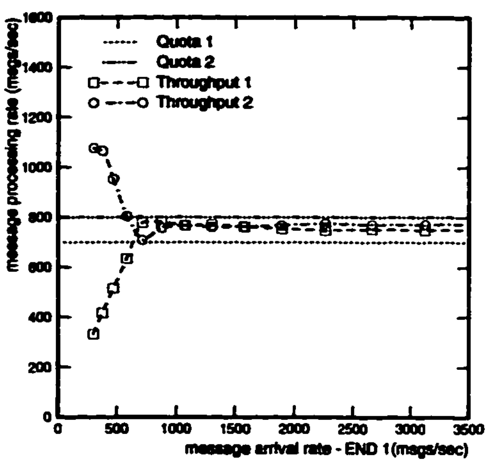
(a) Experiment 1



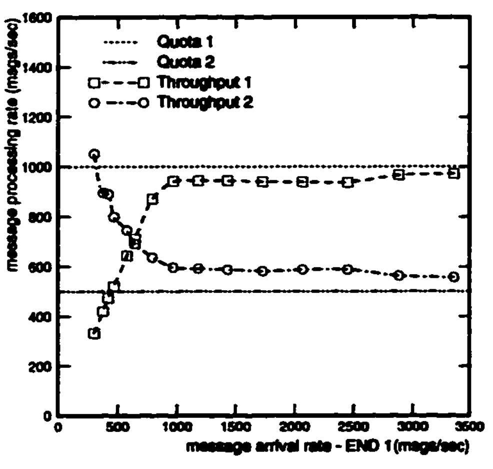
(b) Experiment 4



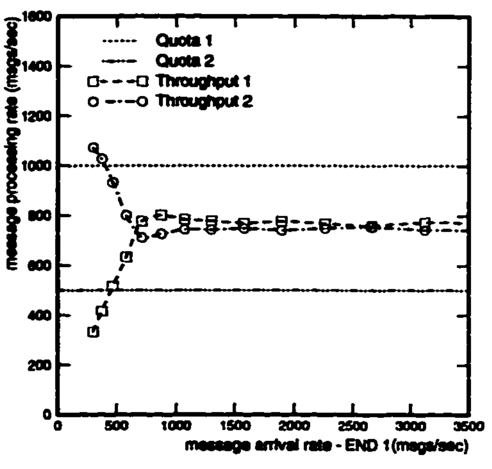
(c) Experiment 2



(d) Experiment 5

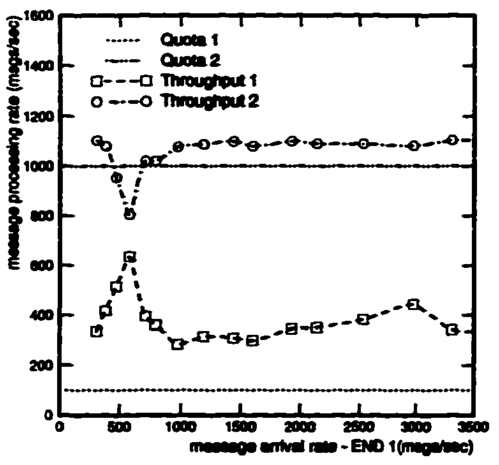


(e) Experiment 3

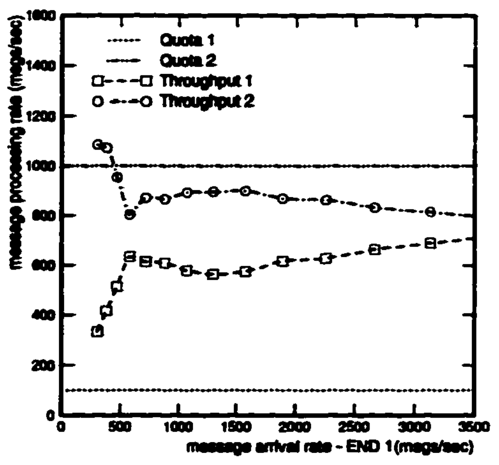


(f) Experiment 5

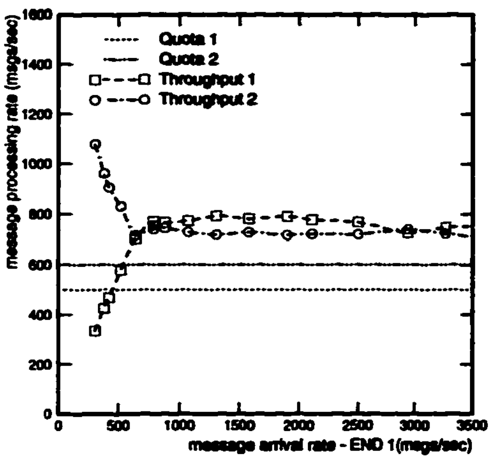
Figure 7.10: Adaptive backoff with two sources: Total reservation = 1500 packets/second.



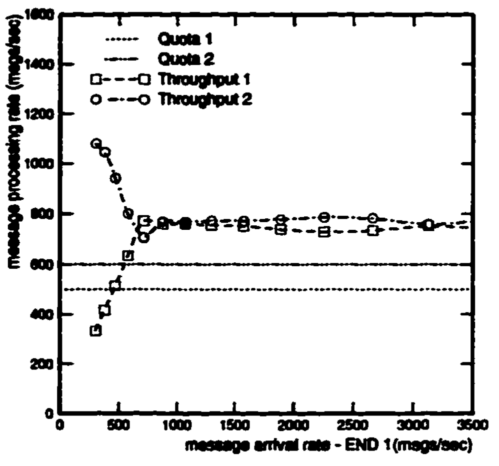
(a) Experiment 7



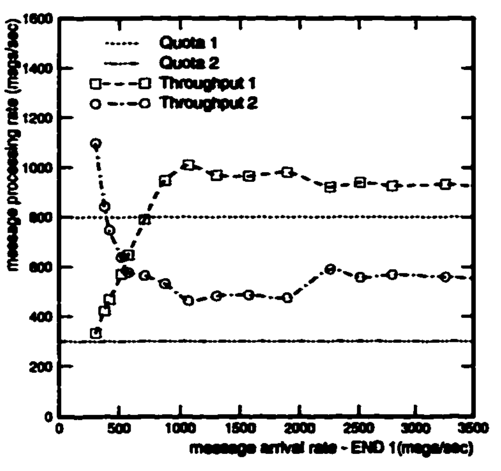
(b) Experiment 10



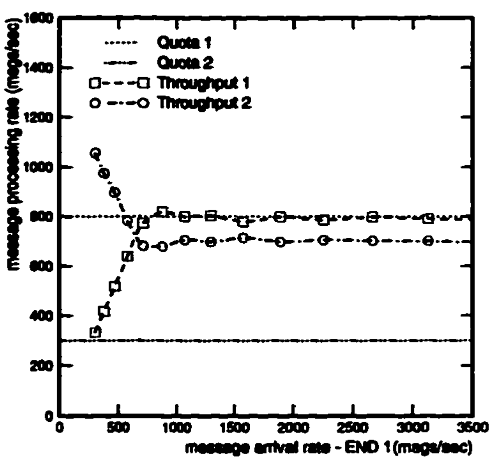
(c) Experiment 8



(d) Experiment 11



(e) Experiment 9



(f) Experiment 12

Figure 7.11: Adaptive backoff with two sources: Total reservation = 1100 packets/second.

7.5 Conclusions

This chapter studied the problem of receive livelock in interrupt driven operating systems. It examined the causes of this problem, and analyzed and implemented various solutions that eliminate receive livelock. Modifications to the host operating system include eliminating interrupts and using polling, or using hybrid solutions like polling triggered by periodic interrupts, or explicitly controlling when the adapter can interrupt the host. While these techniques do succeed in eliminating receive livelock, they require fairly complex modifications to the host operating system. In addition, they do not scale easily when multiple interrupting sources are interfaced to the host.

We proposed and implemented a novel network-adapter-based technique, called adaptive backoff, to avoid receive livelock. In this scheme, the host informs the adapter when its load rises above a specified threshold, and the adapter backs off and reduces the rate at which it interrupts the host. This scheme is very simple to implement since it does not require any changes in the host OS (except that the device driver has to indicate when it detects overload), and it performs as well as any of the host-based schemes. We also demonstrate how this scheme could easily be extended to handle multiple interrupting sources, and partition the host's capacity amongst the multiple sources in a fair manner. We also showed how the backoff parameters determine the share of capacity available to each interface, and how some of these parameters affect the fairness and stability of this scheme under different load conditions.

All the adapter models in this chapter were implemented using *END*, once again demonstrating its versatility in studying a wide variety of problems in communication subsystem design in general, and network adapter design in particular.

CHAPTER 8

CONCLUSIONS

Communication subsystems must be designed keeping in mind application requirements for QoS in end-to-end communication. It is therefore necessary to design the components of the communication subsystem such that the hardware and software components are integrated well. This minimizes the overhead of the interactions between the host and the network adapter, and between the network adapter and the network. Depending on the requirements and capabilities of the platform, support for QoS must be suitably divided between the host and the network adapter. Hardware/software codesign helps meet these goals by ensuring that hardware is designed keeping software requirements in mind, rather than paying later the cost of overcoming incompatibilities due to *ad hoc* design of hardware.

We now recapitulate the primary contributions of this dissertation, and suggest avenues for future research.

8.1 Research Contributions

This dissertation examined various issues involved in designing and implementing communication subsystems with QoS requirements. In particular, it focused on the design of network adapters, and their interface to end hosts. Network device emulation was proposed as a useful technique for hardware/software codesign. Our network device emulation tool, *END*, is easy to implement and use, and can be highly accurate. Major contributions were made in the following areas:

Implementation of real-time communication services: We described the architecture and implementation of a real-time communication subsystem, including the API, protocol architecture for connection management, and run-time support for real-time channels. This implementation used a commercial, best-effort network adapter to interface to a switched network fabric. We demonstrated how such a network adapter must be characterized to be able to adapt it for real-time communication, and explored the tradeoffs between best-effort throughput and real-time performance.

Network adapter design: We presented the architecture and implementation of *END*, a network adapter design tool. We examined how QoS is affected by the design of various components of network adapters, and how such components may be designed and evaluated using *END*. *END* is a highly versatile tool, and it can also be used to build arbitrary network adapter models interfacing to different types of networks. It may be used to evaluate the performance of communication subsystems, considering either transmission or reception in isolation, and also end-to-end, full-duplex communication. A key advantage of *END* is that it interfaces to real end hosts so that all experiments may be performed in real time, in a realistic environment, with real host software (applications, communication protocols, OS, etc.). However, this can limit *END* to only model systems that are similar to the implementation platform. Some of these restrictions were overcome by implementing techniques for modeling networks that are faster than the platform's communication medium and CPUs that are faster than *END*'s CPU.

Modeling network adapters using *END*: A case study was performed by using *END* to improve the design of a real network adapter, the Ancor VME CIM 250. We demonstrated how *END* was used to build a representative model of the adapter. This model captured both the functional interface of the adapter to the host, and also accurately modeled the interactions of various architectural components of the adapter. This model had throughput and latency behavior that was very similar to the real device. We identified some of the performance bottlenecks due to the design of this adapter, and modified the model to incorporate design improvements in the adapter architecture and host interface. For different configurations of the adapter (software modifications and/or additional hardware), and different traffic patterns, the throughput of the modified adapter increased by 15–50%. A designer may use these results to examine the price/performance tradeoffs of alternative designs even before building a prototype network adapter.

The device driver that interfaced to the *END* model of the CIM was almost identical to the device driver for the real CIM. We also claim that the structure of software in the *END* model of the CIM would be very similar to that of the real network adapter firmware. All these factors demonstrate not only that *END* may be used to design and evaluate network adapters before they are built, but also that a large part of the code used in building this model may be reused for the final product, thereby ensuring that debugged and tested code could be written even before the hardware was ready, minimizing wasted effort in the development of the *END* model, and speeding up the design and production cycle.

QoS issues in adapter design: *END* was used to build various models of network adapters interfacing either to a point-to-point network, or to a shared network medium. Various configurations of the host operating system, network adapter and networks were studied to determine how they affected QoS. We demonstrated that for any continuously available network medium, like in point-to-point networks, there is no need to build long packet queues on the adapter, and the host can schedule network transmissions to help provide QoS guarantees. On the other hand, in shared networks, the host has sporadic access to the network and in order to maximize utilization of the network bandwidth, it is necessary to buffer many packets on the network adapter, increasing the probability of priority inversion for FIFO devices. For different levels of traffic policing and shaping in the host OS, we studied various schemes to minimize the interference between the different traffic streams and evaluated how effective they were in providing QoS guarantees for soft real-time applications.

Receive livelock: Certain issues in network interface design for data reception were also studied. In particular, we addressed the problem of receive livelock and analyzed various host and adapter based techniques to eliminate it. We proposed a novel adapter-based solution, "adaptive backoff", and demonstrated that it was at least as effective as host-based solutions. In contrast to host-based solutions, adaptive backoff requires no changes to the host OS, other than minor modifications in the device driver. In addition, this solution is very general, and may be used in conjunction with any host scheduling policy. We demonstrated how it was used to partition the host resources amongst multiple network interfaces in a fair manner, and guaranteed each interface a minimum capacity during reception overload. When an interface is temporarily idle, the resources it has reserved are automatically made available to busy interfaces. Once again, all these studies were performed using *END*.

To summarize, in this dissertation, we addressed various issues in the design of communication subsystems for QoS. We not only proposed, implemented and analyzed specific solutions, but also proposed and implemented useful design techniques and tools. In particular, we designed and implemented a network adapter design tool that uses device emulation (*END*), and demonstrated the versatility of this tool by using it to (a) build a representative model of an existing network adapter and improve its performance, (b) to study OS and adapter based QoS for various network architectures, and (c) to study solutions to receive livelock.

8.2 Future Work

This dissertation presented techniques for designing host communication subsystems, as well as solutions to specific problems. We studied the design of communication subsystems and identified how QoS may be provided for transmission of data. These techniques can be extended to support QoS in data reception, including data-transfer optimizations, providing early demultiplexing of messages, and supporting application-specific network interfaces.

END is a very general and versatile tool with potential for improvement. At present, although *END* has a well-defined structure that can accommodate various device configurations and significant code segments can be re-used from one model to the next, each adapter model needs to be hand coded. It would be useful to develop a taxonomy for various components of the communication subsystem, and use this to develop a language and/or graphical interface. This may be used to configure network adapters, and build their *END* models using automatic code generation tools and standard libraries for adapter components, interrupt wrappers, schedulers, different types of queues, stochastic models for traffic generation, etc. *END* can be adapted to emulate wide-area networks, and to study the reliability of communication protocols by using it to inject faults into the network data stream. It may also be used to study other I/O devices like disks.

END has a well-defined, deterministic, interface with its network model. It could be extended to incorporate models of networks that use statistical multiplexing (e.g., Ethernet), and to capture the interactions between hosts connected by such a network. Such models may be used to study the efficacy of schemes to provide probabilistic QoS guarantees in these networks.

While *END* may be used to study various architectures for communication subsystems and predict their performance, this analysis should be combined with traditional hardware design methodology to determine the feasibility and cost of actually implementing such an architecture.

Adaptive backoff solved the receive livelock problem for multiple network interfaces. However, in some configurations, and for some load patterns, it failed to provide each network interface its reserved quota. It needs to be studied further to account for the effects of other system parameters, such as buffer management, on resource reservation.

While all the proposed solutions for receive livelock were effective to various degrees, they all simply discarded excess traffic. It is necessary to examine more sophisticated schemes that discard excess traffic selectively, based on the data types and priorities. One approach would be to attach tags to data packets that may be used to evaluate their relative importance, both within a particular data stream, and across multiple data streams. This would be necessary to provide QoS guarantees to selected high-priority applications even in the presence of network overload. These schemes can be made to provide guaranteed service to individual data streams by integrating the network interface management techniques with host process scheduling.

All the issues discussed so far was in the context of communication-intensive hosts like network servers. These paradigms should also be applied to other platforms, such as routers, that need to deliver high-performance, highly reliable, network services.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] 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.

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

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

[4] *CXT 250 16 Port Switch Installer's/User's Manual*, ANCOR Communications, Inc., 1993.

[5] D. P. Anderson, S. Y. Tzou, R. Wahbe, R. Govindan, and M. Andrews, "Support for continuous media in the DASH system," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 54–61, 1990.

[6] D. P. Anderson, "Metascheduling for continuous media," *ACM Trans. Computer Systems*, vol. 11, no. 3, pp. 226–252, August 1993.

[7] D. P. Anderson, L. Delgrossi, and R. G. Herrtwich, "Structure and scheduling in real-time protocol implementations," Technical Report TR–90–021, International Computer Science Institute, Berkeley, June 1990.

[8] D. P. Anderson, R. G. Herrtwich, and C. Schaefer, "SRP: A resource reservation protocol for guaranteed performance communication in the internet," Technical Report TR-90-006, International Computer Science Institute, Berkeley, February 1990.

[9] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne, "Real-time communication in packet-switched networks," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 122–139, January 1994.

[10] E. A. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste, "The design of nectar: A network backplane for heterogeneous multicomputers," in *Proc. Third Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 205–216, Boston, April 1989, ACM.

[11] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. C. Verma, and H. Zhang, "The Tenet real-time protocol suite: Design, implementation, and experiences," Technical Report TR-94-059, International Computer Science Institute, Berkeley, CA, November 1994.

[12] A. Banerjea, E. W. Knightly, F. L. Templin, and H. Zhang, "Experiments with the Tenet real-time protocol suite on the Sequoia 2000 wide area network," in *Proc. ACM Multimedia '94*, pp. 183–192, San Francisco, CA, October 1994. Also Tech. Rept. TR-94-020, International Computer Science Institute, Berkeley, CA, April 1994.

[13] T. Barzilai, D. Kandlur, A. Mehra, D. Saha, and S. Wise, "Design and implementation of an RSVP-based quality of service architecture for integrated services Internet," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 543–551, May 1997.

[14] A. Bas, V. Buch, W. Vogels, and T. von Eicken, "U-net: A user-level network interface for parallel and distributed computing," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 40–53, December 1995.

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

[16] R. Bettati and A. Gupta, "Dynamic resource migration for multiparty real-time communication," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 646–656, May 1996.

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

[18] G. Blair, A. Campbell, G. Coulson, F. Garcia, D. Hutchison, A. Scott, and D. Shepherd, "A network interface unit to support continuous media," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 264–275, February 1993.

[19] R. Braden, D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: An Overview.* Request for Comments RFC 1633, July 1994.

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

[21] K. Buchenrieder, "Hot topics: Hardware-software codesign: Codesign and concurrent engineering," *Computer*, vol. 26, no. 1, pp. 85–86, January 1993.

[22] 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.*, pp. 385–389, June 1995.

[23] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *IEEE Trans. Software Engineering*, vol. 21, no. 5, pp. 475–480, May 1995.

[24] C.-H. Chang, R. Flower, J. Forecast, H. Gray, W. R. Hawe, A. P. Nadkarni, K. K. Ramakrishnan, U. N. Shikarpur, and K. M. Wilde, "High-performance TCP/IP and UDP/IP networking in DEC OSF/1 for Alpha AXP," *Digital Technical Journal of Digital Equipment Corporation*, vol. 5, no. 1, pp. 44–61, Winter 1993.

[25] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *IEEE Micro*, vol. 14, no. 4, pp. 26–36, August 1994.

[26] C.-C. Chou and K. G. Shin, "Statistical real-time video channels over a multiaccess network," in *Proc. High-Speed Networking and Multimedia Computing Symposium,IS&T/SPIE Symposium on Electronic Imaging Science and Technology*, pp. 86–96, February 1994.

[27] C.-C. Chou and K. G. Shin, "A distributed table-driven route selection scheme for establishing real-time video channels," in *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pp. 52–59, Los Alamitos, CA, USA, May 30–June 2 1995, IEEE Computer Society Press.

[28] D. D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integrated services packet network: Architecture and mechanism," in *Proc. of ACM SIGCOMM*, pp. 14–26, 1992.

[29] E. C. Cooper, P. A. Steenkiste, R. D. Sansom, and B. D. Zill, "Protocol Implementation on the Nectar Communication Processor," in *SIGCOMM Symposium on Communications Architectures and Protocols*, pp. 135–144, Philadelphia, PA, September 1990, ACM.

[30] G. Coulson, A. Campbell, P. Robin, G. Blair, M. Papathomas, and D. Shepherd, "Design of a QoS controlled ATM based communications system in chorus," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 4, pp. 686–700, May 1995.

[31] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 4, no. 3, pp. 146–158, 1989.

[32] R. L. Cruz, *A Calculus for Network Delay and a Note on Topologies of Interconnection Networks*, PhD thesis, University of Illinois at Urbana-Champaign, July 1987. available as technical report UILU-ENG-87-2246.

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

[34] B. Davie, "The architecture and implementation of a high-speed host interface," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 228–239, February 1993.

[35] B. S. Davie, "A host-network interface architecture for atm," in *Proc. of ACM SIGCOMM*, pp. 307–315, September 1991.

[36] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *Proc. of ACM SIGCOMM*, pp. 3–12, 1989.

[37] Z. D. Dittia, J. R. Cox, Jr., and G. M. Parulkar, "Design of the APIC: A high performance ATM host-network interface chip," in *IEEE INFOCOM*, pp. 179–187, June 1995.

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

[39] P. Druschel and G. Banga, "Lazy receiver processing (LRP): A network subsystem architecture for server systems," in *Proc. Second USENIX Symp. on Operating Systems Design and Implementation*, pp. 261–276, October 1996.

[40] P. Druschel and L. L. Peterson, "Fbufs: A high-bandwidth cross-domain transfer facility," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 189–202, December 1993.

[41] 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.

[42] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton, "User-space protocols deliver high performance to applications on a low-cost Gb/s LAN," in *Proc. of ACM SIGCOMM*, pp. 14–23, London, UK, October 1994.

[43] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. SAC-8, no. 3, pp. 368–379, April 1990.

[44] D. W. Franke and M. K. Purvis, "Hardware/software codesign: A perspective," in *Proceedings of the 13th International Conference on Software Engineering*, pp. 344–352, May 1991.

[45] A. Gokhale and D. C. Schmidt, "Measuring the performance of communication middleware on high-speed networks," in *Proc. of ACM SIGCOMM*, pp. 306–317, August 1996.

[46] S. J. Golestani, "A self-clocked fair queueing scheme for broadband applications," in *IEEE INFOCOM*, pp. 636–646, June 1994.

[47] S. J. Golestani, "Congestion-free transmission of real-time traffic in packet networks," in *IEEE INFOCOM*, pp. 527–536. IEEE, June 1990.

[48] S. J. Golestani, "A stop-and-go queueing framework for congestion management," in *Proc. SIGCOMM Symposium*, pp. 8–18. ACM, September 1990.

[49] S. J. Golestani, "Congestion-free communication in high-speed packet networks," *IEEE Trans. Communications*, vol. 39, no. 12, pp. 1802–1812, December 1991.

[50] R. Gopalakrishnan and G. M. Parulkar, "Bringing real-time scheduling theory and practice closer for multimedia computing," in *Proceedings of ACM Sigmetrics 96*, pp. 1–12, May 1996.

[51] R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 68–80, 1991.

[52] A. Gupta, W. Howe, M. Moran, and Q. Nguyen, "Resource sharing for multi-party real-time communication," in *IEEE INFOCOM*, pp. 1230–1237, June 1995.

[53] S. Han and K. G. Shin, "A non-intrusive distributed monitoring support in fault injection experiments," in *IEEE International Workshop on Evaluation Techniques for Dependable Systems*, October 1995.

[54] M. Hemy and P. Steenkiste, "Gigabit I/O for distributed-memory systems: Architecture and applications," in *Proc. of Conf. on Supercomputing*, San Diego, CA, December 1995.

[55] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[56] 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, April 1995.

[57] D. B. Ingham and G. D. Parrington, "Delayline: A wide-area network emulation tool," *Computing Systems*, vol. 7, no. 3, pp. 313–332, Summer 1994. The USENIX Association.

[58] *pSOS$^+$/68K User's Manual*, Integrated Systems Inc., version 1.2 edition, September 1992. Document No. KX68K-MAN.

[59] T. B. Ismail and A. Amine Jerraya, "Synthesis steps and design models for codesign," *Computer*, vol. 28, no. 2, pp. 44–52, February 1995.

[60] C. R. Kalmanek, H. Kanakia, and S. Keshav, "Rate controlled servers for very high-speed networks," in *Proc. GLOBECOM*, pp. 12–20, December 1990.

[61] D. D. Kandlur, D. L. Kiskis, and K. G. Shin, "HARTOS: A distributed real-time operating system," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 3, pp. 72–89, July 1989.

[62] D. D. Kandlur and K. G. Shin, "Design of a communication subsystem for HARTS," Technical Report CSE-TR-109-91, CSE Division, Department of EECS, The University of Michigan, October 1991.

[63] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

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

[65] K. A. Kettler, D. I. Katcher, and J. K. Strosnider, "A modeling methodology for real-time/multimedia operating systems," in *Proc. of the Real-Time Technology and Applications Symposium*, pp. 15–26, May 1995.

[66] H. Kopetz and G. Grunsteidl, "Ttp - a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, January 1994.

[67] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf, "A framework for hardware/ software codesign," *Computer*, vol. 26, no. 12, pp. 39–45, December 1993.

[68] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable communication protocol processing in real-time mach," in *Proc. 2nd Real-Time Technology and Applications Symposium*, pp. 220–229, June 1996.

[69] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison Wesley, May 1989.

[70] 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 *Proc. of Conf. on Supercomputing*, pp. 174–183, November 1994.

[71] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, January 1973.

[72] C. Maeda and B. N. Bershad, "Protocol service decomposition for high-performance networking," in *Proc. ACM Symp. on Operating Systems Principles*, pp. 244–255, December 1993.

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

[74] A. Mehra, A. Indiresan, and K. Shin, "Resource management for real-time communication: Making theory meet practice," in *Proc. of 2nd Real-Time Technology and Applications Symposium*, pp. 130–138, June 1996.

[75] A. Mehra, A. Indiresan, and K. Shin, "Structuring communication software for quality-of-service guarantees," in *Proc. of 17th Real-Time Systems Symposium*, pp. 144–154, December 1996.

[76] H. E. Meleis and D. N. Serpanos, "Designing communication subsystems for high-speed networks," *IEEE Network*, pp. 40–46, July 1992.

[77] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proc. IEEE International Conference on Multimedia Computing and Systems*, pp. 90–99, May 1994.

[78] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," Computer Science Technical Report CMU–CS–93–157, Carnegie Mellon University, May 1993.

[79] C. W. Mercer, J. Zelenka, and R. Rajkumar, "On predictable operating system protocol processing," Technical Report CMU-CS-94-165, Carnegie Mellon University, May 1994.

[80] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," in *Winter USENIX Conference*, pp. 99–111, January 1996.

[81] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: The multiple node case," in *IEEE INFOCOM*, pp. 521–530, March 1993.

[82] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks – the single node case," in *IEEE INFOCOM*, pp. 915–924, 1992.

[83] 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.

[84] P. Ramanathan, D. D. Kandlur, and K. G. Shin, "Hardware-assisted software clock synchronization for homogeneous distributed systems," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 514–524, April 1990.

[85] L. Rizzo, "Dummynet: A simple approach to the evaluation of network protocols," *Computer Communication Review*, vol. 27, no. 1, pp. 31–41, January 1997.

[86] O. Rose, "Statistical properties of MPEG video traffic and their impact on traffic modeling in ATM systems," Institute of Computer Science Research Report Series 101, University of Wuerzburg, February 1995.

[87] M. A. R. Saghir, P. Chow, and C. G. Lee, "Exploiting dual data-memory banks in digital signal processors," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 234–243, Cambridge, Massachusetts, October 1–5, 1996.

[88] D. Saha, S. Mukherjee, and S. K. Tripathi, "Carry-over round robin: A simple cell scheduling mechanism for ATM networks," in *IEEE INFOCOM*, pp. 630–637, March 1996.

[89] 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.

[90] H. Schulzrinne, J. Kurose, and D. Towsley, "An evaluation of scheduling mechanisms for providing best-effort, real-time communication in wide-area networks," in *IEEE INFOCOM*, pp. 1352–1361, June 1994.

[91] K. G. Shin, D. D. Kandlur, D. L. Kiskis, P. S. Dodd, H. A. Rosenberg, and A. Indiresan, "A distributed real-time operating system," *IEEE Software*, pp. 58–68, September 1992.

[92] K. G. Shin, "HARTS: A distributed real-time architecture," *IEEE Computer*, vol. 24, no. 5, pp. 25–35, May 1991.

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

[94] P. Steenkiste, "Analyzing communication latency using the Nectar communication processor," in *Proc. of ACM SIGCOMM*, pp. 199–209. ACM, ACM, New York, NY, USA, August 1992.

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

[96] P. A. Subrahmanyam, "Hot topics: Hardware-software codesign: Cautious optimism for the future," *Computer*, vol. 26, no. 1, pp. 84, January 1993.

[97] L. Trajkovic and S. J. Golestani, "Congestion Control for Multimedia Services," *IEEE Network*, pp. 20–26, Sept. 1992.

[98] B. Traw and J. Smith, "A high performance host interface for ATM networks," in *Proc. of ACM SIGCOMM*, pp. 317–325, September 1991.

[99] C. B. S. Traw and J. M. Smith, "Hardware/software organization of a high-performance ATM host interface," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 240–253, February 1993.

[100] C. Venkatramani and T. Chiueh, "Design, implementation, and evaluation of a software-based real-time ethernet protocol," in *Proceedings of the ACM SIGCOMM*, pp. 27–37, August 1995.

[101] A. S. Wenban, J. W. O'Leary, and G. M. Brown, "Codesign of communication protocols," *Computer*, vol. 26, no. 12, pp. 46–52, December 1993.

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

[103] 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.

[104] G. G. Xie and S. S. Lam, "Delay guarantee of a virtual clock server," *IEEE/ACM Trans. on Networking*, pp. 683–689, December 1995.

[105] H. Zhang and D. Ferrari, "Rate-controlled static-priority queueing," in *IEEE INFOCOM*, pp. 227–236, June 1993.

[106] L. Zhang, "VirtualClock: A new traffic control algorithm for packet switching networks," in *Proceedings of the SIGCOMM Symposium*, pp. 19–29. ACM, September 1990.

[107] L. Zhang, "Virtual Clock: A new traffic control algorithm for packet-switched networks," *ACM Trans. Computer Systems*, vol. 9, no. 2, pp. 101–124, May 1991.

[108] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource ReSerVation Protocol," *IEEE Network Magazine*, pp. 8–18, September 1993.

[109] Q. Zheng and K. G. Shin, "Real–time communication in local area ring networks," in *Conference on Local Computer Networks*, pp. 416–425, September 1992.

[110] Q. Zheng and K. G. Shin, "Synchronous bandwidth allocation in FDDI networks," in *Computer Graphics (Multimedia '93 Proceedings)*, pp. 31–38. ACM, Addison-Wesley, August 1993.

[111] Q. Zheng and K. G. Shin, "On the ability of establishing real-time channels in point-to-point packet-switched networks," *IEEE Trans. Communications*, vol. 42, no. 2/3/4, pp. 1096–1105, February/March/April 1994.