Resource Management for Real-Time Communication: Making Theory Meet Practice

Ashish Mehra Atri Indiresan
Kang G. Shin

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

ABSTRACT

A growing number of real-time applications (e.g., real-time controls, and audio/video conferencing) require certain quality-of-service (QoS) from the underlying communication subsystem. Real-time communication services are needed in the communication subsystem (host as well as network) to provide the required QoS to these applications while providing reasonably good performance for best-effort traffic. At the host, real-time communication necessitates that shared host resources such as CPU and link bandwidth be consumed according to the relative requirements of the active connections. This requires appropriate resource management policies for admission control and scheduling that are typically formulated assuming idealized resource models. However, when implementing these policies one must account for the performance characteristics of the hardware and software components involved, which could deviate significantly from those of the idealized resource models.

In this paper, we focus on bridging the gap between theory and practice in the management of host CPU and link resources for real-time communication. Using our implementation of *real-time channels*, a paradigm for real-time communication in packet-switched networks, we illustrate the tradeoff between resource capacity and channel admissibility, which determines the number and type of real-time channels that can be accepted for service and the performance delivered to best-effort traffic. We demonstrate that this tradeoff is affected significantly by the choice of implementation paradigms and the grain at which CPU and link resources can be multiplexed amongst active channels. In order to account for this effect, we extend the admission control procedure for real-time channels originally proposed using idealized resource models. Our results show that practical considerations significantly reduce channel admissibility compared to idealized resource models. Further, the optimum choice of the multiplexing grain depends on several factors such as the overheads of resource preemption, the relationship between CPU and link bandwidth, and the manner in which allocation of link bandwidth interacts with allocation of CPU bandwidth.

Key Words — Real-time communication, resource management, QoS-sensitive 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 grants 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

The advent of high-speed networks has generated an increasing demand for a new class of distributed applications that require certain quality-of-service (QoS) guarantees from the underlying network. QoS guarantees may be specified in terms of several parameters such as the end-to-end delay, delay jitter, and bandwidth delivered on each active connection; additional requirements regarding packet loss and in-order delivery can also be specified. Examples of such applications include distributed multimedia applications (e.g., video conferencing, video-on-demand, digital libraries) and distributed real-time command/control systems. In order to support these applications, both the communication subsystem in end hosts (or simply hosts) and the network must be designed to provide QoS guarantees on individual connections.

Given appropriate support within the network, host communication resources must be managed in a *QoS-sensitive* fashion, i.e., according to the relative importance of the connections requesting service. For a sending host,[1] communication resources include CPU bandwidth for protocol processing and link bandwidth for packet transmissions, assuming sufficient availability of buffer space. QoS-sensitive management of communication resources necessitates policies for admission control and resource scheduling that together guarantee that each connection will obtain at least its required QoS. These resource management policies are typically formulated using idealized models of the resources being managed. For example, it may be assumed that a given resource is immediately preemptible and/or the cost of preemption is negligible. More importantly, it may be assumed that a required set of resources can be accessed, and hence allocated, independent of one another. However, the above assumptions can be violated when implementing resource management policies, since the performance characteristics of the hardware and software components employed can deviate significantly from those of the idealized resource models.

In this paper, we focus on bridging the gap between theory and practice in the management of host CPU and link resources for real-time communication. For this purpose we utilize *real-time channels*, a paradigm for real-time communication in packet-switched networks [1]. The model of real-time channels is similar to other proposals for guaranteed-QoS connections [2]. Using our implementation of real-time channels [3], we illustrate the tradeoff between useful resource capacity and channel admissibility. Useful resource capacity refers to the proportion of the raw resource capacity that can be utilized effectively. In the context of real-time channels, useful resource capacity determines the number and type of real-time channels that can be accepted for service and the performance delivered to best-effort traffic.

A channel requires a portion of the available CPU bandwidth to process each generated message and packetize it. Similarly, it requires a portion of the available link bandwidth to transmit each packet on the link. Since these two resources typically differ in their performance characteristics, they present different tradeoffs for resource management. Assuming that both the CPU and link can be reallocated only at packet boundaries, allocating the link to transmit a packet from another channel usually has no additional overheads associated with it. However, allocating the CPU to perform protocol processing for another channel incurs significant overheads in the form of context switches and cache misses. Excessive CPU preemption, therefore, reduces available resource capacity, effectively increasing the resource usage attributed to a channel. Limiting preemption, however, increases the temporal window of priority inversion for CPU access, potentially impacting channel admissibility negatively. Correspondingly, if the size of each packet is increased, this reduces the total CPU bandwidth required to process a given message. However, this is accompanied by

[1]The issues involved in resource management at the receiving host are similar, but beyond the scope of this paper.

an increase in the transmission time of a packet, and hence in the temporal window of priority inversion for link access; again, this potentially impacts channel admissibility negatively.

We demonstrate that the above-mentioned tradeoff is affected significantly by the choice of implementation paradigms and the (temporal) grain at which CPU and link resources can be multiplexed amongst active channels. In order to account for this effect, we extend the admission control procedure for real-time channels that was originally proposed using idealized resource models. Our results show that, compared to idealized resource models, practical considerations significantly reduce channel admissibility. Further, the optimum choice of the multiplexing grain depends on several factors such as the overheads of resource preemption, the relationship between CPU and link bandwidth, and the manner in which allocation of link bandwidth interacts with allocation of CPU bandwidth.

The rest of the paper is organized as follows. A brief description of the real-time channel model of guaranteed-QoS communication is provided in Section 2. Section 3 discusses the issues involved in managing CPU and link bandwidth for QoS-sensitive protocol processing and packet transmissions, respectively, in the context of real-time channels. The modifications required in the admission control procedure to manage CPU and link bandwidth simultaneously are presented in Section 4. Section 5 studies the implications for channel admissibility by illustrating the tradeoff between useful resource capacity and channel admissibility. We also discuss the selection of the optimal resource multiplexing grain for real-time communication while simultaneously carrying best-effort traffic. Our work is contrasted with related work in Section 6. Finally, Section 7 concludes the paper.

2 Real-Time Channels

A real-time channel is a simplex, fixed-route, virtual connection between a source and destination host, with sequenced messages and associated performance guarantees on message delivery. The data flow on real-time channels is unidirectional, from source to sink via intermediate nodes, with successive messages delivered in the order they were generated. Corrupted, delayed, or lost data is of little value; with a continuous flow of time-sensitive data, there is not sufficient time to recover from errors. Data transfer on real-time channels occurs without acknowledgements and retransmissions, and hence has unreliable-datagram semantics.

An application requests a real-time channel by specifying its QoS requirements and traffic characteristics. Since network resources (buffers, processing capacity, and link bandwidth) are finite, the communication subsystem and the network must perform admission control to provide any kind of performance guarantees. As part of admission control tests, the resources required to satisfy the application's request are computed based on the specified worst-case traffic, and the request accepted if sufficient resources can be reserved for it. A local bound, which determines the worst-case transit delay seen by a packet on this channel plus a certain slack, is also assigned to each node. Once the channel is successfully established, the communication subsystem along with the network maintain QoS guarantees via resource management, traffic policing and enforcement policies. When the application requests that the channel be destroyed, all resources allocated for the channel are released by the network and the communication subsystems at the source and destination hosts.

2.1 Traffic Generation Model

The traffic generation model in real-time channels is based on a *linear bounded arrival process* [4, 5], which is characterized by three parameters:

- maximum message size (S_{max} bytes),
- maximum message rate (R_{max} messages/second), and
- maximum burst size (B_{max} messages).

In any interval of length δ, the number of messages generated is bounded by $B_{max}+\delta\cdot R_{max}$. Message generation rate is bounded by R_{max}, and its reciprocal, I_{min}, is the minimum 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. To ensure that a real-time channel does not use more resources than it reserved at the expense of other channels' QoS guarantees, this model uses the notion of *logical arrival time* to enforce a minimum separation I_{min} between messages on the same real-time channel. The logical arrival time, $\ell(m)$, of a message m is defined 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; $\ell(m_i)$ is the time at which m_i would have arrived (generated) if the maximum message rate constraint was strictly obeyed.

The QoS on a real-time channel is specified in terms of a desired deterministic, worst-case bound on the end-to-end delay experienced by a message; the delay bound is, therefore, specified independent of the desired bandwidth. If d is the desired end-to-end delay bound for a channel, message m_i generated at the source is guaranteed to be delivered at the sink by time $\ell(m_i)+d$. More details on the real-time channel model can be found in [1].

2.2 Resource Management

As with other proposals for guaranteed-QoS communication [2], there are two related aspects to resource management for real-time channels: admission control and (run-time) scheduling. Admission control for real-time channels is provided by Algorithm `D_order` [1], which uses fixed-priority scheduling for computing the worst-case delay experienced by a channel at a link. A message is considered to be a set of one or more packets, where the packet size is bounded. This enables message transmission to be interrupted at the end of a packet transmission, without loss; note that packet transmission on the link is non-preemptive. When a channel is to be established at a link, the worst-case response time for a message (when the message completes transmission on the link) on this channel is estimated based on non-preemptive fixed-priority scheduling of packet transmissions. The priority assigned to the new channel depends upon the characteristics of the other channels going through the link. The total response time, which is the sum of the response times over all the links on the route of the channel, is checked against the maximum permissible message delay and the channel can be established only if the latter is greater. The permissible message delay is split proportionally among the different links. The priority assignment algorithm ensures that the new channel does not affect the QoS promised to existing channels. Note that

`D_order` assumes that for all channels, the worst-case delay at each link for any channel does not exceed its message inter-arrival time. The total end–to–end delay, however, can exceed the message inter-arrival time of the channel.

Contrary to the approach for admission control, run-time message scheduling is governed by a variation of the multi-class EDF policy. The (message) scheduler maintains three queues, Queue 1, Queue 2, and Queue 3 for each outgoing link, corresponding to three service classes. Queue 1 and Queue 3 contain packets which belong to real-time channels, while Queue 2 contains all other types of packets. More specifically, Queue 1 contains *current* real-time packets, which have to be scheduled in the order of their deadline, hence it is organized by increasing packet deadlines. Current real-time packets are those packets whose logical arrival time is less than the current clock time at the node. Accordingly, Queue 1 is assigned the highest priority for link access. Queue 2 contains packets for which no guarantees are given, i.e., they receive best-effort service, and is organized as a first-in-first-out (FIFO) queue; Queue 2 is assigned a link access priority below Queue 1. The service provided for this class of packets is improved by giving them priority over real-time packets which are not *current*. Packets in Queue 3 are those which have arrived early, either because of burstiness in the message generation or because they encountered delays which were smaller than the budgeted worst-case delays at some upstream nodes. These packets are stored in the order of their logical arrival time because they have to be transferred to Queue 1 as they become current. Queue 3 effectively *shapes* early (non-compliant) traffic into current (compliant) traffic. More details on the processing done by the link scheduler are given in [1], and the overheads and effectiveness of the implementation discussed in [3].

Note that the above model of real-time channels only applies to management of link bandwidth and does not account for management of CPU processing bandwidth at the host. More importantly, it cannot be extended directly to CPU bandwidth management, as discussed in Section 3.

2.3 Implementation

We have implemented a QoS-sensitive communication subsystem architecture featuring real-time channels [3]. Our implementation employs a communication executive derived from x-kernel 3.1 [6] exercising complete control over a Motorola 68040 CPU. The protocol stack utilized for real-time communication includes protocols for resource reservation (channel establishment and teardown), remote procedure call, transport-level fragmentation, network-level encapsulation, and clock synchronization. The fragmentation protocol is an sequenced, unreliable protocol used to packetize large messages so that communication resources can be multiplexed between channels on a packet-by-packet basis. The network-level encapsulation protocol is a connection-oriented protocol that provides for data transport across a point-to-point communication network. The choice of the transport and network layer protocols is based on the perceived protocol requirements for real-time channels.

2.3.1 QoS-sensitive CPU and Link Scheduling

The implementation 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 handler is associated with each channel to perform protocol processing for all messages generated on the channel. Channel handlers are scheduled for execution using a multi-class earliest-deadline-first (EDF) scheduler 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 the CPU access latency, this allows us to study the influence of preemption granularity and overheads on channel admissibility.

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 design is 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 are scheduled for transmission by a non-preemptive multi-class EDF link scheduler, as explained in Section 2.2. More details of the protocol stack and the real-time channel implementation are given in [3].

2.3.2 Null Network Device

In order to explore the effects of the relationship between CPU and link bandwidth, we have implemented a device emulator, referred to as the *null device*, that can be configured to emulate any desired packet transmission time $\mathcal{L}_{xmit}$. Packet transmission time is the minimum time that must elapse between successive packet transmissions on the link. Thus, it suffices to ensure that successive packet transmissions can be invoked every $\mathcal{L}_{xmit}$ time units apart. This can be achieved by *emulating* the behavior of a network adapter such that $\mathcal{L}_{xmit}$ time units are consumed for each packet being transmitted.

The device emulator is simply a thread that, once signalled, tracks time by consuming CPU resources for $\mathcal{L}_{xmit}$ time units before signalling completion of packet transmission. This emulator is implemented on a separate processor that is connected via a backplane system bus to the processor implementing the communication subsystem (the host processor). Upon expiry of $\mathcal{L}_{xmit}$ time units (and hence completion of packet transmission) the emulator issues an interrupt to the host processor, similar to the mechanism employed in typical network adapters. While the emulator allows us to study a variety of tradeoffs, including the effects of the relationship between CPU and link processing bandwidth, it is not completely accurate since no packet data is actually transferred from host memory. However, this does not have a significant effect on the trends observed and performance comparisons reported here.

3 Implementation Issues in Managing CPU and Link Bandwidth

As mentioned earlier, in order to admit a channel algorithm `D_order` computes the worst-case response time for a message. This response time has two components: the time spent waiting for resources and the time spent consuming resources. At the host, the time spent consuming communication resources is equal to the *message service time*, the time required to process and transmit all the packets constituting the message. In order to calculate the time spent waiting for resources, one must consider the preemption model used for resource access.

The real-time channel model presented in [1] accounts for the effects of non-preemptive packet transmissions, but assumes an ideal preemption model for CPU access. That is, the CPU can be allocated to a waiting higher-priority handler immediately at no extra cost. Under such a model of CPU allocation, the message service time is determined solely by the CPU processing bandwidth required to packetize the message, and the link bandwidth required to transmit all the packets. The time spent waiting for resources can then be calculated by accounting for resource usage by messages from all higher-priority channels, and the one-packet delay (due to non-preemptive packet transmission) in obtaining the link. However, as explained below, implementation issues necessitate extensions to the model to account for implementation overheads and constraints.

3.1 Implementation Issues

Several issues related to implementation impact resource management policies. These include handler execution and processing requirements, the implementation of link scheduling, and the relationship between CPU and link bandwidth.

3.1.1 Handler Execution

In most modern processors, preemption of an executing process/thread comes with a significant cost due to context switch and cache miss penalty. Preemption effectively increases the CPU usage attributed to a channel, which in turn reduces the CPU processing bandwidth available for real-time channels; immediate preemption is thus too expensive. It is desirable to limit the number of times a handler is forced to preempt the CPU in the course of processing a message. At the other extreme, non-preemptive execution of channel handlers implies that the CPU can only be reallocated to a waiting handler after processing an entire message. This results in a coarser (temporal) grain of channel multiplexing on the CPU and makes admission control less effective. In order to bound the window of preemptibility, admission control must consider the largest possible message size across all real-time and best-effort channels.[2] An intermediate solution is to preempt the CPU only at specific preemption points, if needed. Since the CPU processing in question involves packetization of a message, the CPU can be preempted after processing every packet of the message. Thus, the important parameter here is the number of packets processed between preemption points, which determines the (temporal) grain at which the CPU can be multiplexed between channels. The admission control procedure must be suitably modified to account for the extra delay in obtaining the CPU which may be currently allocated to a lower-priority channel handler.

The total CPU time required to process a message is directly proportional to the number of packets constituting the message. Clearly, assuming that the communication subsystem does not copy message data unnecessarily,[3] the CPU processing time will be minimum if a single packet constituted the entire message, i.e., if the packet size was the same as the message size. However, as explained in Section 3.1.4, the total time required to transmit a message on the link is determined primarily by the size of the message, although initiation of transmission involves non-zero per-packet overhead. If the set of channels requesting service have identical traffic specifications, and hence the same maximum message size, then single-packet messages maximize channel admissibility. However, under a heterogeneous mix of real-time channels (with large and small messages), a large packet size would significantly reduce the admissibility for channels with messages smaller than the

[2]The maximum message size for best-effort traffic may not even be known *a priori*.

[3]This is true for our implementation of real-time channels.

chosen packet size. Selection of the packet size, therefore, also plays a significant role in determining channel admissibility.

3.1.2 Implementation of Link Scheduling

An assumption often made when formulating resource management policies for communication resources is that CPU and link bandwidth can be independently allocated to a channel. This assumption may get violated in an implementation depending on the paradigm used to implement link scheduling. We consider three different options for implementing link scheduling in software:

O1 Packets are scheduled for transmission either in the context of the currently executing channel handler (via a function call) or in interrupt context after each packet transmission.

O2 The link scheduler is implemented as a dedicated thread/process that executes at the highest possible priority and is signalled via semaphore operations.

O3 There is no dedicated thread but packets are scheduled for transmission either in the context of the currently executing channel handler or in the context of a new thread that is fired up after every packet transmission.

Options O1 and O2 differ significantly in the implications for CPU and link bandwidth allocation, since with O2 the link scheduler itself must be scheduled for execution on the CPU. Since option O3 presents tradeoffs similar to option O2, we focus on O1 and O2 in the discussion below.

Selecting a packet for transmission incurs a certain amount of overhead in addition to that of initiating transmission on the link. Additional overhead may be involved if the link scheduler must transfer early packets from Queue 3 to Queue 1 (Section 2.2). In O1, the scheduler is frequently invoked from the interrupt service routine (ISR) of the interrupt announcing completion of packet transmission. Since the scheduling overhead involved can be substantial in the worst case, it is undesirable to incur this penalty in the ISR, since this prolongs the duration for which network interrupts are disabled. If the host is also receiving data from the network, there is now a greater likelihood of losing incoming data.

O2, on the other hand, does not suffer from this problem; since scheduler processing is scheduled for execution, it is performed outside the ISR. In addition to keeping the ISR short, this paradigm also has some software structuring benefits such as a relatively cleaner implementation. However, because the link scheduler is itself scheduled for execution on the CPU, there is now an additional overhead of a context switch and the accompanying (instruction) cache miss penalty for each packet transmission. More importantly, allocation of CPU and link bandwidth is closely coupled in O2. This coupling can potentially lower the utilization of the link and, as demonstrated in Section 5, significantly reduces channel admissibility while making it unpredictable.

3.1.3 Relationship Between CPU and Link Bandwidth

A conservative estimate of the message service time can be obtained by simply adding the total CPU processing time and the total link transmission time. However, this ignores the overlap between CPU processing and link transmission of packets constituting the same message. The extent of this overlap depends largely on the relationship between the CPU and link bandwidth, i.e., on the relative speed of the two. In order to improve channel admissibility, the message service time must be calculated to account for this overlap. The extent of the overlap also depends on the implementation option used for link scheduling. While O1 allows link utilization to be kept relatively high, O2 can cause the link to be kept idle even when there are packets available for transmission. From another perspective, O2 forces link scheduling to be non-work-conserving while O1 allows for work-conserving transmission of packets.

Note that while admission control can utilize the overlap between CPU processing and link transmission of packets belonging to a message, it cannot do so for the potential overlap between CPU processing and link transmission of packets belonging to *different* messages. This is because message arrivals serve as renewal points for the system and no *a priori* assumptions can be made about the presence of messages in the system.

3.1.4 Determination of $\mathcal{L}_{xmit}$

As mentioned earlier, the packet transmission time $\mathcal{L}_{xmit}(\mathcal{S})$ for a packet of size $\mathcal{S}$ measures the delay between initiation and completion of packet transmission on the network adapter; it determines the minimum time between successive packet transmission invocations by the link scheduler. For a typical network adapter, this delay depends primarily on two aspects, namely, the overhead of initiating transmission and the time to transfer the packet to the adapter and on the link. The latter is a function of the packet size and the data transfer bandwidth available between the host memory and the adapter. If $\mathcal{C}_{startup}$ is the overhead to initiate transmission on an adapter feeding a link of bandwidth $\mathcal{B}_{medium}$ bytes/second, then the packet transmission time can be approximated as

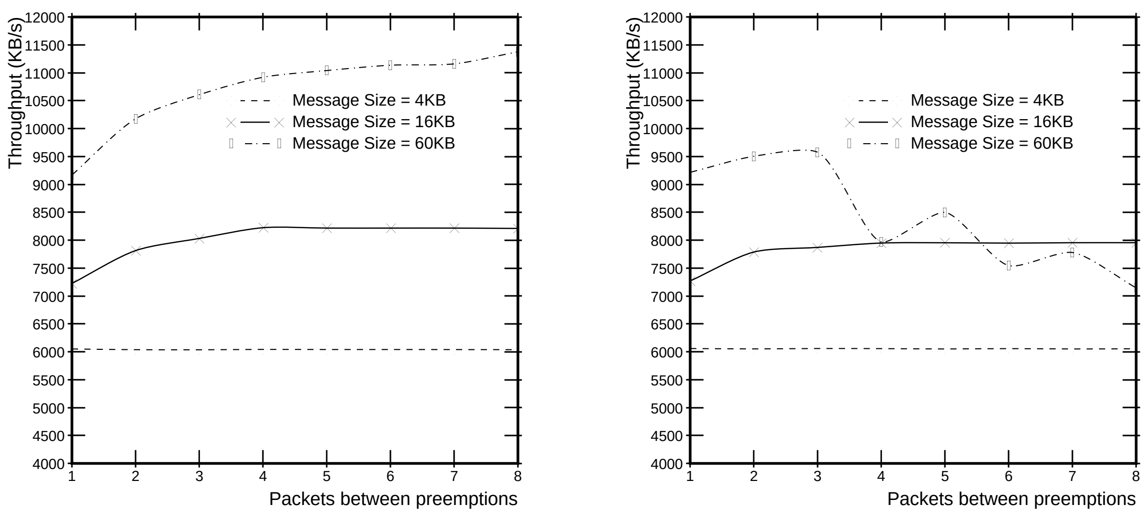
$$\mathcal{L}_{xmit}(\mathcal{S}) = \mathcal{C}_{startup} + \frac{\mathcal{S}}{\min(\mathcal{B}_{medium}, \mathcal{B}_{xfer})},$$

where $\mathcal{B}_{xfer}$ is the data transfer bandwidth available to/from host memory. $\mathcal{B}_{xfer}$ is determined by a variety of factors including the mode (direct memory access (DMA) or programmed IO) and efficiency of data transfer and the degree to which packet transmissions can be pipelined on the adapter. $\mathcal{C}_{startup}$ includes the cost of setting up any DMA transfer operations, if any. Note that with non-preemptive packet transmissions on the link, $\mathcal{L}_{xmit}(\mathcal{S})$ is also the delay experienced by a waiting highest-priority packet to commence transmission.

In order for this model of packet transmission time to be useful, $\mathcal{C}_{startup}$ and $\mathcal{B}_{xfer}$ must be determined for a given network adapter and host architecture. This involves experimentally determining the latency-throughput characteristics of the adapter. Since we want to explore the effects of the relationship between CPU and link bandwidth, we select $\min(\mathcal{B}_{medium}, \mathcal{B}_{xfer})$ to conform to a desired link (and data transfer) speed, measured in nanoseconds required to transfer one byte (ns per byte). On the null device, $\mathcal{C}_{startup}$ is determined by the granularity of time-keeping and the overhead of communication with the host processor; the measured value of $\mathcal{C}_{startup}$ is $\approx 40\ \mu s$.

3.2 Performance Implications

To illustrate the performance implications of some of the above-mentioned issues, we ran experiments using our real-time channel implementation to measure system throughput (kilobytes(KB)/second) as a function of the number of packets processed between preemption points, the packet size, and link speed. For a given CPU processing power, varying the speed of the link allows us to explore the relationship between CPU and link bandwidth. In all the experiments reported here, four best-effort channels were created and messages generated on these channels continuously.



(a) Option O1 (b) Option O2

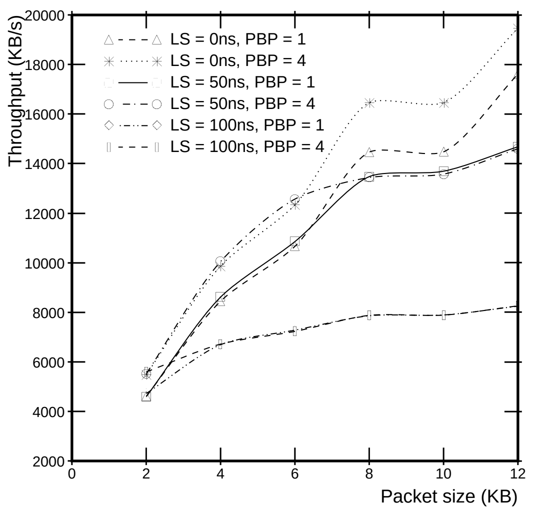
Figure 1: Throughput as a function of packets processed between preemption points

Figure 1 shows system throughput (useful resource capacity) as a function of the number of packets processed between preemption points (PBP) for several message sizes. The packet size is fixed at 4 KB and the link speed is set at 0 *ns* per byte, i.e., the link is "fast" relative to the CPU. For option O1 (Figure 1(a)), changing PBP has no effect when the message size is 4 KB; this is expected for single-packet messages. As message size increases, so do the number of packets and throughput increases until PBP becomes equal to the number of packets in the message. After this point PBP has no effect on throughput. As can be seen, for large messages an increase in PBP improves the throughput significantly and consistently.

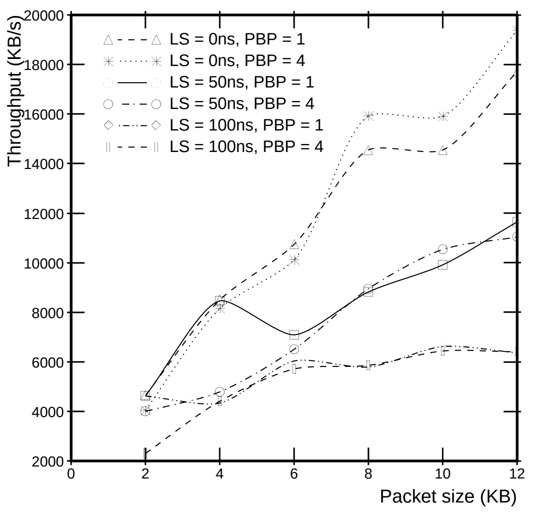
Option O2 (Figure 1(b)) reveals the same behavior as O1 for small- to medium-sized messages. However, for large messages throughput rises initially as PBP increases. Subsequently, the throughput starts falling sharply, but in a non-linear fashion. The general decline in throughput is due to increasingly poor utilization of the link bandwidth and a corresponding increase in the time to transmit all the packets belonging to the message. The oscillations in throughput are due to the subtle interactions between the CPU preemption window and link transmission, and will be investigated in Section 4.

Figure 2 shows the measured system throughput as a function of packet size and link speed, with option O1 (2(a)) and option O2 (2(b)). Three values of link speed (LS) are considered: 0 *ns* per byte (fast link), 50 *ns* per byte (medium speed link), and 100 *ns* per byte (slow link). For each value of link speed we consider two values of PBP, namely, 1 and 4. The message size is kept fixed at 32 KB and the packet size is varied from 2 KB to 12 KB.

Consider Figure 2(a) showing the system throughput for O1. We can again notice that, with a fast link (when the CPU is the bottleneck), increasing PBP from 1 to 4 provides a significant gain in throughput. For a given value of PBP and link speed, throughput increases with packet size since the CPU processing time reduces due to a reduction in the number of packets constituting the message. An increase in packet size from 8 KB to 10 KB does not change the number of packets, and hence the throughout remains unchanged. As the link becomes slower, however, there is a



(a) Option O1



(b) Option O2

Figure 2: Throughput as a function of packet size and link speed

saturation in the achieved throughput due to the link tending to become a bottleneck. After a certain packet size, for a given link speed, link transmission time exceeds the protocol processing time; thus any gains from a higher PBP cease to matter and the two curves converge. From Figure 2(a) we can be seen that this occurs at a packet size of 8 KB for link speed of 50 *ns* per byte and at 4 KB for link speed of 100 *ns* per byte.

Figure 2(b) shows the system throughput for O2. The trends are similar to those observed above when the link is either very fast (CPU is the bottleneck) or very slow (link is the bottleneck) because CPU and link processing can overlap almost completely. For a medium speed link (CPU and link bandwidths are more balanced), one can see that the throughput behavior is more non-linear. This is due to subtle interactions between CPU preemption window and the link transmission time, which causes the link to stay idle until the next preemption point. This explains the drop in throughput at a packet size of 6 KB and PBP of 1. Subsequently the throughput climbs again because link utilization improves and CPU requirements continue to decrease. This effect is analyzed in Section 4.

4 Computing Worst-Case Service and Wait Times

For a channel requesting admission, `D_order` can compute the worst-case response time for a message (referred to as the *system time requirement* in [1]) by accounting for three components:

- the worst-case waiting time for the message ($\mathcal{T}_{wait}$) due to lower-priority handlers or packets,
- the worst-case service time for the message ($\mathcal{T}_{service}$), and
- the worst-case waiting time due to servicing of message arrivals on all the existing higher-priority channels ($\mathcal{T}_{total}^{highpri}$).

Symbol	Description
$\mathcal{C}_{switch}$	time to switch contexts between channel handlers
$\mathcal{C}_{cache}$	penalty due to cache misses resulting from a context switch
$\mathcal{P}$	number of packets processed between preemption points
$\mathcal{C}_{prot}$	per-packet protocol processing cost
$\mathcal{C}_{link}$	per-packet link scheduling cost
$\mathcal{S}$	packet size in bytes
$\mathcal{L}_{xmit}(\mathcal{S})$	link transmission time for packet of size $\mathcal{S}$

Table 1: Important system parameters

We show below how $\mathcal{T}_{wait}$ and $\mathcal{T}_{service}$ can be estimated to account for the implementation-related issues highlighted above; $\mathcal{T}_{total}^{highpri}$ can then be recomputed using $\mathcal{T}_{service}$.

Suppose the CPU is reallocated to a waiting channel handler, if needed, every $\mathcal{P}$ packets; that is, up to $\mathcal{P}$ packets are processed between successive preemption points. Further, suppose that the packet size is $\mathcal{S}$, the context switch overhead between channel handlers is $\mathcal{C}_{switch}$, the cache miss penalty due to a context switch is $\mathcal{C}_{cache}$, and the packet transmission time is $\mathcal{L}_{xmit}(\mathcal{S})$ for packet size $\mathcal{S}$. The per-packet protocol processing cost is $\mathcal{C}_{prot}$ and the per-packet (link) scheduling overhead of picking a packet for transmission is $\mathcal{C}_{link}$. These system parameters are listed in Table 1.

4.1 Estimating Service Time

Consider a message of size $\mathcal{M}$ bytes such that it has $\mathcal{N}_{pkts} = \lceil \frac{\mathcal{M}}{\mathcal{S}} \rceil$ packets, with $\mathcal{N}_{pkts} - 1$ packets of size $\mathcal{S}$ and the last packet of size $\mathcal{S}^{last} = (\mathcal{M} \bmod \mathcal{S})$ if $(\mathcal{M} \bmod \mathcal{S}) \neq 0$, else $\mathcal{S}^{last} = \mathcal{S}$. Accordingly, the link transmission time is $\mathcal{L}_{xmit}(\mathcal{S})$ for all but the last packet and $\mathcal{L}_{xmit}(\mathcal{S}^{last})$ for the last packet. The protocol processing cost for the first packet is $\mathcal{C}_{prot}^{first}$ while subsequent fragments each incur a lower cost $\mathcal{C}_{prot}$. $\mathcal{C}_{prot}^{first}$ includes the fixed cost of obtaining the message for processing, the cost of a timestamp, and the cost of preparing the first packet.[4] Both $\mathcal{C}_{prot}^{first}$ and $\mathcal{C}_{prot}$ include the cost of network-level encapsulation. We estimate $\mathcal{T}_{service}$ for O1 and O2 separately.

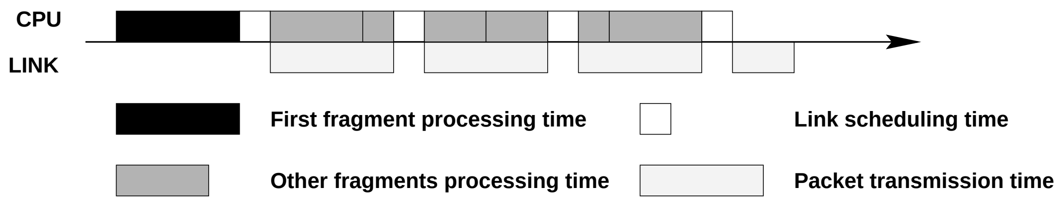
Option O1: Given the system parameters listed in Table 1, the worst-case service time for O1, $\mathcal{T}_{service}^{O1}$ is given by

$$\mathcal{T}_{service}^{O1} = \begin{cases} \mathcal{C}_{prot}^{first} + \mathcal{L}_{xmit}^{msg} + \mathcal{C}_{link}^{msg} + \mathcal{C}_{preempt} & \text{if } \mathcal{C}_{prot} < \mathcal{L}_{xmit}(\mathcal{S}) \\ \mathcal{C}_{prot}^{msg} + \mathcal{C}_{link}^{msg} + \mathcal{L}_{xmit}(\mathcal{S}^{last}) + \mathcal{C}_{preempt} & \text{otherwise} \end{cases}$$

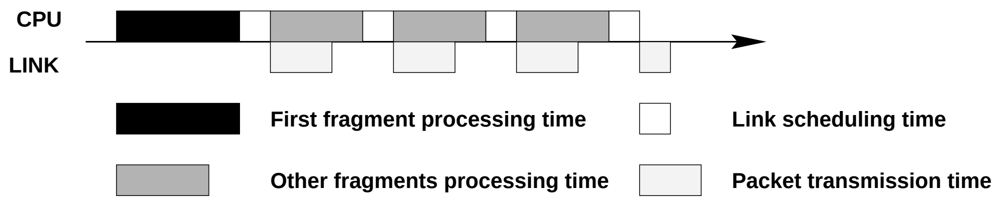
where $\mathcal{C}_{prot}^{msg} = \mathcal{C}_{prot}^{first} + (\mathcal{N}_{pkts} - 1)\mathcal{C}_{prot}$ represents the total protocol processing cost for the message, $\mathcal{C}_{preempt} = (\frac{\mathcal{N}_{pkts}-1}{\mathcal{P}})(\mathcal{C}_{switch} + \mathcal{C}_{cache})$ represents the total cost of preemption during the processing of the message, $\mathcal{C}_{link}^{msg} = \mathcal{N}_{pkts}\mathcal{C}_{link}$ represents the total link scheduling overhead for the message, and $\mathcal{L}_{xmit}^{msg} = (\mathcal{N}_{pkts} - 1)\mathcal{L}_{xmit}(\mathcal{S}) + \mathcal{L}_{xmit}(\mathcal{S}^{last})$ represents the total link transmission time for the message.

If $\mathcal{C}_{prot} < \mathcal{L}_{xmit}(\mathcal{S})$, there is at least the cost of processing the first packet. Since the link transmission time dominates the time to process subsequent packets (see Figure 3(a)), the message

[4]Our fragmentation protocol traverses a slower path for messages larger than S bytes and hence the higher processing cost for the first packet.


CPU
LINK
First fragment processing time
Link scheduling time
Other fragments processing time
Packet transmission time

(a) $\mathcal{C}_{prot} < \mathcal{L}_{xmit}(\mathcal{S})$


CPU
LINK
First fragment processing time
Link scheduling time
Other fragments processing time
Packet transmission time

(b) $\mathcal{C}_{prot} \geq \mathcal{L}_{xmit}(\mathcal{S})$

Figure 3: Protocol processing and link transmission overlap in O1

service time is determined by the link transmission time for the message and the total link scheduling and preemption overheads incurred. If $\mathcal{C}_{prot} \geq \mathcal{L}_{xmit}(\mathcal{S})$ (Figure 3(b)), however, the message service time corresponds to the total protocol processing time for the message plus the time to transmit the last packet, in addition to the total link scheduling and preemption overheads.

Option O2: Calculation of the worst-case service time for O2, $\mathcal{T}^{O2}_{service}$, is done similarly; however, we must now consider the processing of *blocks* of packets with each block comprising no more than $\mathcal{P}$ packets. The number of blocks in a message with $\mathcal{N}_{pkts}$ packets is given by $\mathcal{N}_{blocks} = \lfloor \frac{\mathcal{N}_{pkts}-1}{\mathcal{P}} \rfloor + 1$. The protocol processing cost for the first block is given by $\mathcal{C}^{first}_{block} = \mathcal{C}^{first}_{prot} + (\max(\mathcal{N}_{pkts}, \mathcal{P}) - 1)\mathcal{C}_{prot}$, while the cost of processing the last block of packets is given by

$$\mathcal{C}^{last}_{block} = \begin{cases} \mathcal{C}_{block} & \text{if } (\mathcal{N}_{pkts} \bmod \mathcal{P}) = 0 \\ (\mathcal{N}_{pkts} \bmod \mathcal{P})\mathcal{C}_{prot} + \mathcal{C}_{cache} + \mathcal{C}_{switch} & \text{otherwise} \end{cases}$$

where $\mathcal{C}_{block} = \mathcal{P}\mathcal{C}_{prot} + \mathcal{C}_{cache} + \mathcal{C}_{switch}$ is also the cost of processing the other blocks, if any. The worst-case service time for O2, $\mathcal{T}^{O2}_{service}$, is given by

$$\mathcal{T}^{O2}_{service} = \begin{cases} \mathcal{T}^{A} + \mathcal{T}^{O2,cpu}_{wait} & \text{if } \mathcal{C}_{block} < \mathcal{L}_{xmit}(\mathcal{S}) \\ \mathcal{T}^{B} & \text{otherwise} \end{cases}$$

where

$$\mathcal{T}^{A} = \mathcal{C}^{first}_{block} + \mathcal{L}^{msg}_{xmit} + \mathcal{C}^{msg}_{link}$$

and

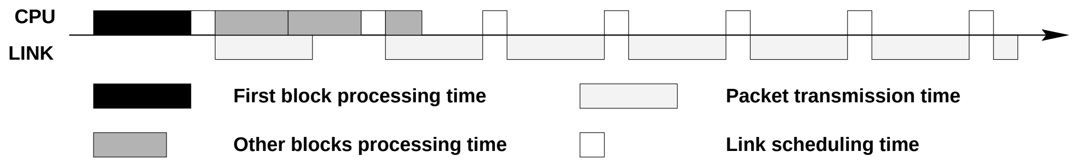
$$\mathcal{T}^{O2,cpu}_{wait} = \mathcal{C}^{first}_{prot} + (\mathcal{P} - 1)\mathcal{C}_{prot} + \mathcal{C}_{cache} + \mathcal{C}_{switch} + \mathcal{C}'_{link},$$

with $\mathcal{C}'_{link} = \mathcal{C}_{link} + \mathcal{C}_{cache} + \mathcal{C}_{switch}$ in the case of O2. $\mathcal{T}^{B}$ is given by

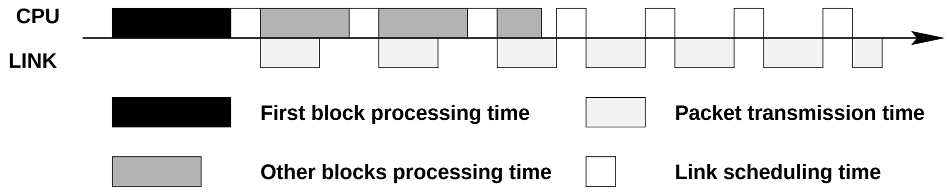
$$\mathcal{T}^{B} = \begin{cases} \mathcal{T}^{A} & \text{if } \mathcal{N}_{blocks} = 1 \\ \mathcal{T}^{C}_{a} + \mathcal{T}^{C}_{b} & \text{otherwise} \end{cases}$$

where

$$\mathcal{T}^{C}_{a} = \mathcal{C}^{first}_{block} + (\mathcal{N}_{blocks} - 2)\mathcal{C}_{block} + \max(\mathcal{C}^{last}_{block}, \mathcal{L}_{xmit}(\mathcal{S}))$$



(a) $\mathcal{C}_{block} < \mathcal{L}_{xmit}(\mathcal{S})$



(b) $\mathcal{C}_{block} \geq \mathcal{L}_{xmit}(\mathcal{S})$

Figure 4: Protocol processing and link transmission overlap in O2

and

$$\mathcal{T}_b^C = (\mathcal{N}_{pkts} - \mathcal{N}_{blocks})\mathcal{L}_{xmit}(\mathcal{S}) + \mathcal{L}_{xmit}(\mathcal{S}^{last}) + \mathcal{C}_{link}^{msg}.$$

If $\mathcal{C}_{block} < \mathcal{L}_{xmit}(\mathcal{S})$ (see Figure 4(a)), each block will complete processing its packets before a packet gets transmitted. This is because in O2, the link scheduler does not get to run until the CPU is preempted. So the message service time simply corresponds to the time to process the first block of packets and transmit each packet. However, the link scheduler may now have to wait for time $\mathcal{T}_{wait}^{O2,cpu}$ to obtain access to the CPU and initiate packet transmission; in the worst-case this wait penalty must be assumed for each packet that is to transmitted. If $\mathcal{C}_{block} \geq \mathcal{L}_{xmit}(\mathcal{S})$, however, the link transmission time is faster than the time to process the packets in a block, as shown in Figure 4(b). Except for the first block, each block will overlap precisely one packet transmission. This implies that message service time is determined by the processing time for all the blocks, the transmission time of $(\mathcal{N}_{pkts} - \mathcal{N}_{blocks})$ packets, and the total link scheduling overhead. There is no wait time in this case because the link scheduler is guaranteed to run next, even if the handler must preempt the CPU to another handler. If the last block of packets is shorter than $\mathcal{L}_{xmit}(\mathcal{S})$, only the time to transmit the last packet contributes to the message service time; else the time to process the last block must be considered.

4.2 Estimating Wait Time

In order to estimate $\mathcal{T}_{wait}$ we need to calculate the total wait time experienced by a message due to execution of lower-priority handlers and non-preemptive transmission of lower-priority packets. We first consider the time spent waiting for a lower-priority handler to relinquish the CPU. Subsequently we consider the time spent waiting for the link. As before, we consider O1 and O2 separately.

Option O1: The worst-case processing time for a block of packets is $\mathcal{C}_{block}^{first}$. During this time up to $\lceil \frac{\mathcal{C}_{block}^{first}}{\mathcal{L}_{xmit}(\mathcal{S})} \rceil$ packets could complete transmission. Therefore, the worst-case CPU wait time for O1 is given by

$$\mathcal{T}_{wait}^{O1,cpu} = \mathcal{C}_{block}^{first} + (\lceil \frac{\mathcal{C}_{block}^{first}}{\mathcal{L}_{xmit}(\mathcal{S})} \rceil)\mathcal{C}_{link} + \mathcal{C}_{cache} + \mathcal{C}_{switch}.$$

Now consider the worst-case link wait time. A lower-priority transmission could be started in an ISR just before the currently executing handler makes a packet ready for transmission. Thus, the packet generated by the handler will have to wait for the lower-priority packet to complete transmission. Therefore, the worst-case link wait time is simply $\mathcal{T}_{wait}^{O1,link} = \mathcal{L}_{xmit}(\mathcal{S})$, and $\mathcal{T}_{wait}^{O1} = \mathcal{T}_{wait}^{O1,cpu} + \mathcal{T}_{wait}^{O1,link}$.

Option O2: Under O2, on the other hand, the worst-case CPU wait time corresponds to the time to process up to $\mathcal{P}$ packets of a message on a lower-priority channel followed by a context switch to the link scheduler, followed by another context switch before the waiting handler gets to run. Thus,

$$\mathcal{T}_{wait}^{O2,cpu} = \mathcal{C}_{prot}^{first} + (\mathcal{P} - 1)\mathcal{C}_{prot} + \mathcal{C}_{cache} + \mathcal{C}_{switch} + \mathcal{C}_{link}^{'},$$

where $\mathcal{C}_{link}^{'} = \mathcal{C}_{link} + \mathcal{C}_{cache} + \mathcal{C}_{switch}$ in the case of O2. Similarly, $\mathcal{T}_{wait}^{O2,link}$ is derived as follows. If $\mathcal{L}(\mathcal{S}) <= \mathcal{C}_{prot}^{first}$, $\mathcal{T}_{wait}^{O2,link} = 0$; else $\mathcal{T}_{wait}^{O2,link} = \mathcal{T}_{wait}^{O2,cpu}$. This can be explained as follows. If the link scheduler completed transmission before the first block is processed, the next packet can be transmitted as soon as the handler yields the CPU. In the worst case, a lower-priority handler begins processing a non-preemptive block of packets, making the link scheduler wait for this block to end before it gets a chance to run and transmit the next higher-priority packet. Thus $\mathcal{T}_{wait}^{O1} = \mathcal{T}_{wait}^{O2,cpu} + \mathcal{T}_{wait}^{O2,link}$.

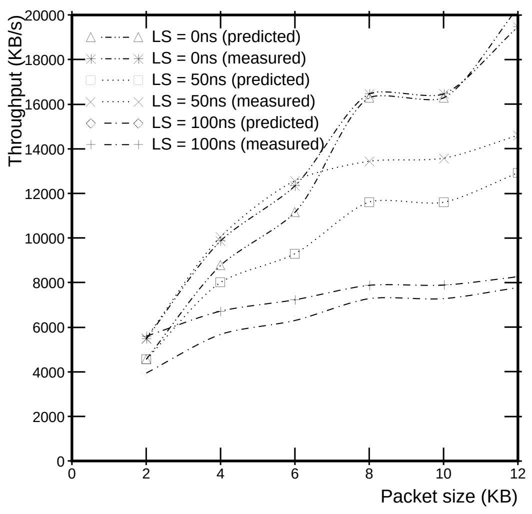
4.3 Validation

Our implementation of real-time channels provides admission control based on the above estimates of service and wait times. While these estimates are geared towards real-time guarantees, and therefore are necessarily conservative, it would be insightful to compare the throughput predicted by these estimates and the best-effort throughput measured using the real-time channel implementation. In the experiments reported here, we use system parameter values obtained from an extensive parameterization of the communication subsystem, including the protocol stack [3].

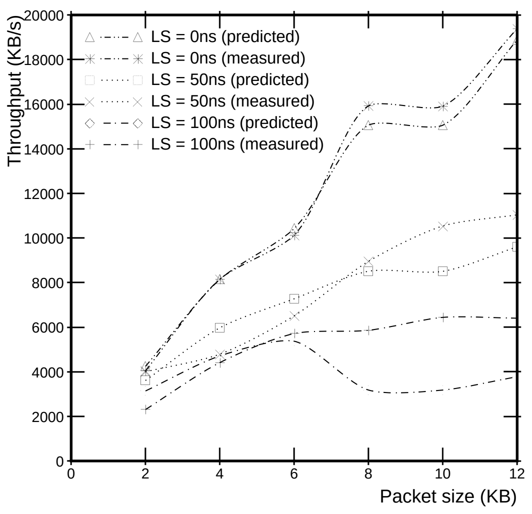
Figure 5 presents a comparison of the predicted and measured throughput as a function of packet size, for three values of link speed (LS) as before. Consider option O1 (Figure 5(a)). As can be seen, the predicted throughput reveals roughly the same trends as those of the measured throughput. While the predicted throughput tracks the measured throughput well when the link is either fast or slow, it diverges significantly for intermediate link speeds. We believe that this divergence is due to the overly conservative estimates used for system parameters such as context switch overhead and cache miss penalty. Further, the estimates are necessarily conservative in accounting for worst-case wait times which, though necessary for real-time traffic, may be relatively small on average.

Similar trends are observed with option O2 as shown in Figure 5. Again the predicted throughput tracks the measured throughput well. However, when the link is slow (link speed = 100 *ns*) the predicted throughput becomes very conservative after a packet size of 6 KB. Once again this is due to the incorporation of worst-case delays that are not encountered on average.

These validation experiments reveal certain shortcomings in the values assigned to the system parameters listed in Table 1; part of the discrepancy stems from the unpredictability introduced by caches. More refined experiments are necessary in order to select accurate values for system parameters such as $\mathcal{C}_{prot}$ and $\mathcal{C}_{cache}$.



(a) Option O1



(b) Option O2

Figure 5: Comparison of measured and predicted throughput

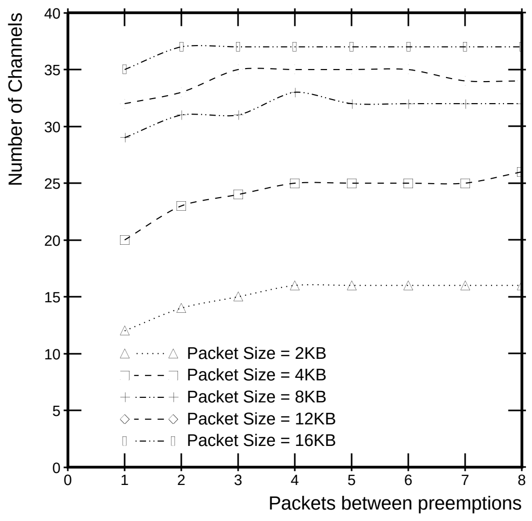
5 Implications for Channel Admissibility

In this section we explore the tradeoff between resource capacity and channel admissibility, and demonstrate that this tradeoff is influenced significantly by $\mathcal{P}$, the number of packets between preemptions, and $\mathcal{S}$, the packet size. As expected, the mechanism employed to implement link scheduling and the relationship between CPU and link bandwidth also have a profound effect on channel admissibility. We studied channel admissibility for O1 and O2 for a range of link speeds, message sizes, rates and deadlines. In the following we present and compare the results for a link speed of 50 ns per byte, message size of 32 KB, and message inter-arrival of 100 ms. We admit as many channels as possible with deadline of 100 ms.

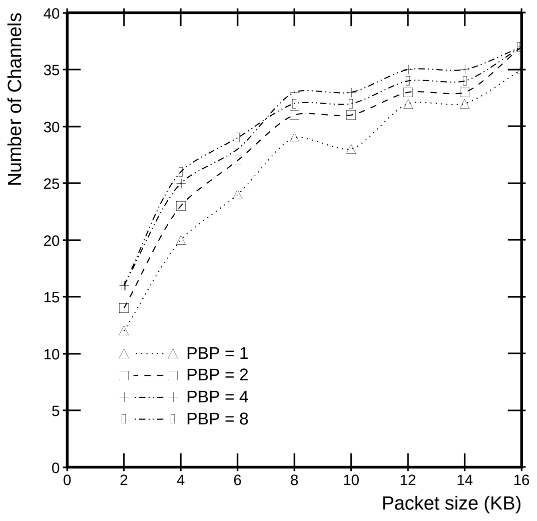
5.1 Channel Admissibility in O1

Consider Figure 6(a). The channel admissibility in O1 rises with both $\mathcal{P}$ and $\mathcal{S}$. This is because of the accompanying reduction in protocol processing cost and work-conserving transmission of packets on the link. As $\mathcal{P}$ rises, protocol processing costs decline, resulting in a small increase in channel admissibility. As $\mathcal{P}$ continues to rise, the marginal benefits in protocol processing costs decline. Due to an increase in the window of non-preemptibility, channel admissibility either saturates or shows a small decline. Figure 6(b) illustrates that increasing $\mathcal{S}$ increases channel admissibility substantially, since the reduction in the required CPU bandwidth more than compensates for the increase in the non-preemptability window.

The above results might suggest that allowing arbitrary sized packets, i.e., sending each message as a single packet, is desirable to maximize channel admissibility. While this would be true if all channels carried same-sized messages, the same cannot be said for channels with smaller (single-packet) messages. Increasing $\mathcal{P}$ and $\mathcal{S}$ arbitrarily only serves to increase the window of non-preemptability with no reduction in CPU processing requirements for small messages. Large values



(a) Effect of $\mathcal{P}$



(b) Effect of $\mathcal{S}$

Figure 6: Effect of $\mathcal{P}$ and $\mathcal{S}$ on channel admissibility in O1

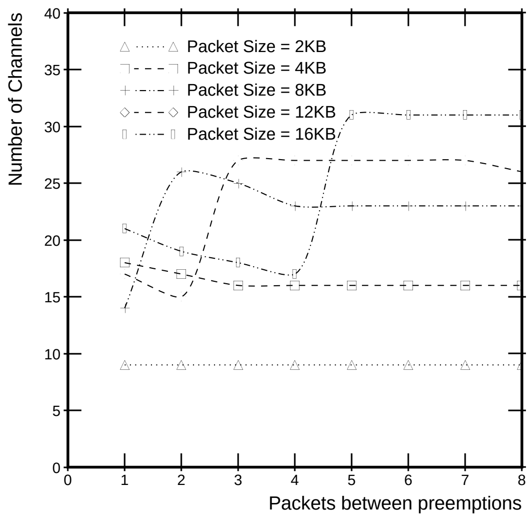
of $\mathcal{P}$ and $\mathcal{S}$ tend to lower the admissibility of channels with small messages, especially those with tight deadlines. Selection of $\mathcal{P}$ and $\mathcal{S}$ therefore depends not only on the system parameters, but also the targetted mix of communication traffic.

5.2 Channel Admissibility in O2

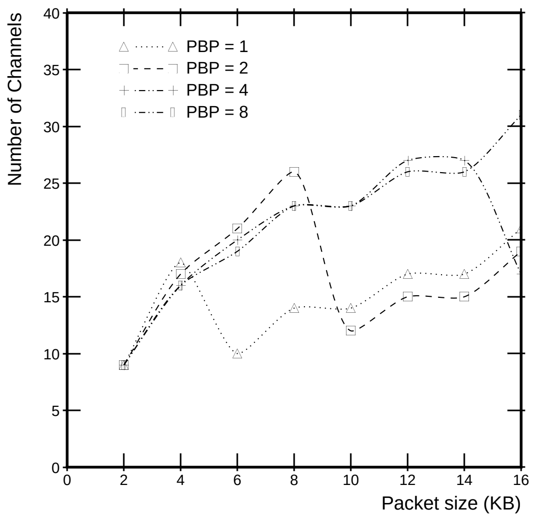
Figure 7 (a) shows the effect of $\mathcal{P}$ on channel admissibilty when using O2 to schedule messages. In contrast to O1, the channel admissibility behavior is highly non-linear. This can be easily explained using our preemption model. Consider, for instance, the case with 8 KB packets. With the given link speed and $\mathcal{P}$ equal to 1, link transmission time is greater than the protocol processing time for a block of packets. The model in Figure 4(a) applies, making the channel susceptible to long idle periods, as discussed in Section 4. When $\mathcal{P}$ is increased to 2, the transmission time for a packet remains unchanged, but the processing time for a preemption block increases, making it more than the link transmission time. This results in the scenario in Figure 4(b), in which the worst-case transmission time is reduced substantially, thereby increasing the number of channels admitted.

As $\mathcal{P}$ increases further, there is no further change in the nature of the overlap between CPU processing and link transmission. The number of channels admitted either remains unchanged or declines slightly due to an increase in the window of non-preemptability. This transition occurs for all but the smallest packet sizes. In general, the larger the packet, the greater the value of $\mathcal{P}$ that causes a change in the nature of overlap, namely, from packet transmission time being slower to it being faster than the processing time for the longest block of packets.

Figure 7 (b) presents the same information, but as a function of $\mathcal{S}$. As packet length increases, there is an initial increase in admissibility due to reduced protocol processing load. At a certain value of $\mathcal{S}$, the link transmission time becomes larger than the block processing time, thus changing the scenario from that in Figure 4(b) to that in Figure 4(a). Further increase in packet size brings a slow rise in channel admissibility, due to reduced CPU bandwidth requirements by the



(a) Effect of $\mathcal{P}$ (PBP)



(b) Effect of $\mathcal{S}$

Figure 7: Effect of $\mathcal{P}$ and $\mathcal{S}$ on channel admissibility in O2

channels. As seen from Figure 7, the best operating point for O2 depends critically on system parameters. Since a change in channel characteristics will significantly change channel admissibility, a system parameterized and optimized for a particular workload is unlikely to perform well under a heterogenous workload.

Using a model of ideal resources, i.e., with no CPU preemption cost and an immediately preemptible CPU, we found $\approx$ 40% improvement in channel admissibility over and above O1 with $\mathcal{P} = 1$. This shows that it is necessary to account for non-ideal characteristics, such as context switch overhead and cache miss penalty, of real systems.

6 Related Work

This paper is an extension of the policies proposed in [1], with a focus on CPU and link bandwidth management for admission control. We have implemented a QoS-sensitive architecture [3] that uses the extensions described here to provide admission control and run-time support for real-time channels.

Our implementation methodology and analysis is also 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 [2]. Similar issues are being explored in the context of providing integrated services on the Internet [7–9]. The Tenet real-time protocol suite [10] is an advanced implementation of real-time communication on wide-area networks (WANs); however, they have not considered incorporation of protocol processing overheads into the network-level resource management policies. In particular, they do not address the problem of making protocol processing inside the host QoS-sensitive.

The implications of priority inversion due to non-preemptible critical sections was studied in [11]; however, the costs of preemption (context switches and cache misses) and the resulting degradation in useful resource capacity were not considered. The need for scheduling protocol processing at priority levels consistent with those of the communicating application was highlighted in [12] and some implementation strategies demonstrated in [13]. More recently, processor capacity reserves in Real-Time Mach [14] have been combined with user-level protocol processing [15] to make protocol processing inside hosts predictable [16]. Operating system support for multimedia communication is explored in [17] and [18]. In [17] the focus is on provision of preemption points and earliest-deadline-first scheduling in the kernel. Similarly, the focus of [18] is on the scheduling architecture.

7 Conclusion and Future Work

In this paper, we have focused on the management of host communication resources for real-time communication. In particular, we identified the issues involved in extending and implementing resource management policies originally formulated using idealized resource models. Using our implementation of real-time channels, we extended the admission control procedure to account for protocol processing and implementation overheads. The extensions were validated against measured performance of the implementation and used to study the implications for channel admissibility.

Our main conclusions can be summarized as follows. In order to best utilize CPU and link bandwidth, one has to account for implementation overheads that reduce the useful (available) resource capacity. Further, one must also consider the implications of the implementation paradigm adopted to manage CPU and link bandwidth. We studied two implementation paradigms that both realize link scheduling but differ significantly in performance. The realization of the link scheduler as a dedicated process results in a significant degradation in channel admissibility. Since it couples allocation of link bandwidth with allocation of CPU bandwidth, this paradigm suffers from conservative admission control. Implementing link scheduling in interrupt context provides significant performance advantages in terms of higher channel admissibility and reduced sensitivity to system parameters such as packet size and number of packets processed between preemptions. However, these advantages come at the expense of an unpredictable increase in the interrupt service time, which could be highly undesirable in systems with stringent response time constraints.

While our present work has been done in the context of real-time channels, it is applicable to a wide variety of proposals for real-time communication and QoS guarantees [2]. The issues of simultaneous management of CPU and link bandwidth to support real-time communication are of wide-ranging interest. Many of the tradeoffs we highlighted are applicable to other machines as well. More importantly, the extensions provided to the real-time channel model are general and are applicable to other host and network architectures. While we only considered management of communication resources, the present work can be extended to incorporate application scheduling as well. Our analysis is directly applicable if a portion of the host processing capacity can be reserved for communication-related activities [14,16].

As part of future work, we plan to continue with more extensive validation of the proposed extensions to the real-time channel model. This would involve careful parameterization of the communication subsystem and incorporation of the results into the service and wait time calculations. Our extensions were based on fairly conservative assumptions about the worst-case scenarios. We are exploring the possibility of relaxing some of these assumptions without compromising real-time guarantees. Lastly, we plan to extend our null device into a more sophisticated network device emulator. We plan to use the emulator to study various tradeoffs associated with provision of real-time services in communication subsystems.

References

[1] 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.

[2] 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.

[3] A. Mehra, A. Indiresan, and K. Shin, "Design and evaluation of a QoS-sensitive communication subsystem architecture," Technical Report CSE-TR-280-96, University of Michigan, January 1996.

[4] 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.

[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] 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.

[7] 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, August 1992.

[8] R. Braden, D. Clark, and S. Shenker, "Integrated services in the Internet architecture: An overview," *Request for Comments RFC 1633*, July 1994. Xerox PARC.

[9] S. Floyd and V. Jacobson, "Link-sharing and resource management models for packet networks," *IEEE/ACM Trans. Networking*, vol. 3, no. 4, , August 1995.

[10] 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.

[11] C. W. Mercer and H. Tokuda, "Preemptibility in real-time operating systems," in *Proceedings of the 13th IEEE Real-Time Systems Symposium*, December 1992.

[12] 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.

[13] 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.

[14] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[15] 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.

[16] 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.

[17] O. Hagsand and P. Sjodin, "Workstation support for real-time multimedia communication," in *Winter USENIX Conference*, pp. 133–142, January 1994. Second Edition.

[18] C. Vogt, R. G. Herrtwich, and R. Nagarajan, "HeiRAT: The Heidelberg resource administration technique design philosophy and goals," Research Report 43.9213, IBM Research Division, IBM European Networking Center, Heidelberg, Germany, 1992.