Implementing Traffic Shaping and Link Scheduling on a High-Performance Server *

Sung-Whan Moon
C-EISA
Samsung-dong 169-6
Seoul, South Korea
swmoon@c-eisa.com

Kang G. Shin
Real-Time Computing Lab.
University of Michigan
Ann Arbor, MI 48105
kgshin@eecs.umich.edu

Abstract

This paper examines the implementation of traffic shaping and link scheduling mechanisms on an end-host server. We first compare various implementation strategies, and then present a network interface architecture with dedicated support for traffic shaping and link scheduling for use in end-host servers. This results in significant load reduction on the server CPU, as shaping and scheduling tasks can execute concurrently on the network interface. This allows the server to provide very fine-grain link multiplexing, and consequently more diverse set of QoS guarantees for a large number of flows. We present two alternative implementations of our architecture. The first uses custom hardware while the second is implemented as a software component running on a dedicated processor on the network interface.

1. Introduction

Recent advances in network technology have brought about substantial increases in bandwidth along with widespread Internet access through high-speed connections beyond the corporate and academic communities and into homes. This has resulted in the introduction of a large number of applications with a wide range of quality-of-service (QoS) requirements [9]. Representative of these new applications are those with real-time traffic, such as video and audio. This real-time communication [2, 30] requires guarantees such as bounded end-to-end delay, bounded cell-loss rates, and guaranteed bandwidth from the network. Today's packet-switched networks can employ a variety of methods to provide the QoS guarantees for present and future applications.

As a packet travels from its source towards its destination, contention for resources along its path will result in queueing of some packets while others get access to those resources. Assuming packets can be classified into their corresponding flow (classification is based on one or more fields in the packet headers), QoS mechanisms will service packets such that all flows receive their desired levels of QoS. One such mechanism is traffic shaping, which monitors and controls flows so that they abide by their specified traffic parameters. On the other hand, link schedulers multiplex among queued packets from different flows onto a single link for transmission. By combining these two mechanisms, flows can be guaranteed to receive service with bounds on delay, delay jitter, and bandwidth [29, 34]. These bounds will vary depending on which combination of algorithms are chosen for the shaping and scheduling.

However, delivering QoS guarantees requires an end-to-end solution [3]. In other words, traffic shaping and link scheduling mechanisms must be in place both within the network at switches and routers, and also at the flow's source (i.e., the server). Although most research focuses on implementation at nodes within the network, this paper focuses on implementing traffic shaping and link scheduling on an end-host server with a single outgoing link. Without a shaping and scheduling mechanism on the end-host server, the outgoing traffic pattern of a flow is determined by when the application is scheduled on the server CPU, and how much data the application is allowed to process and transmit at a time. During heavy server CPU loads and congestion at the outgoing link, the resulting flow's outgoing traffic pattern will not match the flow's specified traffic pattern. This deviation will be worse for flows which need to transmit relatively small amounts of data within short periods of time. In a high-end content server, which needs to process both web page content and deliver audio/video

*The work reported in this paper was supported in part by the National Science Foundation under Grant EIA-9806280. Any opinions, finding, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the NSF.

streams for real-time playback, such deviations can result in interrupted playback of real-time streams at end-clients. Although larger buffers and a longer startup time can reduce the effects of these deviations in traffic patterns, continuous deviations under heavy server loads will still result in playback interruptions. Assuming QoS mechanisms within the network, both clients' satisfaction (small startup time delay and no playback interruptions) and efficient use of network resources (buffers) will be maximized if the outgoing traffic pattern of each flow closely matches the stream's playback rate. This is important for content providers who must maximize the number of clients receiving streams from a server, and also maximize the number of clients receiving satisfactory service from the server.

Incorporating a shaper-scheduler on the server can eliminate traffic pattern deviations mentioned above. Unlike switches and routers which simply react to incoming packets, depending on how the shaper-scheduler is implemented on the server, the resulting implementation will either react to packets generated by applications running on the server, or will directly impact the generation of packets by applications. This direct impact can change the QoS received by each flow. To highlight these issues, and to motivate the need for a dedicated shaper-scheduler on the server, we will present and examine more traditional implementation strategies. By "traditional" we refer to software implementations, either at the operating system or application level, which run on the same server CPU(s).

This paper presents a network interface architecture with dedicated traffic shaping and link scheduling support. When a new flow is initiated, its traffic parameters are downloaded into the network interface through its programmable interface. Assuming that the application receives enough CPU cycles to process its stream into packets, the network interface will smooth the flow's stream, regardless of how bursty the actual processing packet is, and schedule transmission on the outgoing link so that all flows receive their desired QoS. This approach allows the shaper-scheduler to operate concurrently with the rest of the server, thereby reacting to the flows. The architecture also significantly reduces server CPU load during heavy loads (both CPU usage and number of flows being served). Very fine-grain link multiplexing, and consequently a more diverse set of QoS guarantees, can be easily supported in this architecture for a large number of flows.

The rest of the paper is organized as follows. We give a brief overview of traffic shaping and link scheduling in Section 2. A detailed examination of shaper-scheduler implementation issues on end-host servers is given in Section 3. In Section 4 we present our network interface architecture, and describe its operation within the framework of a streaming server. We also present two possible implementations of our architecture. Section 5 presents a performance evaluation of these two implementations, along with a brief description of our simulation environment and traffic models. Section 6 concludes the paper with a summary of our work and a brief list of future directions.

2. Background

There are numerous traffic shaping and link scheduling algorithms in the literature, each characterized by different QoS guarantee properties. Despite this difference, all these algorithms share a common framework upon which they can be mapped. Traffic shapers hold back newly-queued packets from being serviced until the packet's flow conforms to its traffic envelope. This implies marking each packet with a conformance (eligibility or start) time, and having the shaper move packets out of the shaper and into the scheduler queue only when the system time reaches the start time. Shapers differ in how they compute the start time. On the other hand, when the link becomes available, the link scheduler chooses the next packet to transmit among all eligible packets queued in the scheduler queue. This implies that packets are assigned a priority (also called *deadline* or *finish time*), with the scheduler choosing the packet with the highest priority. Scheduling algorithms simply differ in how they compute the priority. We illustrate these points by describing several well-known algorithms.

2.1. Leaky-bucket Shaper

This algorithm [12, 26, 30] is conceptually very simple and is the basis for most shapers in the literature. The shaper generates tokens for a flow i at a rate of ρ_i, where σ_i is the maximum number of tokens that can be accumulated in the bucket. A newly-arrived packet is eligible for transmission only if there are enough tokens in the bucket. Otherwise, it must wait in the shaper until enough tokens have accumulated. When a packet is eligible and moves into the scheduler, it grabs the necessary number of tokens from the bucket. In the case of fixed-sized packets (i.e., ATM cells), each token corresponds to a single packet. Assuming the k^{th} packet from flow i arrives at time $A(p_i^k)$, and requires L_i^k tokens, the packet's start time $S(p_i^k)$ (earliest time the packet is eligible) can be computed very easily. Since the shaper will serve packets from the same flow in FCFS order, the shaper need only keep one entry for each flow with one or more packets queued in the shaper. The packets can be ordered as a simple list. When the first, or head-of-the-line (HOL) packet becomes eligible, the start time for the flow's next queued packet is calculated by setting its arrival time as the start time of the previous packet. If there are packets queued in the shaper for flow i when a new packet arrives, the new packet is simply appended to the end of the list without the need to calculate its start time.

2.2. Link Scheduling

In a static priority algorithm each flow has a predetermined priority number associated with it. All packets belonging to the flow are marked with the same priority. The special case where there is only one priority level is the FCFS algorithm, where no packets have priority over others.

Under EDD [2, 30], each packet is assigned a due date (deadline), with the scheduler transmitting smallest deadline first. With these schemes each flow i provides the minimum packet interarrival time I_i and a local delay bound d for each node the packet passes in the network.

Packet-by-packet generalized processor sharing (PGPS) [25], also known as Weighted FQ (WFQ), Frame-based FQ (FFQ) [28, 33], Starting-potential based FQ (SPFQ) [33], Start-time FQ (SFQ) [20], Worst-case Fair WFQ (WF2Q) [6] and WF2Q+ [5], and Self-clocked FQ (SCFQ) [19] are several examples of packetized versions of fair queueing (FQ) algorithms [12, 25]. Despite the large number of variations, the basic foundation for all these algorithms is the same. The function $V(t)$ returns the system time (or system potential) at time t. For each packet that enters the scheduler, it is assigned a start time $S(p_i^k)$ and a finish time $F(p_i^k)$. Packets are then transmitted in increasing order of start time or finish time, depending on the algorithm. The difference among the various PFQ algorithms lies in how they compute the system, start, and finish times. Typically, each flow is specified only by its allocated rate ρ_i.

Other link scheduling algorithms include weighted round-robin [22], hiearchical round-robin [21], Stop and Go [31] (a framing strategy), Virtual Clock [35], and many others.

2.3. Framework

The common denominator among all these algorithms is the notion of stamping each packet with a number, with service order based on that number. Shapers queue packets and service them based on their start times, while the scheduler queues packets based on their priorities. Packets in the shaper queue can only be considered for transmission when they are eligible (i.e., the shaper moves the packet to the scheduler queue). Several researchers have proposed various shaper-scheduler implementations [11, 26, 29], whose results are incorporated into this paper. The basic framework of these implementations includes two sorted queues (shaper and scheduler), a mechanism for determining and moving eligible packets into the scheduler queue, and a control mechanism which computes the start and finish times of each packet. After each packet transmission, the shaper needs to compute the start times of any new HOL packets that have arrived and insert them into the shaper queue.

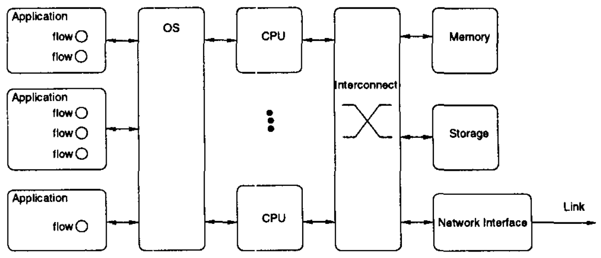


Figure 1. A generic server model

Next, the shaper must move any eligible HOL packets into the scheduler queue. If these HOL packets are not the last in the flow's linked list, then the new HOL packets must be processed by the shaper and inserted into the shaper queue. The scheduler then needs to determine and transmit the next packet.

3. Implementation Strategies

This section examines several traditional means of implementing a shaper-scheduler on an end-host server. We will discuss the mechanisms and limitations of these solutions. We first state assumptions regarding the server configuration.

3.1. Assumptions

Figure 1 shows a generic server that we assume in this paper. One or more processors are connected to secondary storage devices and the network interface through an interconnect. In most PC server configurations this is just a shared bus. However, as processors and network link speeds become faster, and larger storage devices become more readily available, servers will be capable of serving more and more requests. Based on this trend, the current shared bus will become a major bottleneck, and will be replaced by a faster and more efficient interconnect mechanism. Although we don't assume a specific type of interconnect, we do assume that the interconnect will be fast enough not to be the bottleneck.

We also assume that applications will receive enough CPU cycles for packet processing [1]. This implies a process scheduler on the OS, which is important for applications which need to process and transmit date at periodic intervals. The process scheduler sets an upper bound on the time an application must wait to access the CPU, and guarantees that the application can process the minimum amount of data the application will transmit during this time interval. Without this, any shaper-scheduler implementation will be made ineffective. We assume that each application is associated with one or more flows, and that each flow is specified by a set of traffic parameters (i.e., burst size and minimum burst interval). For example, a stored video-on-demand server application can transmit several different flows (i.e., video streams) to remote clients for real-time playback. Although the entire video file is stored on disk, streaming the file allows the user to start playback without waiting for the entire file to download. As video files tend to be very large, streaming results in more efficient use of network resources, and allows the server to deliver good performance to a larger number of clients. In this example, each video stream's traffic parameter is specified by its playback rate. Transport layer protocols such as TCP simply try to detect and avoid congestion within the network, and are not suitable for such streaming applications which require a specified spacing between each transmission burst.

3.2. Application-level traffic shaping

Applications, such as streaming servers which need to pace each flow at a specified rate, can take advantage of a traffic shaper. However, without any OS-level support, one solution is to have each application pace its own flows. This assumes either a FCFS or other link scheduler at the transmit queue. Such self-pacing requires that each application monitor each flow and hold back transmissions for a flow if it violates the flow's traffic parameters. Since a send call to the OS will attempt to transmit the data as soon as possible, the application can do one of two things to pace out its data transmission. It can either process the data and then perform a check (conformance to the traffic parameters) right before the send call, or it can delay the processing of the data (using timers) to the next start time. Both methods require a form of delay mechanism, which is not feasible nor accurate for small delays and large number of applications and flows. If each application or flow is processed by a different process or thread, the number of context switches from user to user and from the user to kernel during the send call can add significant overhead and load on the server CPU(s). For example, if each flow transmits 2KB every 0.1 second, 1000 simultaneous flows will require at least 10,000 task switches in a second. To reduce this overhead, each application can increase both the burst size and burst interval to 20KB per second. This, however, requires larger buffers at each node in the network for each flow. Larger bursts also can increase the probability of packet drops during congestion within the network.

This type of self-shaping mechanism is employed by current commercial stream servers such as Apple's QuickTime Streaming Server, Microsoft's Windows Media Services, and RealNetwork's RealServer. Under low server CPU loads, such self-time mechanisms might work well. However, under heavy loads, especially when the streaming server application must share the server CPU with other applications, the amount of time the streaming server application must wait for the server CPU could be much larger than a flow's minimum burst interval. This results in interrupted playback at the client.

3.3. Operating system support

An alternative solution to self-shaping, is to add shaping and link scheduling support at the OS level. Whenever the application gets access to the server CPU, each flow can buffer enough data at the shaper to last several burst intervals, and allow the OS to perform the traffic shaping and link scheduling. This implies that the actual send call is not made until the scheduler decides to transmit an eligible packet. Also, to ensure that flows are shaped according to their parameters, the shaper needs to be periodically executed. Although in Section 2 we defined this interval to be after each packet transmission, an actual implementation can make this period larger. This allows the shaper-scheduler to batch schedule several packets for transmission which become eligible during each period. However, a large period can distort a flow so that it no longer adheres to its traffic parameters. Also, as we will show later, the amount of processing required to shape and schedule is not trivial. This load grows with the number of flows, thus taking away CPU from applications, and lowering the number of flows supported by the server. In the worst case, the processing of the shaper and scheduler could interfere with the processing required by the applications.

4. Dedicated NIC Support

Both self-shaping and OS-supported shaping and scheduling rely on running on the same CPU as the applications. When the number of flows and load on the server CPU are low, both these schemes will work well. However, when either increases, the processing needs of the shaper-scheduler will interfere with the processing needs of the applications. Our solution is to add dedicated support on the network interface (NIC). This allows concurrent operation of the shaping and scheduling with the rest of the server. The following subsections describe the basic operation of our NIC, and two possible implementations.

4.1. Basics

4.1.1 Flow setup

An application will setup a new flow only if the admission control algorithm [3] determines that the resources required to process and transmit the new flow will not exceed server resources. The admission control algorithm also guarantees that flows that are currently in service will not be affected when the new flow is admitted. Once the flow is admitted, an internal flow id is assigned to that flow. All references to the flow are made using the flow id. The flow's data parameters are then downloaded into the NIC's shaper-scheduler.

4.1.2 Packet movement

Before data can be transmitted it needs to be packetized and moved to the NIC buffers. Packetization involves moving the data down a networking protocol stack, which appends headers to the data. Assuming a UDP transport protocol, an IP network protocol, and an ethernet link, the data will be appended with a UDP header, an IP header, and an ethernet header before being copied into the NIC buffers. In order to reduce the packet processing and movement overheads [4, 13, 14, 24, 27], many researchers have proposed various network interface architectures. Single-copy schemes [4, 13, 14, 27], which move data directly from user space to the NIC buffers, attempt to reduce the multiple memory copies necessary as the data moves down the various protocol stacks. Other schemes move some or all of the packet processing onto the NIC [15, 24, 32]. Instead of adding to the NIC, other schemes introduce new buffer-management mechanisms [16], reduce DMA overheads [7], or give user-level applications direct access to the NIC [17].

We borrow from these results by assuming the following model. Each flow is assigned a user-level memory space which does not get paged out of physical memory. The amount of memory required is dependent on the maximum amount of time an application must wait to use the CPU. As we stated in Section 3.1, the process scheduler is responsible for setting an upper bound on this time, even under heavy CPU loads. For example, if the process scheduler can guarantee an application enough CPU cycles every 2 seconds to process a 300 kbps video stream, the size of the memory will need to be 70 KB. The address and size of this memory will be fixed for the duration of the flow. The NIC is initialized with the starting address of this block of memory at the time of flow setup. For simplicity, we assume that all addresses are physical addresses and no address translation is needed for virtual addresses. Packets are then copied into NIC buffers using a DMA mechanism on the NIC. At this point, the packet has been fully processed and is ready for transmission, requiring no further packet processing at the NIC.

4.1.3 Buffer management

To amortize DMA overheads, the buffer manager on the NIC downloads several packets at a time for each flow. Each flow will have an associated linked list of packets in the NIC buffers. The buffer manager keeps track of the current number of packets in the buffers for each flow, and downloads packets from memory when needed. When the scheduler

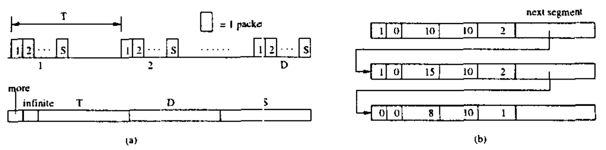


Figure 2. Data structure used to encode traffic parameters

determines the next packet to transmit, the packet's flow id is used by the buffer manager to read the packet from NIC buffers into the NIC's physical interface FIFO. A separate linked list is used for best-effort packets. Best-effort packets are transmitted by applications which are not associated with any flows and do not require any QoS guarantees.

4.1.4 Control interface

A memory-mapped control interface is used to communicate with the NIC. All commands (download traffic parameters, stop a flow) are delivered through the command FIFO. Any exceptions, status information, and shaper-scheduler requests are made through a request FIFO.

4.1.5 Traffic parameters

Figure 2(a) shows the data structure (schedule segment) used to encode the traffic parameters for each flow. Like most shaping and scheduling disciplines, we use a simple rate-based scheme. The rate is specified by its burst interval (T), burst size (S), and duration (D). The duration refers to the number of bursts required to send the entire flow. For a stored video file, this is simply the file size divided by S. For a continuous flow (live video, audio), the infinite flag is set to tell the shaper-scheduler to ignore D. Both T and S are in units of chunks, which are fixed-size segments. We assume that all packets are integer multiples of chunks. For example, if S=2, and T=20, then the flow requires 2 chucks to be transmitted for every 20 chunks transmitted.

This traffic model can also be used to characterize VBR-encoded video streams. Smoothing techniques [18] can be used to compute a transmission schedule which consists of a small number of constant-rate transmission intervals. This can be encoded as several schedule segments, as shown in Figure 2(b), which uses the more flag to chain 3 schedule segments.

4.1.6 Shaper-scheduler operation

Each active flow has an associated shaper tag which is queued in the shaper queue. The shaper tag consists of the flow id, the start time, and finish time of the flow's HOL packet. The start time is the finish time of the previous packet plus the burst interval (T). For the first packet in the flow, its start time is the current time. An internal counter keeps track of the time by counting the number of chunks scheduled for transmission so far. The finish time is the packet's start time plus T. In other words, its deadline is the earliest time the next packet in the flow can begin transmission. When the HOL packet becomes eligible (its start time is greater than or equal to the current time), the shaper will create a scheduler tag (flow id and finish time) and insert it into the scheduler queue. If there are more packets to transmit in the flow, the shaper reads the flow's schedule segment to create a new shaper tag. If this packet is the last in the flow, an end-of-flow message is written into the request fifo. After each packet transmission, the shaper will move any packets that have become eligible, while the scheduler determines the next packet to transmit. The scheduler then writes a data tag (flow id) into the transmit FIFO, which is processed by the buffer manager. If there are no eligible packets, the shaper will release a best-effort packet into the scheduler.

4.2. Hardware Implementation

Figure 3 shows the block diagram of the hardware implementation, which consists of two main blocks. The buffer manager contains the packet buffers, as well as the state machine to update the linked list of packets and interact with the DMA engine. The control block contains pointer memory, an SRAM module, and the traffic shaper and link scheduler. The pointer memory, indexed by flow id, keeps track of the list of scheduler segments for each flow. Each line of the schedule segment SRAM contains one schedule segment. Since a flow can have multiple schedule segments, a FIFO (not shown) keeps track of free lines in the schedule segment SRAM that can be used to store new schedule segments. When a flow ends, or when the schedule segment ends, its address is returned to this FIFO. The heart of the shaper and scheduler is a dedicated priority queue (PQ) mechanism [10, 23], which provides constant time queueing, sorting, insertion, and removal of tags. The shaper's priority queue sorts tags based on the start time, while the scheduler's priority queue sorts tags based on their finish time.

4.3. Software Implementation

The NIC architecture for the software implementation is shown in Figure 4. This is similar to Myrinet's network interface [8] which uses the LANai processor and three DMA engines to move data to and from the network interface. Although Myrinet uses byte-wide Myrinet physical connections, our architecture does not depend on the physical interface and can be built upon any link technology.

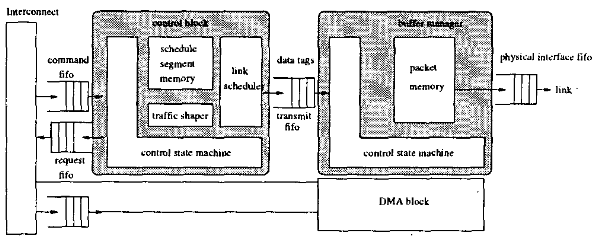


Figure 3. Block diagram of hardware implementation

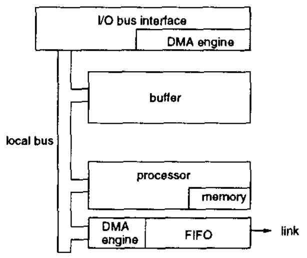


Figure 4. NIC with dedicated processor

All schedule segments, packets, and other information are stored in onboard memory which is accessible by the server processor. The DMA engine and transmission FIFO are addressable by the NIC processor. We also assume that the transmission FIFO has a DMA state machine which allows it to read out packets from memory without involving the NIC processor.

The core of the software implementation runs in an infinite loop creating and removing tags from the shaper and scheduler PQs, updating schedule segments, inserting tags into the transmission FIFO, and checking the command FIFO. A region of the NIC processor's memory is used as the command FIFO which can be accessed by the server processor.

As with the hardware implementation, the traffic shaper will insert as many entries into the link scheduler as possible. This is done to prevent missed deadlines. Consider two connections with the same eligibility time. If only a single entry is written into the scheduler PQ before running the link scheduler, the stream with the less urgent deadline can be scheduled and transmitted first, causing the other stream to miss its deadline.

Instead of transferring the actual page into the transmission FIFO, the NIC processor sets up the DMA engine on the FIFO. Similarly, when reading a page from source into NIC buffers, the NIC processor sets up the DMA engine on the I/O bus interface. Using DMA engines frees up processor cycles for other computations. Since the NIC processor is not actively involved in the actual transfer of packet data, a notification scheme is used to signal the NIC processor at the end of a page write into NI buffers and at the end of a page transmission. This is needed to manage the linked list of pages and to keep track of the NIC buffer usage.

5. Evaluation

To evaluate our proposed architecture we developed a simple event simulator using C, and modeled both hardware and software versions, also in C. This provides us with a common framework which makes comparing our results more meaningful. This also allows us to use the same traces and setup configurations to evaluate both implementations. Simulation results show the efficacy of our architecture in providing QoS-sensitive link multiplexing, especially when dealing with a large number of streams with widely-varying rates on a very high capacity link. We introduce the concept of period division and show the performance improvements obtained by using such fine-grain link multiplexing, and how our architecture can support this feature.

5.1. Simulation Environment

Our simulator allowed us to model and simulate arbitrarily large software components and arbitrarily small hardware components on a single platform. All components are modeled in C, while the simulator uses a single event-based queue with events sorted by time. Each event consists of a timestamp, a pointer to a data structure (object) which corresponds to an instantiation of a component, and a pointer to the next event in the queue. An object consists of three parts. The first includes data structures which describe the current state of the component. The second part contains a list of pointers to other objects which can be triggered by the execution of the current component. The last part is a simple pointer to the task which determines the behavior of the component. For example, in the hardware implementation, the traffic shaper and link scheduler were implemented as separate components. In the software implementation, a single component models the shaping and scheduling program running on the NIC processor.

Threads are used during the execution of software components. Instead of modeling an entire processor, our simulator executes the actual software component code. Timing information is annotated into the original code to model delays in software components, while hardware components assume existence of a global clock. During the simulation, the execution of the code is halted at regular intervals, and resumed by adding a future event into the event queue. Threads provide a convenient mechanism for halting and resuming execution of the code. Although annotating delays into the code only produces rough estimates of the performance of the software, tests showed that the simulator was accurate in estimating trends, which allowed us to correctly analyze the scalability of the software components. Since both the simulator and all models are compiled into a single executable, run times are significantly reduced compared to other methods which interpret component descriptions during runtime. By combining the flexibility of C and the speed of a compiled simulation we were able to quickly explore different design alternatives and to obtain more meaningful data from running simulations of longer run times.

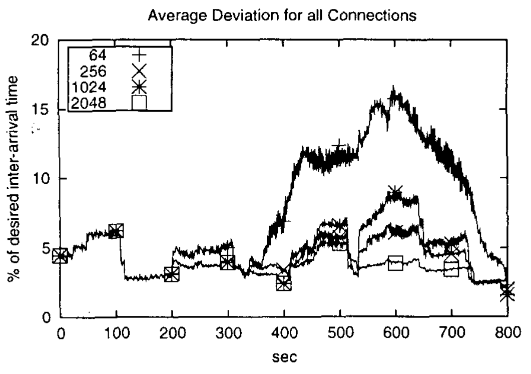
5.2. Reducing Delay and Delay Jitter

In our simulations we used link speeds of 100 Mbps, 622 Mbps, and 1 Gbps, with a fixed packet size of 128 bytes. Streams that need to burst data in larger-sized packets can easily do so by using multiples of 128 bytes. By parsing the packets on the outgoing link based on their stream id, we were able to measure the delay jitter seen by the end clients.

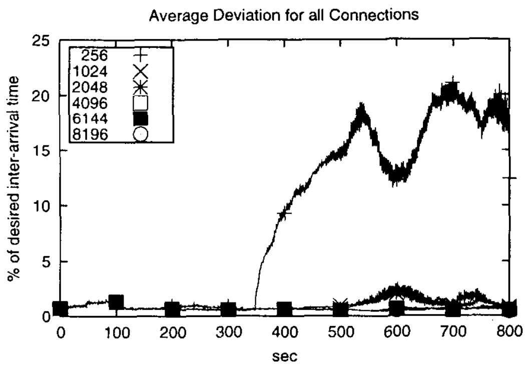
At 100 Mbps the number of simultaneous streams ranges from 20 to 140, while the total number of streams during each simulation run was close to 200. For the 1 Gbps link, these numbers were between 120 to 580, and over 1000 streams. To quantify overall performance we measured the average delay and delay jitter for each stream over a one-second interval. We then converted the delay jitter number into a percentage value (average deviation) based on the desired, or requested, delay. For example, a stream requesting data transmits every 1 msec will see 10% delay jitter if transmissions occur every (1 ± 0.1 msec). These values were then averaged across all streams for each 1 second interval. Link utilization was kept at around 20% for the first 120 secs, 40% for time=[120,320] secs, 85% for time=[320, 750] secs, and 20% for time=[750, 800] secs.

Figure 5 shows average deviation values. As expected, the average deviation increases with increased load on the network link. By increasing the size of the scheduler PQ, we can force the traffic shaper and link scheduler to look ahead even further in time to determine a better link schedule. As shown in the graphs deviation values remain constant throughout the simulation run for very large scheduler PQ sizes. We also see that after a certain size there is no further drop in deviation despite further increases in the scheduler PQ size.

This is mainly because the rates differ greatly among the various streams. Some of the video streams transmit large amounts of data (5 to 25 KB) every 33 msec, while other streams transmit much smaller amounts of data (100 to 1000 bytes) every 1 to 2 msecs. As a consequence these smaller period streams can potentially wait past their deadlines if they get queued behind several large bursts. Even small jit-



(a) 100 Mbps



(b) 1 Gbps

Figure 5. Performance measurements at 100 Mbps and 1 Gbps link speeds

ter values translate into large percentage values because of the relatively small period sizes. Increasing the scheduler PQ size can reduce deviation only up to a certain point because doing so does not solve the problem where the smaller period stream gets queued behind several large bursts.

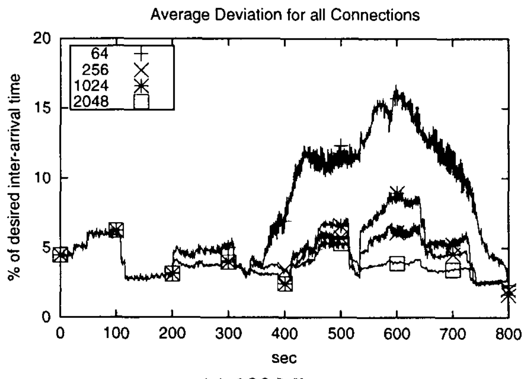
By reducing the size of the large bursts to match the burst size of the smaller period streams we can significantly reduce deviation of these smaller period streams. For example, if a video stream needs to burst 25 KB every 33 msecs, we can divide the period such that bursts are reduced to 750 bytes every 1 msec (factor of 33) or even 1.9 KB every 2.2 msec (factor of 15). We refer to this process as *period division*. Even when a smaller period stream is queued behind several video streams, the queueing time for the smaller period stream is much smaller since the burst size of the video streams has been reduced. As seen in Figure 6 deviation is reduced compared to the deviation values without the period division and using the same scheduler PQ size. This last observation is particularly important for the hardware implementation due to the hardware costs in building very large PQs. This is explained in the next section. Figure 7 shows deviation values for one of the smaller period streams. As expected, these streams gain the most by period division. From Figure 7(b) we see that the same deviation can be achieved with a 256 size PQ using period division or 8196 size PQ without period division, resulting in a factor of 32 in hardware savings. We can see the same trends in Figure 7(a).

5.3. CPU Load

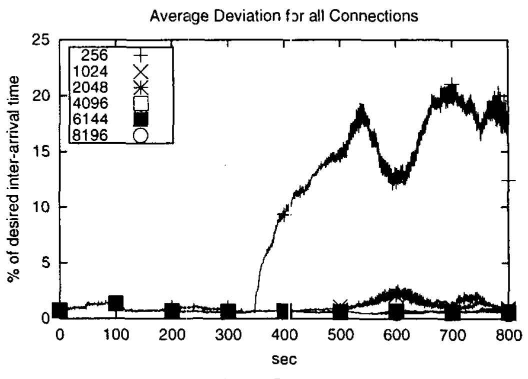
Figure 8 shows processor loads for some of the simulation runs. Since the priority queue and its operations are implemented as a binary heap, processor load does not increase significantly with increased PQ size. However, load does increase by a factor of 2 to 4 when we use period division because the number of operations to transmit the same amount of data has increased. At low link speeds the load is below 10%, but approaches 100% at high link speeds. For even higher link speeds and larger number of simultaneous streams, the computing load required will exceed the capacity of the processor. This means that the server will have to reduce the number of simultaneous clients to deliver the same level of QoS across all streams. Otherwise, the shaper-scheduler's load will exceed 100%, causing the link to go idle even when there are packets to transmit.

6. Conclusions

In this paper we proposed and evaluated a network interface architecture with dedicated support for QoS-sensitive transmission. We defined an architecture and low-level functions for supporting traffic shaping and link scheduling. Based on hardware and software implementations we measured performance seen by the user in terms of delay and jitter, and showed the effect of increasing the scheduler PQ size in reducing delay jitter. We also showed the effectiveness of fine-grain link scheduling in significantly reducing delay jitter without using very large capacity scheduler PQs, which significantly lowers hardware implementation cost. We showed that, by moving the implementation into the network interface, the server can take advantage of the concurrent execution of operations. This allows the server to support finer levels of QoS, without burdening the server processor. Not only does this free the server processor to process other tasks, but it also results in a larger number of connections receiving their desired QoS. This allows the system to make the best use of server resources in terms of processor time and link bandwidth.

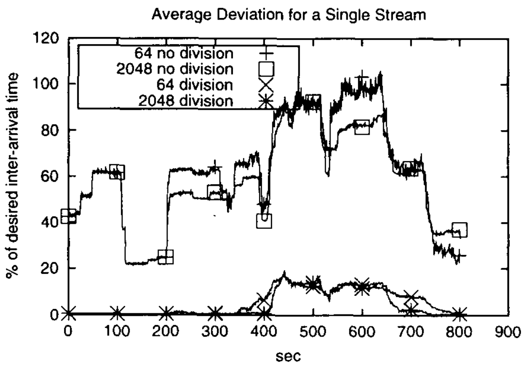


(a) 100 Mbps

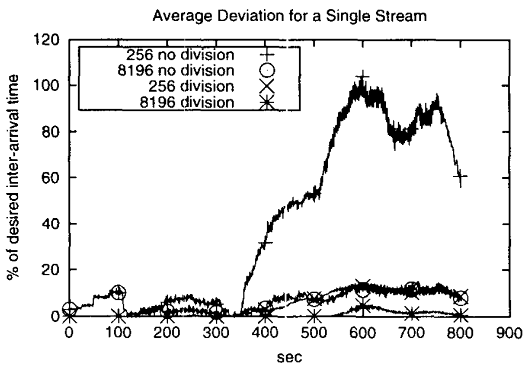


(b) 1 Gbps

Figure 6. Performance measurements after incorporating period division



(a) 100 Mbps



(b) 1 Gbps

Figure 7. Performance measurements of a single small period stream

References

[1] T. Abdelzaher and K. Shin. Qos provisioning with qcontracts in web and multimedia servers. In *IEEE Real-Time Systems Symposium*, pages 44–53, 1999.

[2] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne. Real-time communication in packet-switched networks. *Proceedings of IEEE*, 82(1):122–139, January 1994.

[3] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of qos architectures. *ACM/Springer Verlag Multimedia Systems Journal, Special Issue on QoS Architecture*, 6(3):138–151, May 1998.

[4] D. Banks and M. Prudence. High-performance network architecture for a pa-risc workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[5] J. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. In *SIGCOMM*, pages 143–156, 1996.

[6] J. C. R. Bennett and H. Zhang. Wf2q : Worst-case fair weighted fair queueing. In *IEEE INFOCOM*, pages 120–128, 1996.

[7] M. Blumrich, C. Dubnicki, E. Felten, and K. Li. Protected, user-level dma for the shrimp network interface. In *International Symposium on High-Performance Computer Architecture*, pages 154–165, 1996.

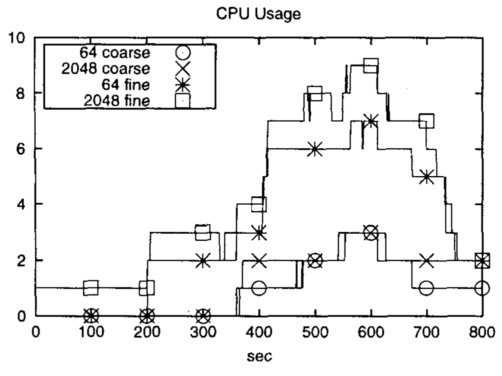
[8] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, January 1995.

[9] B. Carpenter and D. Kandlur. Diversifying internet delivery. *IEEE Spectrum*, 36(11):57–61, November 1999.

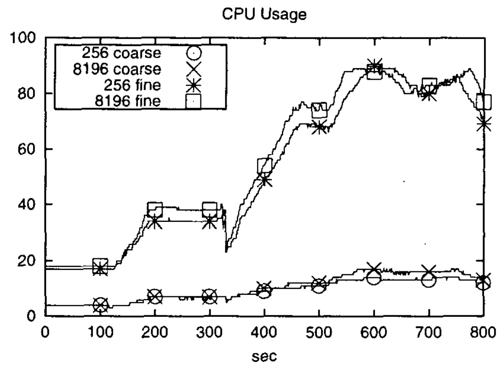
[10] H. J. Chao. A novel architecture for queue management in the atm network. *IEEE Journal on Selected Areas in Communications*, 9(7):1110–1118, September 1991.

[11] H. J. Chao, Y.-R. Jenq, X. Guo, , and C. H. Lam. Design of packet-fair queuing schedulers using a ram-based searching engine. *IEEE Journal on Selected Areas in Communications*, 17(6):1105–1126, June 1999.

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



(a) 100 Mbps



(b) 1 Gbps

Figure 8. Processor loads

[13] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, July 1993.

[14] B. S. Davie. The architecture and implemtation of a high-speed host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):228–239, February 1993.

[15] Z. D. Dittia, J. R. Cox, and G. M. Parulkar. Design of the APIC: A high performance ATM host-interface chip. In *Proceedings of IEEE INFOCOM*, pages 179–187, 1995.

[16] P. Druschel and L. Peterson. Fbufs: A high-bandwith cross-domain transfer facility. In *SIGOPS*, pages 189–202, 1993.

[17] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–76, 1998.

[18] W.-C. Feng and J. Rexford. A comparison of bandwidth smoothing techniques for the transmission of prerecorded compressed video. In *INFOCOM*, pages 58–66, 1997.

[19] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *IEEE INFOCOM*, pages 636–646, 1994.

[20] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, oct 1997.

[21] C. R. Kalmanek, H. Kanakia, and S. Keshav. Rate control servers for very high-speed networks. In *IEEE GLOBECOM*, pages 12–20, 1990.

[22] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8):1265–1279, October 1991.

[23] S.-W. Moon, J. Rexford, and K. G. Shin. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transcations on Computers*, 49(11):1215–1227, November 2000.

[24] G. W. Neufeld, M. R. Ito, M. Goldberg, M. J. McCutcheon, and S. Ritchie. Parallel host interface for an atm network. *IEEE Network*, pages 24–34, July 1993.

[25] A. K. J. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks.* PhD thesis, Massachusetts Institute of Technology, 1992.

[26] J. Rexford, F. Bonomiand, A. Greenberg, and A. Wong. Scalable architectures for integrated traffic shaping and link scheduling in high-speed atm switches. *IEEE Journal on Selected Areas in Communications*, 15(5):938–950, jun 1997.

[27] D. Saha. *Supporting Distributed Multimedia Applications on ATM Networks.* PhD thesis, The University of Maryland, 1995.

[28] D. Stiliadis and A. Varma. Frame-based fair queueing: A new traffic scheduling algorithm for packet-switched networks. In *SIGMETRICS*, pages 104–115, 1996.

[29] D. Stiliadis and A. Varma. A general methodology for designing efficient traffic scheduling and shaping algorithms. In *INFOCOM*, pages 326–335, 1997.

[30] D. Towsley. Providing quality of service in packet switched networks. In L. Donatiello and R. Nelson, editors, *Performance Evaluation of Computer and Communication Systems*, pages 560–586. Springer Verlag, 1993.

[31] L. Trajkovic and S. J. Golestani. Congestion control for multimedia services. *IEEE Network*, 6(5):20–26, September 1992.

[32] C. B. S. Traw and J. M. Smith. Hardware/software organization of a high-performance ATM host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):240–253, Feb. 1993.

[33] A. Varma and D. Stiliadis. Hardware implementation of fair queuing algorithms for asynchronous transfer mode networks. *IEEE Communications Magazine*, 35(12):54–68, Dec. 1997.

[34] H. Zhang and D. Ferrari. Rate-controlled service disciplines. *Journal of High Speed Networks*, 3(4):389–412, 1994.

[35] L. Zhang. Virtual clock: A new traffic control algorithm for packet-switched networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.