Scalable Multimedia-On-Demand via World-Wide-Web (WWW) with QOS Guarantees*

Milind M. Buddhikot
milind@dworkin.wustl.edu
+1 314 935 4203

Gurudatta M. Parulkar
guru@arl.wustl.edu
+1 314 935 7534

R. Gopalakrishnan
gopal@dworkin.wustl.edu
+1 314 935 8563

Computer and Communications Research Center
Department of Computer Science
Washington University
St. Louis, MO 63130

1 Introduction

Large scale storage servers that provide location transparent, interactive, concurrent access to hundreds or thousands of independent clients will be important components of the future information superhighway infrastructure. In recent years, the World Wide Web (WWW) infrastructure has gained immense popularity and experienced explosive growth as a means to disseminate multimedia information in the Internet. Therefore it is imperative that future large scale storage servers provide a WWW based interface as one of the many access interfaces.

This paper focuses on our prototyping effort presently underway at the Washington University in St. Louis under the NSF's National Challenge Award (NCA) grant aimed at deploying a scalable Multimedia-On-Demand (MOD) server. This sever will offer a scalable MOD service that supports end-to-end Quality-of-Service (QOS)[1] guarantees, high concurrency, and a web based access that provides complete playout control functions such as *random search*, *ff*, *rw*, *pause*, *slow-play* and *content based searches*. The rest of this paper describes our approach to achieving these objectives.

1.1 Prototype Architecture

Figure 1 shows the architecture of the distributed storage server currently being prototyped. It consists of eight 133 MHz Pentium PCs equipped with network interfaces from the Efficient Networks, Inc. and interconnected using a Bay Networks 155 Mbps per port ATM switch. Initially, it will support $\approx$ 100 hrs MPEG-2 or $\approx$ $300GB$ storage constructed out of 40 GB storage at each PC. Each PC runs a local NETBSD (Version 1.1) operating system enhanced to handle periodic multimedia streams. The aggregate storage and network throughput of this prototype is expected to be 1.1 Gbps. Upon the availability of the 2.4 Gbps APIC interface chip [3], the existing network interface will be replaced with an APIC card. Sixteen such PCs will then be interconnected into a "storage cluster" by a desk area network constructed by daisy chaining the APIC chips. Eventually, several such storage clusters will be connected to the next generation 2.4 Gbps per port ATM switch, currently being prototyped at Washington University, to realize a multi-gigabit capacity storage server [1].

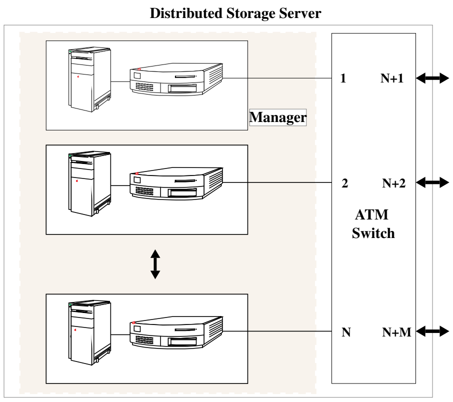

Figure 1: Prototype server architecture

Two important performance metrics for a storage server are **parallelism** and **concurrency** [2]. The parallelism (P_f) metric is defined as the num-

*This work was supported in part by the ARPA, National Science Foundation, and an industrial consortium of Ascom Timeplex, Bellcore, BNR, Goldstar, NEC, NTT, SynOptics, and Tektronix.

[1] QOS guarantees will be in the form of guaranteed bandwidth and bounded delay/delay-jitter.

ber of storage devices simultaneously participating in supplying data for a document being accessed by a client. Large parallelism increases network and storage throughput and thus, increases scalability. On the other hand, the concurrency C_f metric is defined as the number of clients that can simultaneously access the same document f. Ideally, the number of clients served from a single copy should be maximized. In other words, higher concurrency minimizes the need for replication. Clearly, P_f and C_f are related; increasing P_f increases C_f. Thus, the key to achieving scalability is to increase parallelism and concurrency.

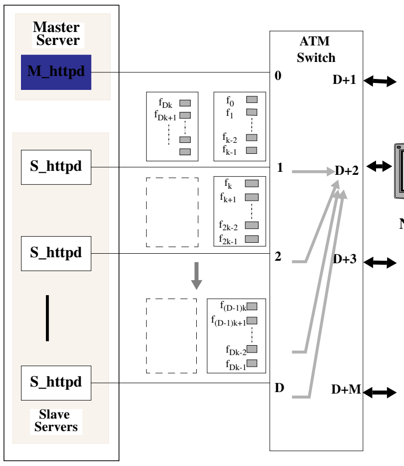

Figure 2: Scalable `httpd` server

Figure 2 shows our approach to achieve scalability in data access. As shown in this Figure, each Pentium PC in our architecture runs a copy of a modified `httpd` web server. One of these servers is designated as a *Master* server and rest of the D servers are called the *Slave* servers. To the external world, however, the collection of `httpd` servers appears as a single server. The data for video, graphics, and animation documents are physically striped over all the slave servers, whereas the data for streams such as audio, text, and data that require less bandwidth are confined to a single slave server. The master server receives the HTTP requests from the external clients. if a request is an ordinary HTTP request, the master forwards it to the appropriate slave server. However, in case of video or other bandwidth intensive media, the master arranges for all slave servers to send data in a synchronized fashion. Specifically, it sets up a many-to-one connection over which each slave server sends data following a distributed scheduling protocol [2]. The master server also maintains a one-to-many control connection to all the slaves, over which it sends control commands received from the client, and any other commands required for correct operation of the httpd server.

Thus, this architecture is truly scalable in terms of storage capacity, network and storage throughput, and number of concurrent clients. Also, due to the use of commodity components it will be quite inexpensive compared to the MIMD or SIMD multiprocessor based large scale servers currently marketed by commercial companies such as Silicon Graphics, Hewlett Packard, Whittaker Communications, Inc. and Sarnoff Real-Time Systems.

2 Supporting Web Based Access

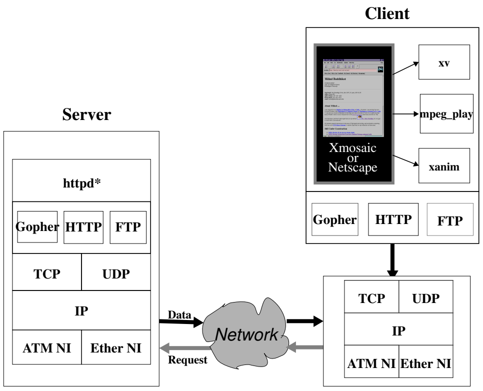

Figure 3: Basics of WWW

The WWW architecture as shown in Figure 3, consists of three entities: the server, the client and the internet that connects them. The server is essentially a daemon that provides access to text, images and data to a client by implementing a set of application protocols such as *Gopher, HTTP,* or *FTP*, layered over the TCP/IP protocol stack. Currently, several commercial and public-domain server implementations of web servers are available. In our work, we use the public-domain NCSA `httpd` server.

A WWW client essentially consists of a browser program such as *NetScape, Xmosaic etc.* which provides a point-and-click interface to multimedia information via the application protocols mentioned above. It also uses helper applications such as *mpeg_play, xanim, xv,* and *ghostview*[2].

The Hypertext Transfer Protocol (HTTP) represents the most common protocol used for information exchange between the WWW clients and servers. In

[2] **mpeg_play** is the software decoder from University of California Berkeley. **Xanim** is the software decoder for QuickTime movies, whereas **xv** is the software decoder for image formats such as *JPEG,gif, tiff, postscript etc.*

essence, HTTP consists of a set of "methods" such as GET, PUT, LINK, and FORM, which are sent to the server by the client. The server executes these methods and returns the response which can be data or a status notification. The GET method is the most common method used to retrieve a document - ordinary text, hypertext, postscript data, image, or audio/video file, from the server. When the client receives the file sent by the server, it derives the file type based on the file name extension. If the received file is not an HTML file, it consults a local database (a system wide or a user defined `.mailcap` file) to invoke the appropriate helper application. Interactive operations required, are implemented at the client and do not involve the server and network.

Thus, the existing WWW (HTTP) framework uses a *"file transfer paradigm"* which is unsuitable for bandwidth and storage intensive streams such as video, graphics, and high quality animations. Instead, for such data types, a *"data streaming paradigm"* that allows data to be sent at a regular rate from the server to the client, and allows control messages to be exchanged between them, should be used.

Some modifications to WWW framework are required to achieve the above goal. We have extended the HTTP protocol by adding a new method called GETSTREAM to the available set of methods. This method is used to request the *httpd* server to stream the data corresponding to a video/audio file to a client. The syntax of the GETSTREAM method allows the client to specify ordinary playout, interactive control operations, and content based searches. A few examples of GETSTREAM requests from the client to server are shown below:

```
1] GETSTREAM /~MPEG_DATA/alien.mpg?
   3000+TCP+OPEN HTTP/1.0 <CRLF>
```

In this request, the client requests the server that the MPEG movie `alien.mpg` in `/ MPEG_DATA/public_html` directory be streamed over a TCP connection to the client side port number 3000.

```
2] GETSTREAM /~MPEG_DATA/alien.mpg?
   3000+CMTP+OPEN_LOOPPLAY HTTP/1.0 <CRLF>
```

On the other hand, in this request, client requests the server that the same movie be streamed using a video transport protocol as in [8], and that it should be opened in the *LoopPlay* mode.

```
3] GETSTREAM /~MPEG_DATA/alien.mpg?
     3000+TCP+RANDOMACCESS+4500 HTTP/1.0 <CRLF>
4] GETSTREAM /~MPEG_DATA/alien.mpg?
   3000+TCP+PAUSE HTTP/1.0  <CRLF>
5] GETSTREAM /~MPEG_DATA/alien.mpg?
   3000+TCP+FRMDADV+45 HTTP/1.0 <CRLF>
6] GETSTREAM /~MPEG_DATA/alien.mpg?
   3000+TCP+LOOPPLAY HTTP/1.0  <CRLF>
7] GETSTREAM /~MPEG_DATA/alien.mpg?
   3000+TCP+CONTENTSEARCH+'Search Elvis'
   HTTP/1.0 <CRLF>
```

The rest of these examples specify various playout control operations on an already open video connection. For instance, example 3, above requests a random access operation starting at 4500^{th} second from the start of the movie.

These and other enhancements[3] to the HTTP protocol have been implemented by modifying the NCSA *httpd* server, and the University of Berkeley's `mpeg_play (Version 2.0)`.

Figure 4: MPEG User Interface

The WWW based interface for this new form of video access via the web consists of an HTML page with inlined images, with associated hypertext links and text briefly describing each video sequence. A client requests a movie by clicking on the image or the hypertext link. In response the server streams the corresponding movie to the client. Also, at any given time, if the resources permit, the client can activate multiple videos simultaneously or multiple copies of the same video. The video playout interface, shown in Figure 4, allows users to specify interactive operations by clicking on appropriate buttons in the control panel. Whenever a user selects a particular interactive operation, the video client sends an explicit control command to the `httpd` server which interprets it and takes the appropriate action in response.

The communication between the server and the client takes place over two connections: a **control**

[3] such as modified Universal Resource Identifier (URI) specification

connection and a **data** connection. The control connection is implemented as a reliable TCP connection and is used for exchange of interactive playout control commands. The data connection can be one of three types: a completely reliable TCP connection or completely unreliable UDP data stream, or partially reliable connection [8]. Note that TCP is not an appropriate protocol for video transport and the only advantage of using it is that it requires minimal modifications to the client side. UDP does not provide any flow and error control and also does not support periodic data transfer and hence is unsuitable for video transport. On the other hand, a mechanism like the one described in [8], is tailored for video transmission; it provides limited retransmission based error recovery combined with error concealment and periodic data transfers in the units of video frames. Also, note that if an unreliable or partially reliable protocol is used for data transfer, the video decoder must be able to handle any losses that may result.

We have implemented the enhanced HTTP protocol above on a Pentium PC platform running NETBSD OS. On this platform, we have measured UDP throughputs close to 90 Mbps and TCP throughputs of upto 75 Mbps over a 155 Mbps ATM interface. Also, the standard UNIX filesystem throughput for a single large sequential file access from a commodity disk has been measured to be 35 Mbps. Employing striping over multiple disks, the file system throughput can be increased further. We plan to use the `ccd` software striping driver available with NETBSD for this purpose. Therefore, using network throughput numbers as the limit, each PC can support upto 15 MPEG-2 (5 Mbps) video streams. However, this number is subject to the assumption that bandwidth guarantees are available at the server on a per connection basis.

At the client side, we found that using the UNIX OS without any enhancements leads to a highly variable performance; for small videos, the software decoder can sustain approximately 20 fps on a powerful (SPARC-20) lightly loaded machine and upto $10 - 12$ fps on a SPARC-2 machine. However, for roughly quarter SIF resolution videos, the frame rate drops to $\approx 4 - 5$ fps, due to increasing decoding overhead and lack of guaranteed access to CPU and other resources. Also, at present due to lack of guarantees, video playout is jittery, clearly indicating the need for QOS guarantees.

3 Providing End-to-End QOS Guarantees

As illustrated in Figure 5, the end-to-end QOS guarantees for multimedia data requires that the server,

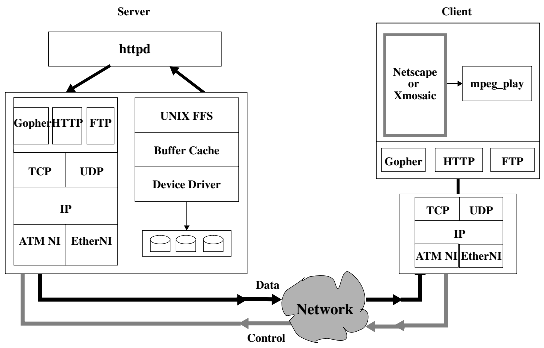

Figure 5: Components of End-to-end QoS Guarantees

the client and the intermediate network components provide such QOS guarantees. In our work, we focus only on providing such guarantees in the end-systems and assume that network either performs peak bandwidth allocation or guarantees QOS by some other mechanism. We propose to use a novel mechanism called **Real Time Upcalls** (RTU) [7] implemented in the NETBSD operating system to provide such guarantees at the client and the server side. Our approach is outlined below.

3.1 Real Time Upcalls: A Mechanism for Providing QOS Guarantees

Recognizing the growing need to provide guarantees for periodic processing within the host operating system, we have designed and implemented a Real-Time Upcall (RTU) mechanism [5, 6, 7]. RTUs are an alternative to real-time periodic threads and have advantages such as low implementation complexity, portability, and efficiency. An RTU is essentially a function in a user program that is invoked periodically in real-time to perform certain activity [7]. Various examples of such activities are protocol processing such as TCP, UDP, multimedia and bulk data processing, and periodic data retrievals from storage systems. The RTU facility allows efficient implementation of communication protocols with zero data copies and with QoS guarantees.

The RTU mechanism is implemented in a manner that does not require any changes to the existing UNIX scheduler implementation. The RTU scheduler is a layer above the UNIX scheduler that decides which RTU (and as a result which process) to run. It uses a variant of the Rate Monotonic (RM) scheduling policy. The main feature of our policy is that there is no asynchronous preemption. The resulting benefits are minimizing expensive context switches, efficient concurrency control, efficient dispatching of upcalls, and elimination of the need for concurrency control between RTUs [7].

3.1.1 Experimental Evaluation of RTUs

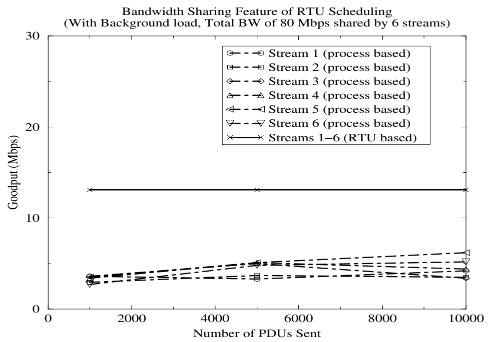

Figure 6: UDP throughput with Real-time Upcalls

We describe an experiment with the RTU facility that demonstrates its ability to provide QOS guarantees for data transfer. The experimental setup consists of two Pentium machines connected over 155 Mbps ATM. The machines run NetBSD1.1, enhanced with the RTU facility and a driver for the ENI ATM adaptor card. Six UDP "streams" were setup between the sending and receiving hosts. Each stream is a pair of processes–a sender and a receiver, that communicate over a UDP socket. In the first part of the experiment, each sender-receiver pair was implemented using the normal UNIX process mechanism. The sender program used the *"sendto"* system call to send N messages, and the receiver used the *"recvfrom"* system call to read the messages. N was varied from 5000 to $20,000$ and message size was 8 KB. In the second part of the experiment, the sender and receiver programs were implemented using RTUs. The sender RTU was upcalled every 20 msec, and it sent a batch of 4 messages. The receiver RTU was upcalled every 10msec and it read a batch of 2 messages. The batchsize and the period determine the stream bandwidth for a given fixed packet size.

Figure 6 plots the throughput for each stream measured at the receiver with respect to N. When RTU based streams are used, the throughput seen by each stream is identical and equals 13.1 Mbps giving an aggregate of 78.6 Mbps. On the other hand, when the process based streams were used, the throughput was much lower. This is because the receiver processes are not getting scheduled often enough to be able to drain their socket buffers. As a result socket buffers overflow causing received packets to be discarded. Other experiments show that RTU based streams maintain bandwidth even when other (non RTU based) processes are active on the system. More significantly, we have implemented the TCP protocol using RTUs, and our measurements show that per connection bandwidth guarantees are obtained even in the presence of background load on the systems[4]. These experiments conclusively establish that RTUs can be used both at the VOD client (decode and display) and at the server (data fetch and transmit) to provide processing guarantees.

3.2 QOS Guarantees at the Server

As shown in Figure 5, the three main tasks that the `httpd` server performs for each active video connection are: (1) retrieve data from the video file using local file system calls, (2) transmit the retrieved data over the data connection, and (3) process any control requests that are received over the control connection. Typically, processing a control command requires changes to data retrieval and transmission tasks (task 1 and 2). The first two tasks must be performed periodically for each connection to guarantee QOS and maintain regular playout, and the third task must be performed often enough to minimize interactive operation latency.

Clearly, it is necessary that the entities such as the file system, disk device drivers, network protocols, network interface driver, and the **`httpd`** daemon that handle the data as it moves from the storage devices to the external network must provide periodic processing guarantees. However, the existing UNIX system does not provide such guarantees. Figure 7 indicates the modifications to the NETBSD kernel we have undertaken to rectify this problem.

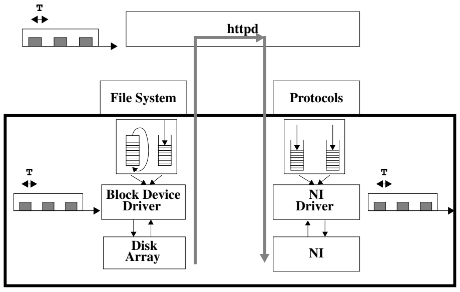

Figure 7: QOS at the Server: Modifications to OS

Existing disk device drivers in UNIX do not treat requests for multimedia data differently than ordinary data requests. So we are currently modifying the SCSI disk driver to recognize two types of requests: real-time periodic, synchronous requests and non-real-time aperiodic, asynchronous requests. The synchronous requests are scheduled in a separate request queue every scheduling period T_c, whereas the asynchronous requests are queued on an as-and-when-received basis. The synchronous request queue is always serviced before asynchronous requests. The service discipline used for the synchronous requests must optimize two conflicting objectives: meet deadlines of the requests and optimize disk utilization. We plan to use a modified rate monotonic scheduling scheme for synchronous requests, and a vacationing server scheme for asynchronous requests. Note that the non-pre-emptible nature of disk I/O complicates these service disciplines.

Similar to the disk driver, the current ATM network interface driver allows only asynchronous requests, and does not provide guarantees on transmissions. We plan to fix this problem in the following way: First, using the hardware capability of the network interface, we will isolate data enqueued by different connections into separate queues; one for each active connection. Also, the connections will be classified into real-time and non-real-time connections. The network interface driver will be modified to schedule periodic transmissions for real-time connections first, followed by the asynchronous data transmission requests for non-real-time connections. Also, the protocol processing associated with TCP, UDP, or other protocols will be carried out with QOS guarantees using the approach outlined in [4]. Note that in case of network interface driver, the service disciplines for request queues must optimize the following objectives: meet deadlines and rate specifications of transmits for each synchronous request and minimize delay for asynchronous requests. Similar objectives apply in the case of the receive data path.

We employ RTUs in both the drivers and the `httpd` code to implement periodic processing model as shown in the Figure 7.

3.3 QOS Guarantees at the Client

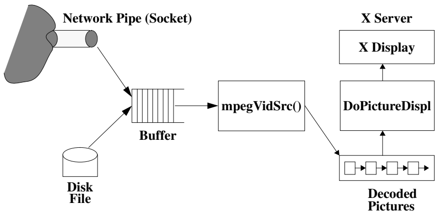

Figure 8: High Level Structure of `mpeg_play`

Figure 8 shows the conceptual organization of the Berkeley `mpeg_play` Version 2.2 software. In its standard form, the decoder can be considered to consist of a data buffer, an X display and four main functions: `get_more_data()`, `mpegRVidsrc()`, `DoPictureDisplay()` and `ControlLoop()`. The `get_more_data()` function reads data from a disk file into the data buffer. The `mpegRVidsrc()` performs the actual decoding of the MPEG data and puts the decoded frames on a ring of frames. This function uses a set of other functions which implement the actual MPEG-1 decoding algorithm. Typically, during the process of decoding when the buffer underflows, these functions call `get_more_data()` to read data from the file. The `DoPictureDisplay()` function removes the decoded frames from the ring of frames and displays them using the Xlib functions. The `ControlLoop()` function processes the interactive commands from the user.

The QOS guarantees required by a MPEG decoder are in the form of periodic reception and decoding of data, and display of decoded frames every $\frac{1}{f}^{th}$ of second for a frame rate of f fps. In our current implementation we use an RTU to periodically invoke the three functions outlined above. We have also modified the decoder to read data from a network data connection using the `socket` interface.

4 Experimental Results

In our experimental setup, we used two 133 MHz Pentium PCs with ENI ATM interface cards connected to a Bay Networks ATM switch. The modified *httpd* server capable of streaming video data was run on the server PC with several small MPEG movie clips stored on its local disk. The file system and the device drivers modifications proposed in Section 3.2 have not yet been incorporated. The MPEG client uses the modified HTTP protocol as outlined in 2. It also uses an RTU to obtain guaranteed share of CPU. Both the control and data connection between the *httpd* server and the MPEG client were standard TCP connections with large (190 KB) socket buffers. We performed experiments to evaluate how well the RTU based MPEG decoder provides QOS guarantees as the load on the client machine is increased. The load was simulated by running the `primes` program available in the standard UNIX `games` directory. This program computes prime numbers and prints them to the `/dev/null` device. We compared this decoder with an MPEG decoder that uses network data connections but does not use RTUs (called the process based decoder).

These experiments were run on several sample video sequences. Here we report results on two sample videos, both of which have plenty of motion, some scene changes and reasonable length. These were: a 320×288 (PAL) size US Space Shuttle Landing movie `sts74.landing.ibp.mpg` (referred to as Clip-1), with

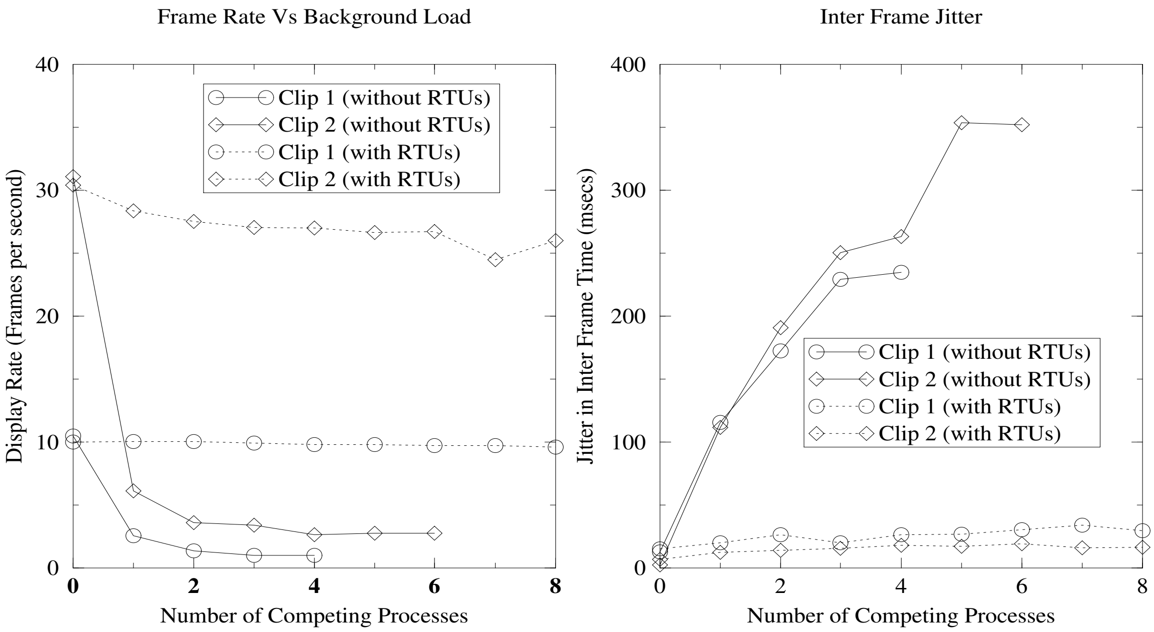

Figure 9: Performance of MPEG Player

1342 frames ($\approx$ 45 seconds), and 160×120 size movie `ToEdgeEinstein.mpg` (refered to as Clip-2), with 1730 frames ($\approx$ 58 secs).

The first part of Figure 9 illustrates for these two videos, the observed frame rate as the load is varied. Both the RTU based decoder and the process based (non-RTU) decoder are shown. It can be clearly seen that both the decoders perform comparably in absence of background load. For instance, both the decoders achieve roughly 10 fps for Clip-1, whereas for Clip-2 almost 30 fps is obtainable.

However, as the background load is increased, the process based decoder suffers dramatically. Even with a load of one competing process, the frame rate drops from $\approx$ 10 fps (in the no load case) to $\approx$ 2.6 fps. With higher loads, the frame rate drops to an intolerable $\approx$ 1 fps. Similar behavior is observed with the Clip-2 as well. On the contrary, the RTU based decoder performs remarkably well. In case of Clip-1, it achieves almost a constant $\approx$ 10 fps, and with Clip-2 $\approx$ 30 fps can be obtained. Thus, in the case of full load, the RTU based decoder provides $\approx$ 950% improvement in frame rate.

The second part of Figure 9 shows the standard deviation of inter-frame display time which is a rough estimate of the frame jitter. It can be clearly seen that for the process based decoder the jitter increases dramatically as the load is increased. This coupled with drop in average frame results in a very slow and jittery playout. On the other hand, the RTU based decoder performs exceedingly well with very small inter-frame jitter.

Limitations of the X Server. Although the RTU based decoder is largely unaffected by background load, we can see that there is a slight drop in frame rate, and an increase in inter-frame jitter as the load is increased. The explanation for this lies in the manner in which the decoder and the X server interact. The decoder sets up a shared memory area with the X server into which it writes the decoded images. The shared memory is organized as as ring of `Ximage` structures each of which can hold a decoded frame. The decoder passes the current frame to the X server, to be displayed in the appropriate window. It then waits until the X server has finished displaying the requested frame so that it can safely reuse the shared memory to store subsequent decoded frames. Since the X server is an ordinary UNIX process, it competes with other active processes for access to CPU and runs without QOS guarantees. Thus, the display event can take variable amount of time as the load increases. This results in back-pressure on the decoder, and subsequent drop in frame rate.

In all our experiments, the X server was run with highest priority (-20) permitted by the UNIX scheduler, to minimize this effect. This clearly indicates that real-time software video/audio decoding with excellent QOS guarantees requires that display handlers such as the X server must provide ways to request real-time display of multimedia data.

Another practical issue concerns the interaction of preemptive real-time scheduling mechanisms and the X library. The X library, as of now, is not reentrant. Thus two or more preemptible threads of control, that independently call into the X library will cause the program to crash unless they take some concurrency control measures. In our case, the main program thread calls the X library to process user events. It can be preempted by the RTU which also calls the X library to display decoded frames.

To prevent the RTU handler from preempting the main program while it is in the X library, the main program simply sets a global flag before accessing X. The RTU handler is expected to check this flag before making any X calls. This simple scheme may not work for other real-time mechanisms such as threads because of asynchronous preemption. Such implementations must therefore use more sophisticated and expensive mutual exclusion mechanisms to deal with the reentrancy problems. So in our experience, the RTU mechanism allows for simpler as well as more efficient programs on existing software platforms.

Barriers to Getting High Frame Rates. Our experiments also throw light on the limited computing power available on even very fast (133 MHz) Pentiums. We see that as the video size is increased from 160×120 to 350×240 the frame rate drops from 30 to 10 fps. This reduction could either be due to the decoder not getting data fast enough over the TCP connection, or due to reaching the limit of the CPU computing power. To determine the cause, we measured the number of times the decoder blocks on data reads from the network. We found them to be zero, clearly indicating that frame rate is limited by the decoding overhead which directly depends on the CPU capacity. The MPEG-1 streams we used in our experiments require 4 times less bandwidth than MPEG-2 streams. This proves that unless serious improvements in processor speed and/or architecture[4] are made, software decoding of full size, full motion MPEG videos may be an elusive target.

5 Conclusions

In this this paper we outlined a system architecture for scalable inexpensive multimedia-on-demand storage server based on commodity components. This server provides a WWW based access interface using an enhanced HTTP protocol. We also described our approach to providing QOS guarantees within the server and client end-systems, based on a novel mechanism called Real Time Upcalls (RTUs). Our experiments conclusively prove the inadequacy of the existing UNIX scheduling for periodic decoding and display of video under all load conditions. We have shown that this can be achieved easily using RTUs. To achieve end-to-end QOS guarantees, we are employing RTUs to provide guaranteed disk bandwidth and network protocol processing for each active client at the server.

[4] Addition of video processing specific instructions to CPU instruction set, similar to ones in UltraSparc.

References

[1] Buddhikot, M., Parulkar, G., M., and Cox, Jerome, Jr., "Design of a Large Scale Multimedia Server," *Journal of Computer Networks and ISDN Systems*, Elsevier (North Holland), pp. 504-524, Dec 1994.

[2] Buddhikot, M., and Parulkar, G., M., "Efficient Data Layout, Scheduling and Playout Control in MARS," Invited for publication in the *Special Issue of ACM/Springer Multimedia Systems Journal.*

[3] Dittia, Z., Cox., J., and Parulkar, G., M., "Design of the APIC: A High Performance ATM Host-Network Interface Chip," *Proceedings of the IEEE INFOCOM'95*, pp. 179-187, Boston, April 1995.

[4] Gopalakrishnan, R., and Parulkar, G., M., "Efficient User Space Protocol Implementations with QOS Guarantees using Real-time Upcalls," Dept. of Computer Science Technical Report WUCS96-11, Washington University, St. Louis, MO.

[5] Gopalakrishnan, R., Parulkar, G.M., "A Framework for QoS Guarantees for Multimedia Applications within an End-system," *Swiss German Computer Science Society Conf.*, 1995.

[6] Gopalakrishnan, R., Parulkar, G.M., "A Real-time Upcall Facility for Protocol Processing with QOS Guarantees," (Poster) *ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, Dec. 1995.

[7] Gopalakrishnan, R., Parulkar, G.M., "Real time Upcalls," Tech. Rep. WUCS-95-06, Washington Univ. Mar 95.

[8] Papadopoulos, C., and Parulkar, G., M., "Retransmission based Error Control for Continuous Media Applications," To appear *Proceedings of NOSSDAV96*, Japan, Apr. 1996.