STREAMER: Hardware Support for Smoothed Transmission of Stored Video over ATM

Sung-Whan Moon, Padmanabhan Pillai, and Kang G. Shin

Real-Time Computing Laboratory, Dept. of EECS
The University of Michigan, Ann Arbor MI 48105
{swmoon, pillai, kgshin}@eecs.umich.edu

Abstract. We propose a hardware design which provides support for smoothed transmission of stored video from a server to a client for real-time playback. The design, called Streamer, resides on the network interface and handles the transmission scheduling of all streams over a single link connected to an ATM network. Streamer minimizes the interaction needed with the server CPU, while supporting very fine grain scheduling of cell transmissions for a large number of streams. Throughput is also maximized by streaming data directly from the storage medium to the network interface buffer over the I/O bus. We observe that the scheduling mechanism can be modified to operate at 2.1 Gbps link speeds and the Streamer as a whole can provide cell-granular, interleaved scheduling with full utilization of 155 Mbps and 622 Mbps links.

1 Introduction

Asynchronous Transfer Mode (ATM) networks use fixed size cells and a connection-oriented scheme to deliver information across a high-speed network. Because of their ability to deliver high-bandwidths and per-connection QoS guarantees, ATM networks are well-suited for real-time multimedia applications, such as delivering stored video for continuous playback at the client. In this application, a client requests delivery of some stored video (i.e., a video clip or an entire movie) from a remote server. The large amount of data versus relatively small buffer capacity on a typical client, coupled with the burstiness of compressed video, makes it difficult to deliver QoS guarantees for uninterrupted playback. Network resource allocation based on peak rates becomes expensive, resulting in under-utilization of resources and limited number of connections. Requiring very large client-side buffer capacity to absorb the bursts also becomes unrealistic due to cost considerations. A solution to this problem involves reducing the rate variability by using smoothing techniques [10][17]. Based on the buffer capacity of the client, these algorithms determine a transmission schedule which results in smoother transmission and decreased peak rate, without starving or overflowing the client buffer.

The work reported in this paper was supported in part by the NAtional Science Foundation under Grant MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the NSF.

Although these algorithms have been shown to increase network utilization [17], they still do not provide an end-to-end solution. These algorithms assume the video server will be able to put data into the network based on the smoothed transmission schedule. For a single stream a server CPU scheduling approach might suffice, but for a large number of streams utilizing close to full capacity of the link, such a server-based approach will not deliver the results obtainable with smoothing. Several researchers [5][6][8][9][14][15][16][18] have argued that the network I/O on current end systems, such as servers, is the main bottleneck in providing applications with the high-bandwidths available to them. In a video server application, Saha [16] recognizes a substantial penalty with a server-initiated transmission due to the cost of moving data from a storage device to the network interface via the server CPU. In this scheme, data is copied multiple times, from storage to the kernel's memory space inside the server's main memory, then to the application's memory space and back to the kernel, and finally to the network interface which sends the data out on the network link. Due to the overhead of multiple copying, the actual bandwidth available to the applications is substantially less than the link bandwidth. A common solution proposed is to reduce the number of data copies so that data can be moved from the source to the network interface with one copy.

In these single-copy schemes [5][6][18], the data that is to be transmitted is copied directly from the application's memory into the network interface's buffer. This copying is done either by the server CPU or by the network interface. Scheduling of transmissions is also done by the server CPU, which notifies the network interface of a transmission via some type of request mechanism. Unfortunately, in a video server application, data is usually on a separate storage medium (e.g., hard disk) and not in main memory. So, another method of copying data from the source to the network interface is required. Saha [16] proposes a streaming approach where the source of the data acts as an autonomous device which can send data without constant interaction with the server CPU. Once initiated, the autonomous device will continuously send data to the destination device. In his experimental platform, the autonomous device is the MMT adaptor, which captures real-time video and audio and then outputs compressed digitized video and audio streams. The MMT adaptor will then copy the data into the network interface over an I/O bus. Dittia *et al*. [8] propose a similar approach where data is streamed from the source to the network interface. Their approach consists of an interconnection of special chips (APIC), with each chip connecting to a single I/O device. This interconnection replaces the traditional I/O bus, with the first APIC connected to the network interface and main system bus. Data from an I/O device destined for the network will simply travel down the chain of APICs till it reaches the network interface.

In order to provide smoothed transmission of stored video, smoothing techniques and data streaming by themselves are not enough. Smoothing techniques can only calculate a smoothed transmission schedule. There is no explanation as to how to achieve such a smooth transmission. Data streaming techniques use a single copy approach to increase throughput available to applications and I/O devices. Although throughput is important, link scheduling is also needed in order to conform to the smoothed transmission schedule. Saha [16] proposes a scheduler based on round-robin scheduling implemented using a processor. This approach has the drawback that fine grain scheduling is limited by how fast the processor can determine the next stream to transmit. The APIC [8] solution is to simply limit the burst size and burst interval for each source of data. Because data travelling down the chain of APICs is continuously buffered at each APIC before reaching the network interface, such a mechanism cannot guarantee any smoothness in the transmission.

In this paper, we propose an integrated solution for smoothed transmission of stored video using both the smoothing schedule and data streaming. Since the I/O device can only control the rate at which it sends data to the network interface, actual link scheduling needs to be done on the network interface. Software scheduling on the server involves the server telling the network interface when to send data, how much data to send and from which stream. For a large number of streams, this approach will potentially stall the transmission due to delay associated with context switches, interrupt handling, other OS overheads, and issues dealing with CPU scheduling [11][12]. This may happen when trying to do fine grain scheduling where each stream transmits a small number of ATM cells at a time. Also, with the software scheduled method, a large number of messages need to be sent from the server CPU to the network interface. This decreases the bandwidth available to the devices for copying data to the network interface. Since the overall goal is to be able to transmit a large number of video streams as smoothly as possible, such a server CPU scheduled approach will not work. A hardware scheduler on the network interface, on the other hand, can off-load the scheduling and multiplexing of stream transmissions from the server, allowing the server to do other tasks. This also reduces traffic between the network interface and server CPU, while allowing for fine-grain scheduling at the ATM cell level. This in turn, allows for a large number of video streams to be transmitted onto the network, with each stream closely following its smoothed transmission schedule, and for better utilization of the link's bandwidth.

In this paper we propose such a hardware design, which facilitates both the streaming of stored video data from storage directly to the network interface, and the transmission scheduling of these video streams based on smoothing algorithms. Our hardware design, which we refer to as the Streamer, takes on the functionality of the transmitter unit on a network interface. The Streamer is responsible for buffering cells from various streams, scheduling the transmission of these cells, and retrieving these cells from the buffer for transmission. Parts of the smoothed transmission schedule are cached inside a buffer on the Streamer, and are used to determine which stream gets access to the link next. We argue that caching of the transmission schedule is very reasonable due to the small size of the schedule itself. The Streamer also allows unused link bandwidth to be used for best-effort traffic.

This paper is organized as follows. Before describing the Streamer, we first describe the smoothing algorithm assumed in our design, and the basic functionality of network interfaces. In Section 3 the Streamer architecture and operations are explained. Evaluation of our implementation and future work are presented in Section 4. Conclusions are presented in Section 5.

Schedule Segment #	Start Interval Frame #	End Interval Frame #	Transmit Size
1	a1	a2 - 1	B1
2	a2	a3 - 1	B2
...	...	...	...
M	aM	N	BM

(a) Smoothed transmission schedule using frame intervals output by smoothing alorithm

Schedule Segment #	Chunk Interval	# of Chunks	Chunk Size	Last
1	I1	N1	C1	0
2	I2	N2	C2	0
...	...	...	...	...
L	IL	NL	CL	1

(b) Possible format for smoothed transmissior schedule using chunk intervals

Fig. 1. Smoothed transmission schedules

2 Background

Before describing our hardware design (Streamer), we will first look at smoothing algorithms and network interface issues, which are the main components of our design. We will first describe the smoothing algorithms presented in the literature, and then make extensions to the schedule for the purposes of our design. Basic architecture and functionality of a network interface are introduced, with emphasis on ATM networks. We then describe how the Streamer fits into the network interface, and explain the operations that are supported and not supported by our design.

2.1 Smoothing Algorithms

Although optimization criteria differs among various researchers, the main goal of smoothing algorithms [10][17] is to reduce the rate variability of stored variable bit rate video. Stored video consists of frames, which are just images, captured 10-30 times a second. Compression techniques, such as MPEG, take advantage of the inherent redundancy to compress the video data, and reduce the total storage required. Such compression, though, results in burstiness due to the varying frame sizes. For real-time playback across a network, such burstiness causes problems in network resource allocation and QoS guarantees when transmitting frames every 10-30 msecs. Smoothing algorithms take advantage of the fact that *a priori* knowledge of the video stream is available for stored video, and determine transmission schedules based on the buffer capacity of the client. The resulting smoothed schedule is characterized by small rate variations, and a fairly constant transmission rate.

The smoothed transmission schedule determines the amount of data to be sent at each of the N frame intervals. For a 30 frames per second video stream, the interval is 33 msecs. Instead of transmitting a different amount of data at each frame interval, the smoothing algorithm generates several constant rate schedule segments. During each schedule segment, which lasts for several frame intervals, the same amount of data will be transmitted at each frame interval. This amount will change slightly between different schedule segments. Fig. 1(a) shows the format for the final schedule. Based on the results reported in [10, 17], the number of these constant rate schedule segments (in the 100s to the 1000s) is considerably less than the total number of frames (in the 100,000s) in a video clip or movie, and decreases with an increase in the client's receive buffer (typically around 1 MB). The typical amount of data transmitted at each frame interval is in the range of 1-5 KB based on Salehi *et al*. [17], and 5-12 KB based on Feng [10].

Considering that each ATM cell's payload is 48 bytes, a 5 KB transmission requires around 105 ATM cells, while 12 KB requires a little over 250 ATM cells. Instead of transmitting these cells in a burst at each frame interval, we assume a schedule which is modified in three ways. First, the transmission size is assumed to be in units of ATM cells. A method for calculating such a schedule is described in [17]. Second, time is specified in units equal to the time required to transmit one ATM cell. In our design, we targeted a 155 Mbps ATM link. Third, frame intervals are further divided into chunk intervals, and the cells that need to be transmitted for the frame interval are divided up evenly and sent at each chunk interval, further smoothing out transmission. On a 155 Mbps link, each cell takes 2.7 µsecs to transmit, hence over 12000 cells can be transmitted within the typical 33 msec frame interval. So instead of bursting, say 200 cells, for a half msec every 33 msecs, chunk intervals can be used to burst 20 cells every 3.3 msecs. Such spread-out transmissions can be seen as more favorable from the network's point of view. We assume that this chunk interval schedule is derived from the frame interval schedule produced by the smoothing algorithm above, and will not result in client buffer overflow or underflow. Here we also make the obvious assumption that if the smoothed transmission schedule is observed, then the network will be able to provide certain guarantees for uninterrupted playback at the client side.

Fig. 1(b) shows the chunk interval schedule that our hardware design assumes. Chunk intervals (I1, I2, ..., IL) are specified in cell transmit time units, while the schedule segment duration is specified by the number of chunk intervals (N1, N2, ..., NL). Chunk size (C1, C2, ..., CL) is given in number of cells. For example, during the first schedule segment, there will be N1 transmission bursts of C1 cells each, with these bursts occurring every I1 cell transmit time units. The "Last" field flags the last schedule segment, and indicates the end of the stream. Each schedule segment can be encoded using a small number of bytes (6-8), so the entire schedule for a movie stream can be encoded with just a few KBytes (1-8). Since each schedule segment lasts for many frame intervals, the time between schedule segment changes will be in the range of 5-50 secs, which is an eternity for hardware.

2.2 Network Interface Architecture and Functions

The function of the network interface is to transmit data from the host to the network, and similarly transfer data received from the network to the host. As shown in Fig. 2, the network interface will typically interact with the rest of the host over the host's I/O bus. When an application on the CPU needs to send data, the necessary software will transfer the data to the network interface buffer. The transmit control on the interface will then handle the necessary overhead associated with transmitting each cell. For ATM, these tasks include segmenting the application data into cells, adding ATM cell headers, and performing CRC generation. The physical layer then handles all the physical medium specific functions, such as cell delineation, framing, and bit timing. When

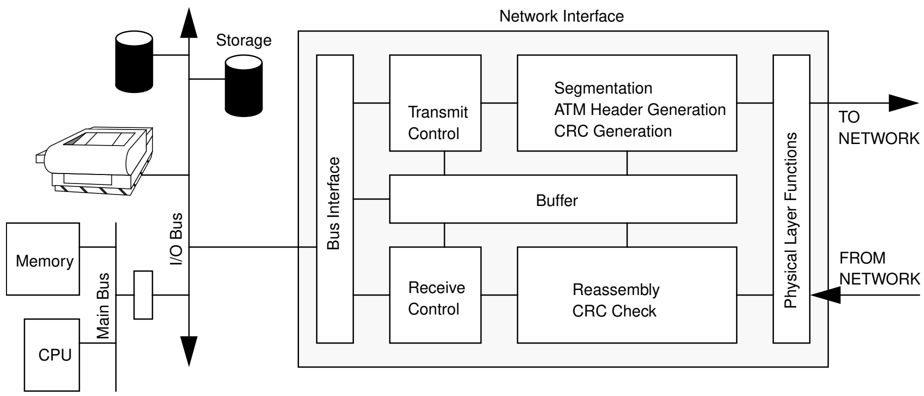

Fig. 2. Generic host system with I/O devices and the ATM network interface.

cells arrive at the interface, the receive control reassembles the cells into application data units before delivering them to the application. Commercial ATM network interface products [2][3][4] follow this general model.

As mentioned earlier, a single copy approach can improve throughput as seen by the application. Various solutions fall under one of two categories. The first consists of allowing the application to write directly into the network interface's buffer [5]. Actual transmission is initiated by sending a special control signal. The other approach is to first send a message to the network interface, which then copies the data directly from the application's memory into its own buffer [6][18]. In either case, the CPU is actively involved in each transmission. For a video server application, where a large amount of data resides on a hard disk, streaming of the data is more appropriate. In this scheme, the CPU simply initializes the hard disk to send data to the network interface. The hard disk is assumed to have enough intelligence to know when to pause and resume data flow based on feedback from the network interface. This streaming scheme is assumed for our design.

Our hardware design, the Streamer, proposed in this paper fits into the transmit control section in Fig. 2. One of the functions of the transmit controller is to determine which of the connections ready for transmission gets access to the link and for how long. Instead of simple FIFO link scheduling, Streamer is a hardware scheduling mechanism which allows for smoothed transmission scheduling of video streams. Streamer does this by maintaining scheduling information for each of the active video streams, storing cells from the hard disks into the buffer, determining which active stream needs to transmit and how much, and removing cells from the buffer for transmission. From product information we were able to obtain, several commercial products also offer some type of link scheduling on the network interface. [2] mentions independent transmit scheduling for each virtual circuit and support for MPEG-2 transport delivery. [7] mentions an ATM adapter with support for pacing of MPEG streams for smooth transmission of data. [4] describes a rate-based traffic shaping scheduler that provides interleaved stream scheduling. It uses a static round-based schedule to divide up link bandwidth among active connections. During each round, data for the corresponding connection is transmitted only if the traffic shaping permits it to do so. In contrast, Streamer dynamically schedules streams, requiring no adjustments for rate changes caused by the smoothed schedule, and also uses a dedicated priority queue mechanism to arbitrate among the active streams. Details of the Streamer are given in Section 3.

3 Streamer

We have discussed the potentials of ATM networks for the transmission of real-time video, the problems and inefficiencies of bursty video traffic in the context of guaranteed service reservation, and the existence of smoothing algorithms that alleviate these problems. We have also established that scheduling the transmission of smoothed video streams in real-time requires hardware support if link utilization is to be kept high. Addressing all of these issues, we have designed a hardware solution to real-time link scheduling of smoothed video streams that avoids CPU and memory bottlenecks by streaming data directly from storage devices to the network interface. The Streamer network interface allows for effective implementation of smoothed video multimedia servers as it can handle many simultaneous streams. We therefore extend the work done with smoothed video streams by providing the hardware link scheduling needed for efficient smoothed video streaming in a multimedia server.

3.1 Operation

The Streamer operates as follows. When the server receives a request for a new video stream, it checks for sufficient resources on the server to handle this new stream. This admission control mechanism takes into account such factors as storage device bandwidth and latency, and available buffer space on the network interface in addition to the conventional admission control on the network. Only then is the appropriate signalling performed to establish the new connection. Based on the client buffer, a smoothed transmission schedule is calculated on-line or is retrieved from storage. The server CPU then downloads the first few schedule segments and the cell header into the Streamer, and allocates sufficient number of buffers in the network interface to avoid buffer starvation.

The server CPU then signals the appropriate storage device to begin streaming data to the network interface. We assume that the storage devices are controlled by bus-mastering capable controllers. Once initialized by the host CPU, these controllers will stream data to the Streamer network interface with little or no further dependence on the host CPU. The devices are assumed to have sufficiently higher throughput than required by the video streams, and enough caching on the controllers to avoid buffer starvation at the network interface. We also assume that a file system optimized for video data stores the streams linearly in the storage devices to facilitate the autonomous retrieval by the bus-mastering controllers and to avoid seek latencies. Furthermore, we assume that the data has been preprocessed for transmission and is fragmented into ATM cells, taking into account overhead for error correction and ATM

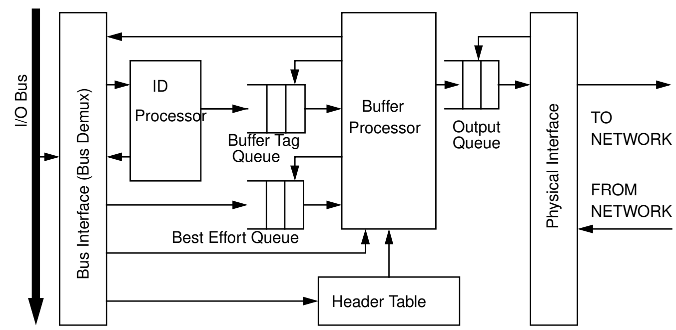

Fig. 3. Streamer.

Adaption Layer information, and is stored as a sequence of 48 byte cell payloads. This last assumption allowed us to focus on the smoothed scheduling support without worrying about the extra ATM cell processing.

Once the initialization has been completed, a control signal is sent to the Streamer to begin transmitting the particular stream. Scheduled transmission continues without host CPU intervention except to supply additional smoothed transmission schedule segments and to handle any errors that may occur.

3.2 Architecture

The basic architecture of the Streamer network interface is shown in Fig. 3. The Streamer board is not a complete network interface as it does not provide segmentation, CRC generation, or any receive functionality. Though a minimal design, it can provide fine-grained (down to cell level) interleaved transmission of up to 255 simultaneous streams based on independent traffic smoothing schedules.

3.2.1 Bus Interface (Bus Demux)

The bus demultiplexor provides the bus interface for the Streamer. In addition to decoding addresses and generating internal control signals, the bus demux also provides simple feedback to the data storage devices in order to prevent buffer overflows. As bus-mastering drive controllers stream data to the Streamer board, the bus demux determines if buffer space is available (based on a per-stream allocation and a table maintained in conjunction with the buffer processor). If buffer space for the particular stream is not currently available, it uses a device-not-ready mechanism to force the drive controller to relinquish the bus and try again later. The bus demux can also generate interrupts when the ID processor or the buffer processor needs servicing from the server CPU.

3.2.2 ID Processor

The ID processor (Fig. 4) is the heart of the Streamer board. This module performs

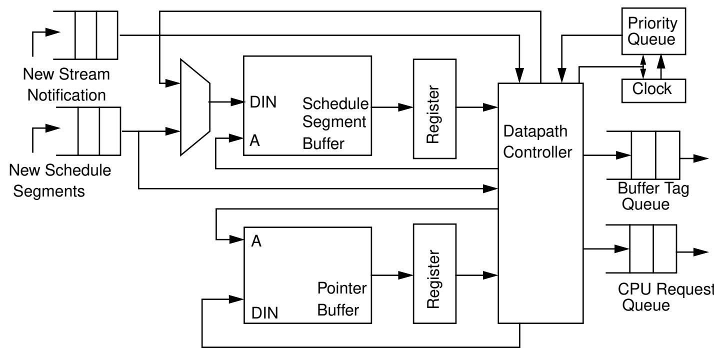

Fig. 4. ID Processor

scheduling among the active streams and generates a sequence of buffer tags, which are control structures that indicate a stream ID and a cell count, and are used to tell the buffer processor what to transmit. For each stream, the ID processor stores up to 4 segments of a smoothed transmission schedule. These segments indicate traffic shaping parameters such as burst length and burst interval to be applied to a specified number of cells. As each segment expires, the ID processor will place a request for more through the CPU request queue.

Scheduling is performed by keeping the departure time for the next chunk of data for each stream in a hardware priority queue [13]. The hardware priority queue allows single cycle insertion, and allows arbitrary interleaving of streams to produce a precise schedule for transmission, unlike the imprecise schemes used elsewhere [4], that are essentially round-robin schemes modified to take traffic shaping into account. Furthermore, the priority queue approach is fully dynamic, and works well with streams that have changing data rates, unlike the round-robin variants, which require the recomputation of a static schedule to reflect each data rate change. The priority queue produces the proper ordering of transmission from the active streams, while a cell counter, acting as a clock, ensures proper time intervals are maintained between subsequent transmissions. When the clock indicates that the stream at the top of the priority queue is ready to transmit, the entry is removed from the queue. Based on the stream's scheduling segment data, a buffer tag is generated indicating the stream ID and the number of cells to transmit. The number of remaining transmissions left for the particular schedule segment is updated, while the departure time for the next burst of data for the stream is computed and inserted into the priority queue. This entire operation can be accomplished in constant time (less than 10 cycles). If it is not time to send video stream data, delay slots can be introduced in the output by producing a special buffer tag that indicates either a best-effort traffic cell or an idle cell should be released instead of data from the video streams.

3.2.3 Buffer Processor

The buffer processor (Fig. 5) is essentially a combined buffer memory and manager

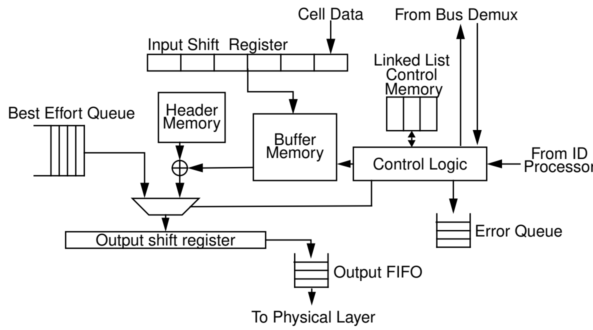

Fig. 5. Buffer Processor

with the added responsibility of sending cells to the physical layer interface. The buffer memory is organized as a bank of 48-byte cell buffers. Buffer allocation and management is performed through simple linked list mechanisms on a per-stream basis. A separate linked-list control memory maintains the buffer pointers for each stream as well as the list of free buffers. For simplicity of control and timing, the different data structures (cell buffers, header table, linked-list pointers) are stored in separate, parallelly accessible memories. Individual streams are limited to a specified allocation of buffers, which is determined during admission control. This is enforced in conjunction with the bus demux through a table that indicates the number of buffers that remain available to each stream. An input shift register decouples the buffer processor operation from the six cycle latency in transferring a cell payload over a 64-bit I/O bus. On the output end, as buffer tags are received from the ID processor, the indicated number of cells for the indicated stream ID are prepended with the ATM 5 byte header from a table of headers, and are transferred octet-by-octet to the physical layer through a shift register and a FIFO queue. This transfer mechanism is based on the ATM Forum Level 1 UTOPIA specifications [1]. If an insufficient number of cells are buffered for the stream, an error is reported in an error queue that causes an interrupt to be signaled. If the special idle tag is received from the ID processor, the next cell in a best-effort traffic FIFO queue is sent, or if that queue is empty, an idle cell is sent to the physical layer to ensure that proper cell timing is maintained.

4 Evaluation and Future Work

Streamer has been implemented in Verilog HDL using both behavioral and structural descriptions. Several Verilog simulations were performed to verify the correctness of our design. Due to the interactive nature of the network interface with the rest of the system, special behavioral Verilog code had to be written to emulate the functioning of the storage devices, I/O bus, and server CPU. Due to the extreme complexity involved in trying to do a precise simulation of these components in Verilog, we used simplified models of their behavior, focusing instead on the interaction with our network interface. These extra behavioral models allowed us to easily create test situations under which our design could be observed.

As expected, we observed that the hardware scheduling ran at significantly greater speeds than the transmission link, and as a result was idle most of the time waiting for the physical layer to catch up. This is expected since each ATM cell requires 2.7 µsecs to transmit, while scheduling of cells, based on a 33 MHz system clock, can be done within 10 clock cycles, or around 300 nsecs. Without modification our hardware priority queue scheduler is therefore capable of scheduling 3.3 million cells per second, corresponding to a link speed of 1.4 Gbps. As the hardware priority queue has already been shown to work at 64 MHz [13] using static CMOS technology, and since 15-20ns access time SRAMs are common, the scheduler can be trivially modified to operate off of a 50 MHz clock, and work with up to 2.1 Gbps links.

In contrast, a software-based scheduler using a dedicated processor on the adapter board will not scale as easily with increasing link speeds. If the processor were to handle only the scheduling of cells, an efficient implementation of the 255 entry priority queue would require between 200 and 300 instructions to schedule a single cell (depending upon the instruction set of the processor: 275 on HP PA-RISC, 290 on SPARC, 232 on RS/6000). This requires an effective throughput of 73 to 110 MIPS to provide cell-granular scheduling on a 155 Mbps link. Higher link speeds require proportionally higher instruction throughput. To put this in perspective, a 100 MHz RISC processor, capable of sustaining one instruction per cycle on the optimized priority queue code with 250 instructions executed per cell scheduled, would be capable of scheduling 1 cell every 2.5 µsecs. This is just within the 2.7 µsec transmission time of cells on a 155 Mbps link. So, even without taking into account other functions the onboard processor may need to perform, this processor is already incapable of providing full link utilization for rates over 170 Mbps. In other words, for very high link speeds a software-based scheduling at cell level granularity will require a very high performance, expensive microprocessor implementation.

As stated above, our scheduling module can be readily adapted to 2 Gbps operation. In our current implementation, the buffer management and transmission unit, based on the 33 MHz clock and the 8-bit wide UTOPIA[1] standard, is limited to 264 Mbps. By removing the 33 MHz and 8-bit wide limitations, faster throughput is attainable. Since the update of the buffer management structures can be performed in 6 cycles, and since two updates occur for each cell transmitted, the buffer management can handle 5.5 million cells per second (over 2.3 Gbps) with a 66 MHz clock. By using a 66 MHz, 32-bit wide interface to the physical layer, the buffer processor can support a 2 Gbps link.

In Streamer, the scheduling is done in parallel with the transmission of previously-scheduled cells, so the link can be kept fully utilized whenever there are cells to transmit. This is assuming that both schedule segment data and video data are present in the network interface buffers. As was mentioned in Section 2.1, each schedule segment will typically last several seconds and this combined with the fact that we store a few schedule segments ahead, allows large server CPU delays to be tolerated when additional scheduling segments are needed.

However, the transfer of video data across the I/O bus and into the cell buffers will restrict throughput and link utilization. Due to the simple data transfer model, in which the storage devices continuously try to push data to the network interface, and the fact

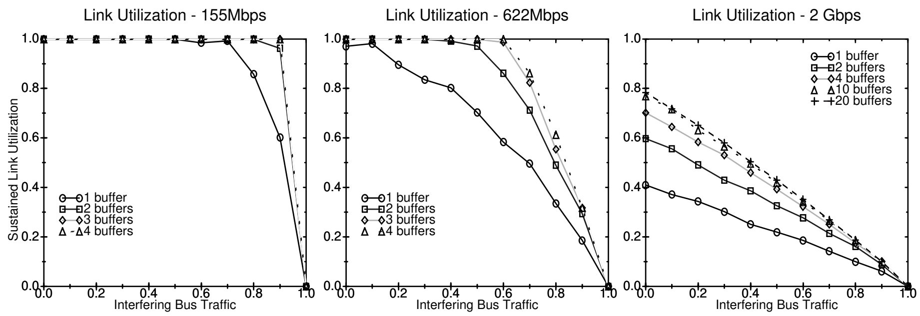

Fig. 6. Sustained Link Utilization

that multiple devices contend for the use of the shared bus, buffer starvation may occur beyond some throughput level. We performed simulations that quantify this effect. We model a Streamer network interface modified to operate at up to 2 Gbps and support more than 256 streams, and a PCI standard server I/O bus, based on the proposed 64-bit, 66 MHz implementation, with a peak throughput of 4.2 Gbps. We assume that bus arbitration results in a round-robin traversal of devices contending for bus access. To isolate the effects of the bus, we assume that each storage device has an internal cache and serves just one stream, so retrieval latencies are hidden and more than sufficient bandwidth is available. We also model interfering traffic on the I/O bus by granting a fraction of the bus accesses to a dummy device. We ran this simulation repeatedly with random sets of streams (mean stream data rate of 2 Mbps, 50% of streams between 1.8 and 2.2 Mbps) corresponding to various link utilizations, and determine the point at which buffer starvation begins to occur. We repeat the simulations, varying the fraction of interfering bus traffic, the number of cell buffers on the network interface for each stream, and the link speeds. The results are summarized in Fig. 6. The storage devices are granted bus access in round-robin fashion, and at each access, the device will send data corresponding to 1 ATM cell, but will abort its access if the corresponding stream's buffers are all full or if the Streamer buffer management is busy (within 4 clock cycles of the start of a cell transmission to the physical layer). As the Streamer does not accept data on the bus when it starts the transmission of a cell, and since the transmission rate is greater for the high speed links, the probability that a storage device will be forced to abort (even though buffer space is still available for the stream) on any particular bus access increases with link speed. With a large number of streams, there is a good chance that some devices will abort repeatedly on consecutive accesses, resulting in buffer starvation for the associated stream. Increasing the per stream buffering helps avoid starvation, as shown by the large jump in sustainable utilization when more than one buffer per stream are used on the 622 Mbps and 2 Gbps links. However, beyond the first few buffers per stream, there is little further increase in sustained utilization. This is apparent in the nearly identical results for 10 buffers and 20 buffers per stream on the 2 Gbps link.

Based on these observations, Streamer in both its original 155 Mbps implementation and in a 622 Mbps version does provide support for a large number of streams, fine grain interleaved link scheduling, full link utilization, low CPU overhead, and potentially improved network utilization resulting from the smoothed transmissions. To maintain all of these benefits at the full potential of 2 Gbps of the hardware priority queue scheduler, further refinements are required. We plan to study the Streamer concept further, and in particular, we would like to vary the operation of the bus interface, the major limitation in the current design. We are considering replacing the current “data push” model of operation, in which the storage device continuously try to send data to the network interface, with a “data pull” approach, in which data is requested by the Streamer only when necessary. This may improve link throughput and bus utilization, at the expense of increased hardware complexity. Another potential modification is to transfer data in blocks larger than the current 48 byte cell sized units and possibly reduce the bus overheads incurred. More significant changes may include entirely replacing the I/O bus with some form of internal network, which may allow for scalable use of multiple Streamer devices in the same system. We plan to develop a detailed, flexible simulator in order to quantitatively evaluate performance, determine optimal values for parameters such as buffer requirements, and characterize quality-of-service provided by the various architectural changes in the Streamer design. Such simulations will also allow us to easily vary the components of the video server and therefore determine system requirements under different scenarios.

5 Conclusion

In order to take full advantage of the features provided by ATM networks and bring them to the end systems, improvements need to be made within the end system. For the video server application, we proposed and described a hardware solution which would do just that. The main idea behind our design is to combine smoothed transmission scheduling with data streaming support on the network interface. This results in high sustained throughput from the server to the network, and also results in better utilization of network resources due to smoothing of bursty data streams. By moving the scheduling away from the server CPU and onto the network interface in the form of a hardware priority queue based scheduler, the server is free to do other work. Communication between the server CPU and network interface is also significantly reduced, and fine grain link scheduling becomes possible. Such fine grain scheduling at the cell level is possible in hardware because, unlike a software scheduler, the hardware scheduler can operate much faster than the cell transmission rate on the network link. We implemented our design using Verilog HDL, and based on Verilog simulations, showed functional correctness and potential performance gains. We also evaluated any limitations on scaling the current design to faster link speeds and indicated some potential future improvements.

References

1. ATM Forum Level 1 UTOPIA Specification 2.01," 1994.
2. ATM-155+ Mbps SAR Controller Module," http://www.chips.ibm.com/products/commun/a155.html, 1996.
3. SunATM Adapter," http://www.sun.com/products-n-solutions/hw/networking/jtf_sunatm.html, 1996.
4. TranSwitch SARA II TXC-05551 Product Preview: Document Number TXC-05551-MB," http://www.transwitch.com/iocd.html, 1996.
5. C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, pages 36–43, July 1993.
6. 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.
7. M. Day, S. Luning, and D. Spence. White paper: IBM mediaStreamer Solutions, A Technical Overview. http://www.rs6000.ibm.com/solutions/videoservers, February 1997.
8. 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.
9. P. Druschel, M. B. Abbott, M. Pragels, and L. L. Peterson. Network subsystem design. *IEEE Network*, pages 8–17, July 1993.
10. W.-C. Feng, *Video-On-Demand Services: Efficient Transportation and Decompression of Variable Bit Rate Video*, PhD thesis, The University of Michigan, 1996.
11. A. Mehra, A. Indiresan, and K. G. Shin. Resource management for real-time communication: Making theory meet practice. In *Proceedings of IEEE Real-Time Technology and Applications*, pages 130–138, 1996.
12. A. Mehra, A. Indiresan, and K. G. Shin. Structuring communication software for Quality-of-Service guarantees. In *Proceedings of IEEE Real-Time Systems Symposium*, 1996.
13. S.-W. Moon, J. Rexford, and K. G. Shin. Scalable hardware priority queue architectures for high-speed packet switches. To appear in *Proceedings of IEEE Real-time Technology and Applications Symposium*, pages 203-212, 1997.
14. 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.
15. K. K. Ramakrishnan. Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications*, 11(2):203–219, February 1993.
16. D. Saha, *Supporting Distributed Multimedia Applications on ATM Networks*, PhD thesis, The University of Maryland, 1995.
17. J. D. Salehi, Z.-L. Zhang, J. F. Kurose, and D. Towsley. Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing. *Performance Evaluation Review*, 24(1):222–231, May 1996.
18. 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, February 1993.