Design Tradeoffs in Implementing Real-Time Channels on Bus-Based Multiprocessor Hosts

Atri Indiresan Ashish Mehra
Kang G. Shin

Real-time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122
{*atri,ashish,kgshin*}@*eecs.umich.edu*

Abstract

There are a growing number of real-time applications (e.g., real-time controls, and audio/video conferencing) that require certain quality-of-service (QoS) from the underlying communication subsystem. The communication subsystem must support real-time communication services that can be used to provide the required QoS of these applications, while providing reasonably good performance for best-effort traffic. In this paper we explore the design tradeoffs involved in supporting real-time communication on bus-based multiprocessor hosts using standard network hardware. These tradeoffs are examined by implementing the concept of *real-time channel*, a paradigm for real-time communication services in packet-switched networks.

We first present a hardware and software architecture facilitating real-time communication on bus-based multiprocessor hosts. The main features of this architecture include a dedicated protocol processor, a split-architecture for accessing real-time communication services, and decoupling of data transfer and control in the communication protocol stack. The implications of network adapter characteristics for real-time communication are considered and desirable adapter features derived. Techniques to circumvent limitations in adapters not providing explicit support for real-time communication are presented. Support for real-time communication necessitates 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 real-time channels. We present data transfer optimizations, mechanisms to schedule protocol processing, and link scheduling mechanisms that together achieve this goal. The effectiveness of our real-time channel implementation is demonstrated through experiments while varying traffic characteristics.

Key Words — Real-time communication, bus-based multiprocessors, network adapter, protocol processing, CPU and link scheduling

The work reported in this paper was supported in part by the National Science Foundation under grant MIP–9203895 and the Office of Naval Research under grant N00014–94–1–0229. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or ONR.

1 Introduction

Emerging distributed applications such as medical imaging, distributed monitoring and control, and audio/video conferencing, critically depend on the delivered performance or quality-of-service (QoS) of the underlying communication subsystem. In particular, they require certain guarantees on performance parameters such as end-to-end delay, delay jitter, available bandwidth, and packet loss. For example, distributed real-time systems must provide predictable end-to-end message delay, since unpredictable message latency can adversely affect the timely execution of communicating tasks. The communication subsystem must provide a set of real-time communication services that can satisfy the QoS requirements of these applications. Besides guaranteeing that the QoS requirements of applications are met, these services should allow best-effort (non-real-time) traffic to coexist with real-time traffic. Though no guarantees are made about the performance delivered to best-effort traffic, it is desirable to avoid unduly penalizing its performance in the presence of real-time traffic.

The *real-time channel* model [1,2] 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 real-time channel 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. A real-time channel provides delivery-delay guarantees for real-time traffic, and at the same time, allows reasonably good performance for best-effort traffic.

This paper examines design tradeoffs involved in implementing real-time channels in (bus-based) multiprocessor hosts using standard adapter hardware for access to the network. Small-scale bus-based multiprocessor configurations are increasingly being employed in multimedia servers and workstations (e.g., SGI Challenge XL, Sun SPARCserver 1000). These end hosts typically have a small number of processors which can individually access the network adapter across the main system bus. With multiple independent real-time channels emanating from each processor, it becomes important to minimize the interference between the traffic on different real-time channels, and to distinguish real-time traffic from best-effort traffic. This necessitates that system resources such as bus 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 real-time channels.

First, we present a hardware and software architecture that realizes this goal 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 in order to offload all protocol processing from the application processors, freeing them from adapter handshake overheads 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. Finally, 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.

Second, we explore three aspects pertaining to the performance of real-time and best-effort traffic on our hardware and software architecture. For this we use a VME bus-based multiprocessor as the end host, with hosts connected by a network constructed from the Ancor CXT 250 crossbar switch [3] and CIM 250 network adapters [4]. 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 the network adapter that support real-time communication, and hence the implementation of real-time channels. For example, FIFO queueing in the network adapter exacerbates the (communication) medium access latency while making it more unpredictable, and hence must be controlled. Other important features include allowing the link scheduler to exercise fine-grain control over packet transmissions, and the ability to efficiently examine packet headers and selectively discard received packets. Next, we consider the software overheads of protocol processing and link scheduling on the (dedicated) protocol processor. With no copying of data during protocol processing, protocol-processing and link-scheduling overheads are directly proportional to the number of fragments and the per-fragment processing cost. Using the overheads as motivation, we propose CPU scheduling mechanisms to preserve QoS guarantees to various 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 paper is organized as follows. Section 2 describes the hardware and software organization of our experimentation platform, while Section 3 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 4, which also highlights desirable features in the network adapter that facilitate real-time communication. Section 5 first 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 overheads of our implementation are presented next, along with CPU scheduling mechanisms to preserve QoS guarantees. Section 6 evaluates the implementation of the link scheduler controlling access to the network. The evaluation focuses on the effectiveness with which the scheduler insulates real-time traffic from best-effort traffic on the one hand, and traffic belonging to different real-time channels on the other. Section 7 discusses other proposals for providing real-time communication services, and compares our real-time channels service to other related approaches. We conclude in Section 8 with an analysis of the results and suggest directions for future work.

2 The Experimentation Platform

In this section we describe the hardware and software architecture of our experimentation platform, which is being developed as a part of HARTS [5]. The primary goal of HARTS is to investigate architectural and operating system issues in distributed real-time computing.

2.1 Hardware

Each HARTS node (also referred to as end host) is a VME bus-based multiprocessor with 2–4 processors, as shown in Figure 1. This multiprocessor configuration provides several benefits over uniprocessor configurations. Many input/output devices and controllers are available for the popular VME bus. Each processor can be dedicated to control a different device on the VME bus, 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.

We divide the available processors in each HARTS node into Application Processors (APs) and a Network Processor (NP); applications execute on APs while communication protocols execute on the NP. Dedicating a processor to control the VME bus-based communication devices has several advantages: the NP offloads all communication processing from the APs, frees the APs from device handshake overheads, and permits greater overlap between useful computation and communication processing[1]. Since the communication device is located on the VME bus, 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

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

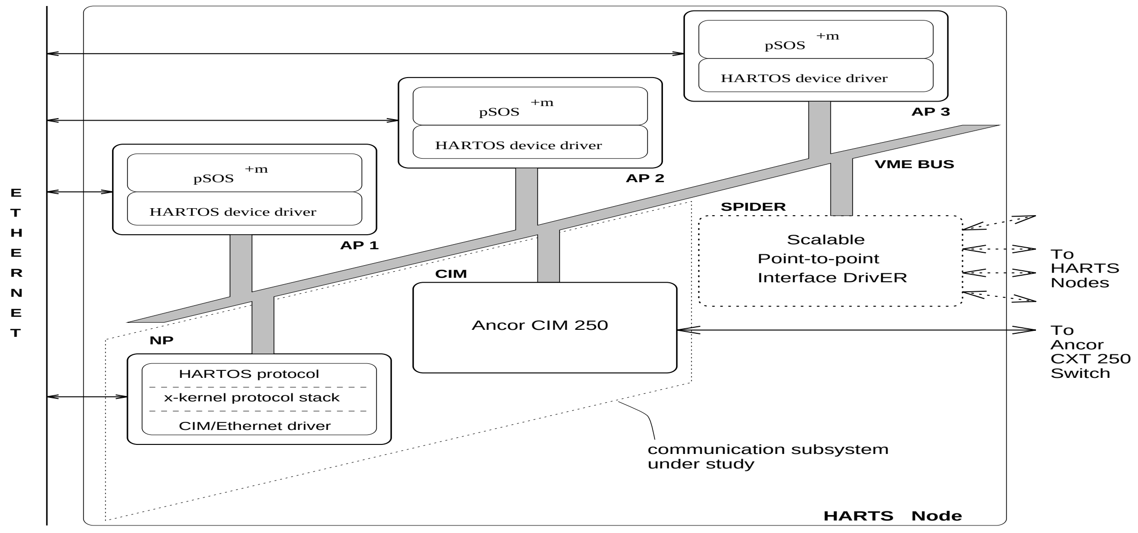


Figure 1: Architecture of each node/host

by their traffic parameters. Each processor is an Ironics IV-3207 card with a Motorola MC68040 CPU; the NP has 16 MB of DRAM while the APs have 4 MB of DRAM.

In the current configuration, the HARTS interconnection network is constructed from the Ancor CXT 250 crossbar switch [3] and Ancor VME CIM 250 network adaptors [4], which implement the ANSI Fibre Channel 3.0 standard [6]. We use the CXT 250 crossbar switch to embed various partially-connected point-to-point topologies for studying multi-hop communication. In addition to communication interface hardware, the CIM has 8MB DRAM, independent DMA controllers for data movement, and an input/output processor that provides support for Fibre Channel operations. Though the CIM has a general-purpose I/O processor, the on-board firmware is controlled by the manufacturer; the NP exercises control over the CIM only through command/response FIFOs. The native interconnection network between HARTS nodes will be provided by SPIDER [7], a custom VME bus-based network adapter now in the final stages of development. SPIDER would allow construction of partially-connected point-to-point topologies with node degrees up to 4.

2.2 Software

HARTOS [8,9], the operating system running at 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 1 highlights the main HARTOS components. The APs run the pSOS^{+m} kernel [10] while the NP runs a protocol stack based on the x-kernel [11]. 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. We could have implemented our real-time protocols in pSOS^{+m} using pSOS^{+m} sockets as the real-time channel API, similar to the approach adopted by the Tenet group [12]. However, we would then be forced to limit ourselves to uniprocessor configurations, with the accompanying process scheduling interference effects, and the semantics and associated overheads 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/writes a command/response mailbox interface (HARTOS API) for services provided on the NP, such as real-time communication and distributed name service. The HARTOS API is split between

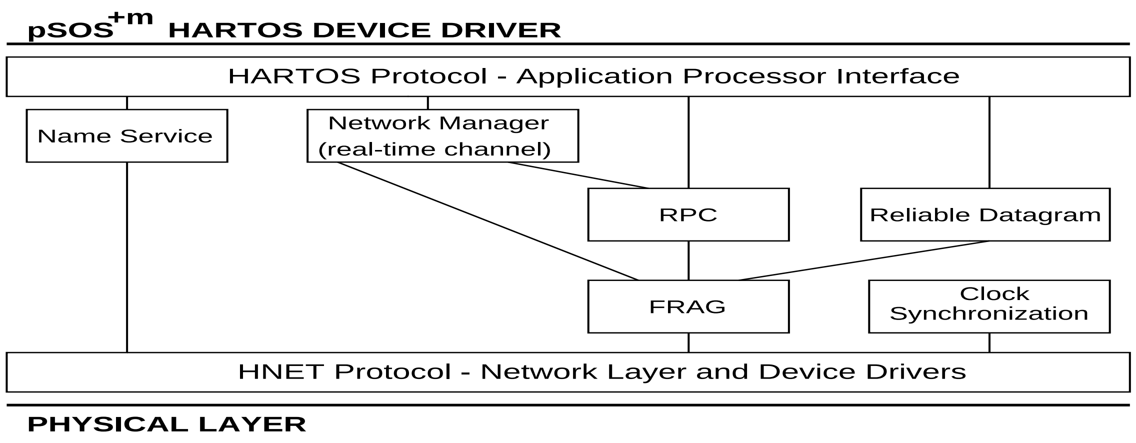

pSOS+m HARTOS DEVICE DRIVER
HARTOS Protocol - Application Processor Interface
Name Service
Network Manager (real-time channel)
RPC
Reliable Datagram
FRAG
Clock Synchronization
HNET Protocol - Network Layer and Device Drivers
PHYSICAL LAYER

Figure 2: The *x*-kernel protocol stack in HARTOS.

the APs and the NP, with API stubs marshalling 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 [11] as the (communication) executive. It employs a *process-per-message* 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 [13]. 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 4.3 [14]; this improves the traffic insulation between different real-time channels.

Figure 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 look up a name globally. Communication protocols include standard support like remote procedure call (RPC), reliable datagrams and fragmentation. The RPC and fragmentation protocols are modified versions of *x*-kernel's CHAN and BLAST protocols, respectively. Our implementation of the Clock Synchronization protocol uses Cristian's probabilistic clock synchronization algorithm [15], 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 or SPIDER-based network) and implements run-time packet scheduling for network access.

2.3 Baseline System Measurements

Table 1 lists the baseline measurements on the NP, the VME bus, and the CIM. The memory bandwidth was measured using the `bcopy`[2] operation, both within the NP and across the VME bus. The CIM has been shown to deliver a maximum throughput of 6 MB/second only for very large ($\approx$ 3 MB) DMA transfers [16]; while the throughput we obtained for smaller packet sizes ($\leq$ 16 KB) was similar to that obtained in [16], we could only obtain a data transfer bandwidth of $\approx$ 3 MB/second with 3 MB DMA transfers.

[2]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 overheads.

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 1: Baseline system performance

3 Implementation of Real-Time Channels

In this section we describe the implementation of real-time channels in HARTOS. First, we describe the application programming interface, next the required support for channel establishment, then the transfer of data from the sending AP to the destination AP through the intermediate nodes. We discuss the optimizations applied to the transmission and reception paths through the x-kernel to minimize/eliminate data copies. Finally, we present the implementation of the link scheduler that coordinates packet transmission/reception and traffic enforcement at each node.

3.1 Real-Time Channel API

Applications create and use real-time channels through the real-time channel API, as shown in Table 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 in 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* [17, 18], 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*, $\ell(m)$, for a message m as:

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

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, $\ell(m)$, is the time m would have arrived if the maximum message rate constraint was 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 truly non-blocking. The sending task can tear down the real-time

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 2: Routines constituting the real-time channel API

channel by invoking `rtc_close` with the local channel identifier; all the resources allocated to the channel are released at this point. Our implementation splits the real-time channel API between the APs and the NP; `rtc_init` and `rtc_recv` execute entirely on the AP running 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 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.

Our implementation uses a distributed network manager comprising network manager protocols (NMPs) 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 NMP, which must now determine if the requested channel can be established or not. If so, NMP selects a route and reserves sufficient resources along the route to accommodate the channel's timing constraints. NMP uses the underlying RPC protocol (see Figure 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 3.4). Each channel in the list stores a unique network-wide channel identifier (a `<node_id, local_channel_id>` tuple), traffic specification from the establishment 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. Figure 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 in [2]). We only consider 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.

1. Select the next link along the source–destination route.
2. Use algorithm `D_Order` [2] 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: Channel establishment procedure – forward phase

3.3 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 acknowledgements. After the initial marshalling 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 later transmission by the link scheduler. We have modified the message manipulation routines in x-kernel to associate the original message deadline with each packet, enabling the link scheduler to correctly order the transmission of packets into 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.

3.4 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 can 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 to meet their deadline as long as they don't violate their input specification, (b) *perform traffic policing*, i.e., prevent channels that violate their traffic specifications from affecting the performance of other 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 transmision 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[3]. 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 must be 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, namely, Queue 1, Queue 2, and Queue 3, in which outbound packets are inserted by protocol threads [2]. Queue 1 contains *current* real-time packets (whose logical arrival time is less than the current clock time), while 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. Best-effort packets are inserted in Queue 2; 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 signalling the `scheduler`. The scheduler semaphore's count is incremented by one each time a packet is 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 they will be buffered longer at downstream nodes, or simply dropped.

On waking up, 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 concluded in Section 4.1. The `scheduler` blocks again if there are two 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, 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.

4 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 significant impact on supporting real-time communication. An adapter's interface characteristics are determined partially 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

[3]Since we have 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.

adapter design features and interface characteristics on data transfer performance, medium access latency, and packet handling on reception. Based on these insights, we highlight the desirable design features (hence provided in SPIDER) to support real-time communication.

4.1 CIM Performance Characteristics and Implications

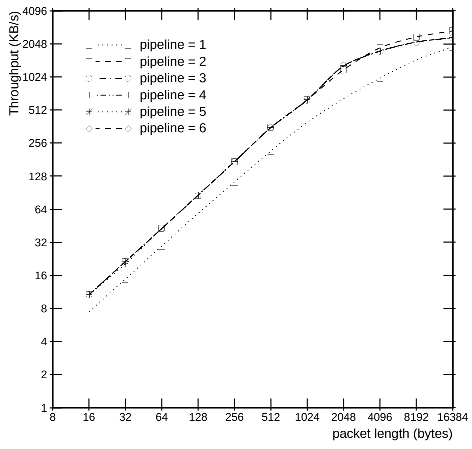
We performed several experiments using the CIM to determine its performance characteristics, namely, the factors that affected data throughput and medium access latency (latency to access and use the network link). Packet size for transmission/reception and the number of outstanding packet transmissions (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 through 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 4(a) plots the achieved throughput, and Figure 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.

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 but starts to saturate beyond a packet size of 2K bytes. 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. In the interrupt mode of operation, each command from the NP generates an interrupt on the CIM while each response from the CIM generates an interrupt on the NP. 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, queueing delays inside the CIM begin to dominate, eliminating any gains in throughput. For larger packets, the time to DMA the packets to the CIM begins to dominate and hence the 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 VME bus (see Table 1 in Section 2.3)[4].

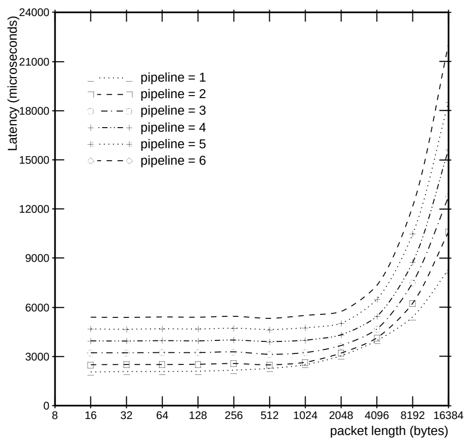
Referring to Figure 4, the medium access latency using the CIM remains roughly constant for packet sizes up to 2 KB; for larger packets the DMA overhead and the transmission time both increase, increasing the medium access latency rapidly beyond a packet size of 4 KB. The latency increases monotonically with pipeline depth since packets awaiting transmission experience higher queueing delays in the CIM. To further understand the behavior of the CIM, we traced all the command/response interactions between the NP and the CIM and measured the individual components of the medium access latency. The results (not shown here) confirmed the trends observed in Figure 4 but also revealed substantial unpredictability in the medium access latency. More specifically, the delay between initiating a DMA on the CIM and getting the transmission-complete interrupt becomes highly unpredictable for packet sizes larger than 4 KB and pipeline depths larger than 2. Both these effects are a direct consequence of FIFO queueing 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, the NP can exercise fine-grain control over the order of packet transmissions; this also helps bound the medium access latency.

An unrestricted pipeline could introduce unacceptable delay jitter by introducing traffic-dependent variations in medium access latency. Since real-time communication necessitates low, predictable medium access

[4] In all fairness to the manufacturer, we have learnt that the new version of the adapter has addressed some of these design weaknesses.



(a) Throughput



(b) Medium access latency

Figure 4: Performance of Ancor CIM 250 network adapter

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 [15] and real-time audio/video. Since the CIM does not distinguish best-effort traffic from 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, we chose to fix the pipeline depth at 2 and packet size at 2 KB; this provides good performance for a mix of real-time and best-effort traffic.

4.2 Desirable Adapter Features

Several aspects of adapter design affect performance on packet reception. The received packet could be in error (as indicated by the CIM) or it may have violated its deadline. Deadline violations could occur under high communication load in statistical real-time channels employing 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, shortest deadline first, etc.). The provision of large on-board memory on the CIM 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 on 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.

Accordingly, the adapter features we consider desirable for real-time communication on bus-based multiprocessor (and uniprocessor) hosts include:

- support for efficient network data transfer through a symmetric location within the host with respect to the sources/sinks of data, simplified device interface, 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 preemptability by decoupling data transfer to the adapter from the initiation of transmission.

We elaborate on these features using our design of SPIDER as a network adapter for real-time communication[5]. SPIDER is also a VME bus-based design, has reasonably large and fast on-board memory (1 MB SRAM), and provides DMA capabilities for moving data directly to/from SPIDER and other VME devices. Other SPIDER features are discussed below.

Flexible buffer management: SPIDER's on-board memory is directly accessible to the NP and is managed as per the NP's buffer management policies. SPIDER views its memory as a collection of memory *pages* which are allocated on a per-packet basis by NP; the NP specifies the pages to use for transmission and reception using *page tags*. On transmission these pages are filled with outgoing data from the source AP (or device) via DMA before the corresponding packet is scheduled for transmission; on reception packet data in the allocated pages is transferred to the destination AP (or device) via DMA, if the packet is accepted. Two page sizes are supported for data transfer: 256-byte pages for headers and short packets, and 1K-byte pages for long packets. The NP allocates contiguous pages to a packet's data; this facilitates DMA transfer of packet data in large chunks and improves the utilization of VME bus bandwidth.

Efficient header inspection and (selective) packet discard: Packet headers, which are placed on separate 256-byte pages, can be examined independently by the NP since it manages all SPIDER pages and has full access to SPIDER's memory. The ability to examine packet headers facilitates reception path optimizations by allowing the NP to make intelligent decisions about the data before the data actually consumes transfer bandwidth within the node. Since SPIDER's on-board CRC unit flags errors on a per-packet basis, the NP can defer handling of such invalid packets until a convenient time or drop the packet right away without using any additional host resources; the buffer space thus freed can be reclaimed by simply reusing the corresponding pages.

Intelligent handling of in-transit traffic: Since SPIDER integrates all the links at the node on a single board, intermediate node traffic can be received and buffered in on-board pages, while the NP examines the headers and schedules the packets for later transmission. Thus, intermediate node traffic need not consume any host resources other than processing bandwidth on the NP. Note that, since the required header modifications are typically minor, the NP can examine and modify header pages for in-transit traffic directly in SPIDER memory; this avoids the movement of header pages between the NP and SPIDER across the VME bus and eliminates expensive interrupt handling. Since it manages SPIDER's memory, the NP can allocate buffers to real-time channels intelligently by exploiting the access cost differential between buffers in NP memory and those in SPIDER memory.

Simplified device interface: SPIDER exports a simpler control interface to the NP, reducing device handshake overhead. Transmission and reception involves a minimum of two interrupts each (if operating SPIDER in the interrupt mode), one indicating completion of DMA transfers and another indicating completion of packet transmission or reception. The device can be programmed to generate additional per-page interrupts or mask interrupts selectively in order to reduce the number of interrupts generated. No interrupts are generated in the polled mode; in this mode the NP periodically examines status information in SPIDER.

[5]The design is in the final stages of development and we expect to fabricate the board in the near future.

Bounded medium access latency: To minimize the interference between real-time and best-effort traffic, SPIDER provides *virtual channels* to partition the network into virtual networks, one for real-time and one for best-effort; arbitration between these virtual channels on each link ensures fairness. Different considerations can be applied to real-time traffic (delay-sensitive) and best-effort traffic (throughput-sensitive) to determine the optimum packet size (and SPIDER's page size) for each. In order to achieve low, predictable medium access latency, the NP must exercise a finer grain of control over link access. This is achieved through the provision of a "preemption point" during data transfer by breaking the coupling between the transfer of data to SPIDER memory and the actual initiation of transmission by the NP. The NP can "schedule" data transfers to/from SPIDER memory while maintaining a distinct schedule for packet transmissions and receptions. Within SPIDER, the medium access latency per virtual channel (and hence per link) is bounded; each virtual channel (link) is serviced within a certain worst-case time independent of the number of contending virtual channels (links). To reduce the medium access latency further and make it more predictable, FIFO queueing inside SPIDER is limited to a depth of six pages for each virtual channel.

5 Overheads of Protocol Processing

The overheads involved in protocol processing on the NP significantly impact the implementation of real-time channels and the ability to support real-time communication in general. We consider two components of these overheads: 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.

5.1 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 [19–22]. 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, we must 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, as determined by the link scheduler's transmission order. Thus, it is essential to optimize data transfer, not just 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 and fragmentation into packets. Besides, this approach still suffers from FIFO consumption of bus *and* adapter/link resources. The absence of priority-based arbitration on the VME bus 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 of 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.

Note that, while we assume application data to be resident in physically-contiguous buffers on APs or other VME bus devices, our optimization approach is applicable in general. The reception path can be optimized similarly to move received data directly from the CIM to the appropriate AP or VME bus device via DMA. Our implementation currently does not optimize the reception path, and there is an intervening copy of data into the NP. Section 8 briefly highlights the ongoing effort to implement reception path optimizations and support scatter-gather transfer of data.

5.2 Software Overheads and Protocol Thread Scheduling

With no data movement costs incurred during protocol processing on the NP, the overheads of fragmentation by FRAG, encapsulation by HNET, and processing by the link scheduler and CIM device driver become important. The fragmentation (and reassembly) overhead is incurred only at the source and destination nodes while the HNET, scheduler and CIM driver overheads are incurred at all the nodes along the route. For a given fragmentation size and with no data copy, the software overheads are directly proportional to the number of fragments and are 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.

5.2.1 Fragmentation and Link Scheduling Overheads

Figure 5 plots the latency of protocol processing and link scheduling as a function of message size with a *null device*, i.e., without the CIM. 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 overheads are 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 $\approx 100\mu s$ (curve labeled `without scheduler (no frag)`). With the scheduler and CIM driver in the path, the per-packet processing time increases to $\approx 250\mu s$ (curve labeled `with scheduler (no frag)`). Of the additional overhead of $150\mu s$, about $60\mu s$ is attributed to instruction cache misses due to context switching to the scheduler; 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 will always run 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 and 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, once it has been transferred to NP memory via DMA, and the corresponding protocol thread 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 ($\approx 400\mu s$) dominates the incremental cost of generating additional fragments ($\approx 90\mu s$); this explains the sudden drop and subsequent slow climb in throughput. Performance again degrades with the scheduler in the transmission path (curve labeled

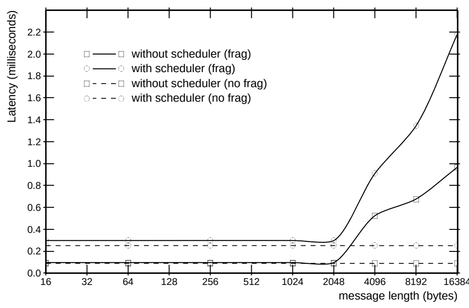


Figure 5: Protocol processing performance with and without fragmentation (null device)

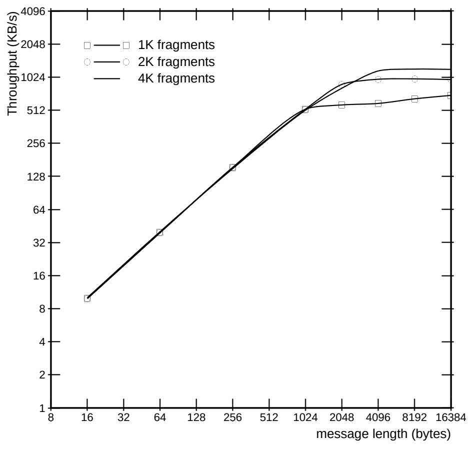
`with scheduler (frag)`), with a per-fragment scheduler processing overhead of $\approx 200\mu s$. The degradation increases with the number of fragments because of higher additional cost of fragmentation as well as an increase in the processing done by the scheduler and the CIM driver.

5.2.2 Scheduling Protocol Threads

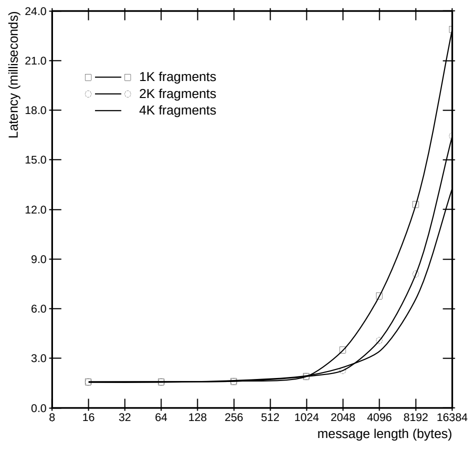
With several real-time channels and best-effort traffic active simultaneously, it is critical that protocol threads be scheduled to consistently maintain QoS guarantees. The protocol-processing bandwidth must be consumed in a global order consistent with the traffic parameters of the active channels. Straightforward FIFO scheduling of protocol threads can introduce significant queueing delays, especially for large messages, as is evident from Figure 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 overheads 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.

Our approach to protocol thread scheduling is based on slot occupancy-based control of thread state (runnable or not), priority-based CPU allocation to runnable protocol threads, and preemption of executing threads through voluntary release of the CPU. The main idea here is to allow a thread to compete for CPU access only if the link scheduler can accommodate the packets generated by this thread, allocate the CPU to the highest priority thread amongst the runnable threads, and define "safe" preemption points during the execution of a protocol thread at which the CPU can be reallocated to a higher-priority thread. Safe preemption points are points during protocol execution when preemption of the thread would not obstruct the execution of other protocol threads and the thread can successfully resume execution at a later time. The x-kernel uses a priority-based (thread) scheduler with 32 priority levels; the CPU is allocated to the highest-priority runnable thread, while thread scheduling within a priority level is FIFO. We follow the following policy for priority assignment to threads. The link scheduler thread is assigned the highest priority for CPU access, while all protocol threads shepherding best-effort traffic are assigned the lowest priority. Protocol threads shepherding real-time traffic are assigned the corresponding channel's priority computed during channel establishment, mapped to the remaining 30 priority levels in the x-kernel.

Since the link scheduler runs at the highest priority, it must relinquish the CPU for runnable protocol threads from time to time. The write semaphore controlling access to the link ensures that the link scheduler will be blocked after initiating the allowed number of packet transmissions (determined by the pipeline depth) each time it is invoked. An executing protocol thread must relinquish the CPU if a higher-priority thread becomes runnable or if the scheduler runs out of slots for this thread's packets. Handing off the CPU to



(a) Throughput



(b) Latency

Figure 6: Protocol processing performance with fragmentation (with CIM)

a higher-priority thread is different from normal scheduling since the relinquishing thread must stay at the head of its queue; this ensures that the CPU is allocated back to this thread once all the higher-priority threads have relinquished the CPU. If the scheduler runs out of slots, the executing thread must block until further notification of slot availability by the scheduler; when woken up the thread is inserted at the tail of the ready queue at its corresponding priority level. The location and number of safe preemption points during protocol processing can only be determined by exploring the tradeoff between the total amount of work the thread will do (number of fragments and the per-fragment processing cost), the desired bound on CPU access latency, and the overhead incurred due to thread preemption. Thread preemption overhead includes the cost of additional context switches and cache misses, the cost of checking the need for preemption at each preemption point, and the efficiency with which notification regarding slot availability can be exchanged between the link scheduler and protocol threads. The exploration of these tradeoffs is a subject of ongoing work.

5.3 End-to-End Performance

Figure 6 plots the end-to-end throughput ((a)) and latency ((b)) of message transfer between the NPs on two nodes using the CIM, with fragmentation and link scheduling. Figure 6 can be directly compared to Figure 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 Figure 4 and Figure 6, for large messages (16 KB) fragmentation using 4 KB fragments reduces the achieved throughput from $\approx$ 2.7 MB/second to $\approx$ 1.2 MB/second.

6 Effectiveness of Link Access Scheduling

In this section we evaluate the effectiveness with which the link scheduler insulates real-time traffic from best-effort traffic, and prevents 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 (queueing 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 actual arrival time. The slot occupancy (or queueing delay) measures the actual time that an outgoing packet occupied a packet slot (equivalent to a buffer) in the link scheduler.

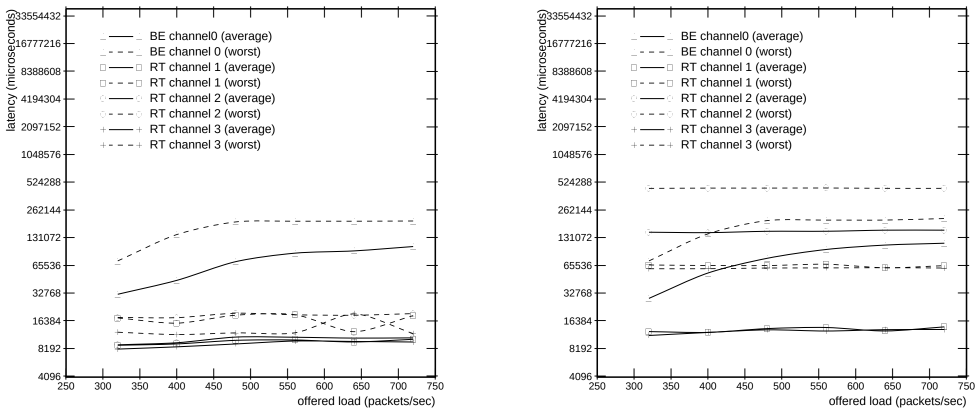
6.1 Outline of Experiments

For the 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. The tasks generating real-time traffic execute on one AP, while the task generating best-effort traffic runs on a different AP. On the NP protocol processing for real-time traffic was 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 5.2. 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 8KB, i.e., messages consist of 4 packets. The deadline for each real-time channel is set at 50 *ms* and the link horizon [2], which controls the degree to which the scheduler is work-conserving, is set at 0 *ms*. Figure 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 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.

6.2 Effects of best-effort traffic load on real-time traffic

Figure 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 has a bursty source; 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 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; 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; subsequently, most additional messages are dropped. Latencies also increased gradually with load, until the system reached saturation. Figure 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 adversely affect best-effort performance.

The experiment in Figure 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 8 (a) looks almost identical to Figure 7 (a), the excess packets on Channel 3 are dropped once the buffers available to Channel 3 are ex-



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

Figure 7: Well-behaved real-time channels with variable best-effort load

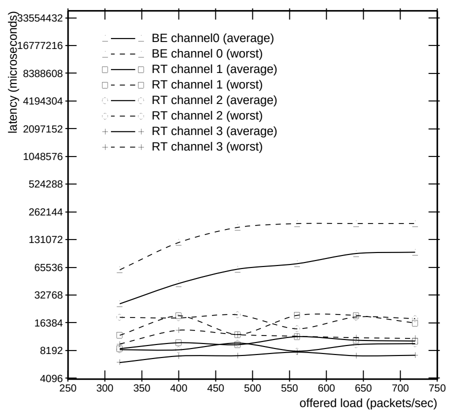
hausted. Note that, though real-time traffic is assigned a higher priority than best-effort traffic, increasing the real-time load in this manner does not significantly affect the performance of best effort or real-time traffic. In addition, the packets of Channel 3 that are delivered at all, are all delivered by their deadlines. A comparison of Figure 8 (b) with Figure 7 (b) shows that queueing delays did not increase for best-effort traffic, or the well-behaved channels; however, queueing delays shot up for Channel 3. Results of the same experiment with a bursty misbehaving source are similar to the ones reported here.

6.3 Effects of burstiness and message size on delay guarantees

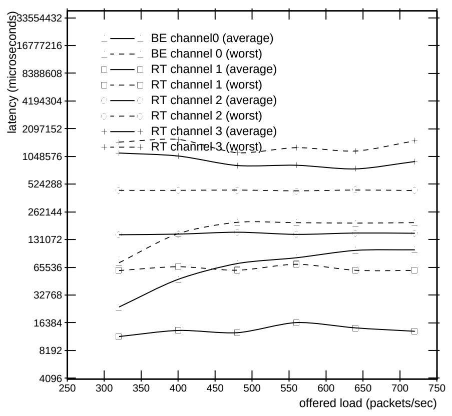
As seen from the results so far, bursty sources have a greater slot occupancy time (larger queueing delay) than periodic sources. We repeated the experiments with message sizes from 4KB–32KB, while retaining the same total loads. Since each slot in the scheduling queue corresponds to a packet, longer 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 with 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 5.2.

7 Related Work

While we have focused on design tradeoffs in implementing real-time channels, similar tradeoffs arise when implementing real-time communication in general. Our implementation methodology is applicable to other proposals for supporting guaranteed real-time communication in packet-switched networks. A detailed survey of the proposed techniques can be found in [23]. The most notable proposals are Weighted Fair Queueing [24], also known as Packet-by-Packet Generalized Processor Sharing [25], Stop-and-Go [26], Hierarchical Round-Robin [27], and Rate-Controlled Static-Priority Queueing [28]. Proposals for predicted (or best-effort) real-time communication include FIFO+ [29] and Hop-Laxity [30]. The Internet Engineering Task Force (IETF)



(a) Message latency



(b) Slot occupancy (queueing delay)

Figure 8: Ill-behaved real-time channels with variable best-effort load

is examining these issues in the context of providing integrated services on the Internet [31, 32].

The Tenet real-time protocol suite [12] is an advanced implementation of real-time communication on wide-area networks (WANs). This protocol suite comprises the RCAP channel administration protocol and the RTMP/RTIP transport and network layer protocols, which implement unicast real-time channels in UNIX[6]. The effectiveness of these protocols in providing and maintaining QoS guarantees has also been demonstrated [33]. Since UNIX-based uniprocessor workstations is the implementation platform, the Tenet approach uses the socket API (hence incurring data copying costs) and does not consider issues arising in a multiprocessor configuration. The problem of making protocol processing inside the host more predictable is also not addressed. While their implementation uses standard network adapters, they do not consider the impact of adapter characteristics on the ability to support real-time communication effectively.

Other protocols for resource reservation include the Session Reservation Protocol (SRP) and resource ReSerVation Protocol (RSVP). The SRP [34] was proposed as a (compound) session establishment protocol for IP networks as part of the DASH project [18, 35]. Recently, the RSVP has been proposed for use in the Internet [36]. While SRP is geared toward unicast sessions with performance guarantees, and is similar in flavor to NMP, RSVP is geared more towards multi-point multiparty communication.

The issue of making protocol processing predictable within (uniprocessor) hosts has received attention recently. The need for scheduling protocol processing at priority levels consistent with those of the communicating application was highlighted in [37] and some implementation strategies demonstrated in [38]. More recently, processor capacity reserves in Real-Time Mach [39] have been combined with user-level protocol processing [19] to make protocol processing inside hosts predictable [40]. Since we dedicate a processor for communication processing, only protocol threads compete with each other for processing resources. We have incorporated similar ideas for priority-based protocol processing in our implementation to make per-channel protocol processing more predictable.

8 Conclusion and Future Work

In this paper we have explored the design tradeoffs involved in supporting real-time communication on bus-based multiprocessor hosts, which are increasingly being employed as multimedia servers and work-

[6]The Tenet group is currently developing Suite 2 of the protocols to support multi-party real-time communication.

stations. As the vehicle for this study, we implemented and evaluated real-time channels on the HARTS experimentation platform.

Our main contributions are summarized as follows. We have presented a hardware and software architecture 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 most 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, we present techniques to handle undesirable features such as unrestricted FIFO queueing in order to bound the medium access latency. After identifying desirable features of the network adapter, we have illustrated how these features are realized in the custom network adapter being developed for the HARTS project. To circumvent lack of hardware support for priority-based access to resources, we presented data transfer optimizations, CPU and link scheduling mechanisms to limit interference between different real-time channels. 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, we demonstrated the performance and effectiveness of our real-time channel implementation through several experiments under varying traffic characteristics.

Besides exploring the tradeoffs involved in CPU scheduling of protocol threads, as identified in Section 5.2, our future work focuses on data transfer optimizations on reception, using the CIM as well as SPIDER. Optimizing the reception path requires modifications to link-layer headers and enhanced support for buffer management on the APs and the NP. The necessary information in link-layer headers may include fragmentation size, message size, fragment number, and an identifier for efficient demultiplexing to the receiving device or end point. Reception path optimizations are greatly facilitated by the use of a known, fixed fragmentation size in the network and the alignment of data buffers on fragment boundaries; if the fragmentation size is an integral multiple of 4-byte words, data buffers need only be word-aligned. In order to deliver arriving data directly to the destination AP (or device), the NP must identify the destination device and may need to manage buffers on behalf of the APs. Pre-registration of application buffers with the NP and/or application buffer management by the NP may be required to DMA received data directly from the CIM to the application's address space. In this case, the remote references maintained by the NP would correspond to logical addresses and must be translated into physical addresses for each received packet. Finally, we are exploring the necessary support for scatter-gather transfer of data within the same optimization framework.

References

[1] 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.

[2] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multihop networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994. An earlier version appeared in the Proc. of the Int. Conf. on Distr. Comp. Sys., 1991.

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

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

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

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

[7] J. Dolter, S. Daniel, A. Mehra, J. Rexford, W. Feng, and K. Shin, "SPIDER: Flexible and efficient communication support for point-to-point distributed systems," in *Proc. Int'l Conf. on Distributed Computing Systems*, pp. 574–580, June 1994.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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," Technical Report TR-94-25, Department of Computer Science, University of Minnesota, 1994.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

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

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

[29] 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.

[30] 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*, June 1994.

[31] S. Shenker, D. D. Clark, and L. Zhang. *A Service Model for an Integrated Services Internet.* IETF working draft, October 1993.

[32] B. Braden, D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview.* IETF working draft, October 1993.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.