Multi-Platform Simulation of Video Playout Performance *

Ladan Gharai and Richard Gerber
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland
College Park, MD 20742
{ladan,rich}@cs.umd.edu

Abstract

We describe a video playout and simulation package, including (1) a multi-threaded player, which maximizes performance via asynchronous streaming and selective IO-prefetching; (2) a compositional simulator, which predicts playout performance for multiple platforms via eleven key deterministic and stochastic time-generating functions; and (3) a set of profiling tools, which allows one to extend the range of target platforms by benchmarking new components, and converting the results into distribution functions that the simulator can access. Using this system, a developer can quickly estimate a video's performance on a wide spectrum of target platforms – without ever having to actually assemble them.

1 Introduction

Digital video is currently used in a variety of applications, such as instructional software, computer games and multi-media presentations. While the developers of these applications usually possess fairly powerful computing platforms, the end-user systems can vary to an enormous degree – e.g., in CPU performance, bus speeds, IO transfer rates, etc. And since software video playout tends to monopolize all of these components, developers usually down-sample their end-result to a set of "typical" target platforms. In the ideal world, this would mean actually running the released application on a wide variety of consumer workstations, and then making successive adjustments to the encapsulated videos – so that the result satisfies the largest number of users. Of course, this ideal is almost never pursued, for two obvious reasons: (1) it would require a lab to purchase a huge number of (possibly obsolete) consumer platforms, and then (2) to spend a large amount of time adjusting the videos to them. In the real world, developers usually test the application on one or two targets, and perhaps augment testing with some rough bit-rate calculations to predict performance on other potential targets.

In this paper we present a cheap, fast alternative to this process, which uses discrete-event simulation, in concert with an abstract model of the playout platform. The model characterizes the datapath of our QuickTime [3] playout software (described in [6]), and its operation on specific CPU/IO configurations. The simulator's inputs are (1) a specific CPU workstation type, (2) a SCSI device model, and (3) a Quicktime video header file. These inputs, in turn, select eleven stochastic

*This research is supported in part by ONR grant N00014-94-10228 and NSF Young Investigator Award CCR-9357850.

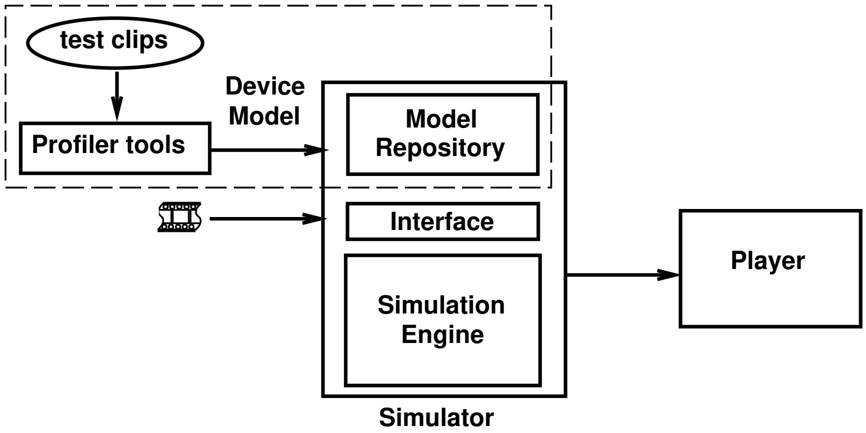

Figure 1: Simulator Package Architecture.

and deterministic time distribution functions – which are then compositionally assembled to gauge frame-by-frame playout performance.

One reason we can achieve a reasonable level of accuracy is as follows: our playout software supports its own API, and bypasses Quicktime's high-level, movie-playing functionality. On one hand, commercial APIs like those provided by QuickTime can simplify a program's interaction with the underlying codec software. But on the other, their long, multi-layered call-paths tend to cause an enormous amount of jitter and bursty frame-dropping at display time. In [6] we describe our alternative implementation, which bypasses all high-level API functions and only interacts with QuickTime's codec support. Hence, our architecture manages all system-level details involved in video playback, such as IO transfers, memory management, buffering, flow control and audio/video synchronization. In our experiments, this resulted in enhanced, more deterministic performance, which scaled to different classes of workstations and input videos. The results were achieved by some simple optimizations, including: (1) prefetching via temporal locality information, i.e., using past performance to estimate which future frames should be selected; (2) coalescing IO requests for neighboring physical frames, which reduced the level of kernel involvement in IO transfers; (3) eliminating all memory-to-memory copying of frames, and (4) taking full advantage of DMA'd, asynchronous IO, which allowed the CPU to concentrate on codec-related activities. By applying these design principles, our software achieved performance gains of up to 345% over Apple's high-level video playout abstractions, with associated improvements in rate variance – while relying on the exact same codec drivers.

Nonetheless, the actual performance of a video on our system (and on any other system) is highly sensitive to the underlying platform configuration – as well as to variations in the video itself. The subtle interaction between system components (e.g., CPU, internal bus, VRAM, SCSI IO device) and between video characteristics (e.g., frame dimension, compression ratio, codec type, color depth) all effect the ultimate playout performance. Thus, it is hardly possible to obtain an analytical prediction in terms of average video bit-rates, IO transfer rates, and processor types. Alternatively, a purely stochastic simulation model would also lead to unpredictable results – after all, many of the underlying time distributions have highly deterministic properties. For example, a given frame's transfer time from any IO device will, to some degree, be roughly proportional its size. Hence, our simulation model contains some distributions that are principally deterministic, and others which are only stochastic.

Figure (1) displays the architecture of the simulation system. The system is composed of (1) a set of profiler tools, (2) the simulation engine, and (3) the playback software. The profiler tools automatically benchmark different parts of the playout datapath, by stressing them with a series of test videos. The resulting time distributions are indexed according to the main physical components involved – the specific IO device, and the CPU type. These "virtual component" models, in turn, include other parameters, which are indexed primarily via a movie's header information (e.g., codec, frame size, frame type). The synthesized device models are then put into the simulator's "plug-in" directory, and are accessed during a simulation run. At that time, the models are combined to create a system configuration. This design style has the following advantages:

1. A developer can "virtually" configure different playback platforms, and test a specific video's performance on each. The actual platform need not be present in the laboratory – all the developer needs is the model directory.

2. The simulator is a "value-added" application, since it can easily be extended by running the profile tools on a new IO or CPU device.

For example, a user may have three options of SCSI IO devices, with average transfer rates of 5Mbytes/sec, 4Mbytes/sec and 1900Kbytes/sec, respectively. For a given processor type, and a given video, the actual performance of each configuration can be quickly obtained by doing three simulation runs – which usually requires a few seconds.

The remainder of this paper is organized as follows. In Section 2 we discuss some of the related work in the field. In Section 3 we present an overview of the components of our simulation package. Then, in Section 4, we describe the profiling process, and how the profiled results generation time distribution functions. In Section 5 we compare the simulator's predictions for selected configurations with corresponding on-line playout performance. Finally, we give some concluding remarks in Section 6.

2 Related Work

In this paper we concentrate on simulating the entire playout process; hence we model many of the components involved at a fairly coarse level. Alternatively, other researchers have constructed detailed models of certain key components, and then subjected them to simulated video workloads. In particular, IO subsystems have often been studied in this manner. For example, the results in [8] illustrate the performance of various disk-scheduling algorithms, when tested with synthetic video simulations. For single video streams, decent performance was shown to be realized by both CSCAN and SCAN-EDF – a hybrid of the traditional SCAN technique, and the "earliest-deadline-first" strategy used in real-time thread schedulers. Another technique, the Group Sweeping Scheme [5], is a hybrid of round-robin and SCAN. A number of "groups" are scheduled via round-robin, whereas within each group the SCAN algorithm is used. To a large extent, this allows for compromising between the disk-head's ability to "sweep up" physically neighboring blocks, and the temporal requirements imposed on concurrent, time-based media streams.

We do not capture this aspect (and other aspects) of the IO architecture, for two basic reasons. The first is fairly simple: if one wishes to use off-the-shelf disks, then one must live with the vendor-supplied, proprietary policies which are hard-coded into controller's micro-program. (On the positive side, however,disk controllers are increasingly being optimized for "multimedia systems" – which usually translates into good "sustained" read/write performance over contiguous blocks.) However, the second reason we abstracted out disk-scheduling – as well as internal disk caching – is somewhat more subtle: we found that for single-client workloads, details like these are "smoothed out" during the simulation's datapath; especially when the end-point, monitored process records frame-display times. As we show in the sequel, for our purposes an IO device can be sufficiently modeled by its rate and latency distributions.

Although described in the context of networked video, Stone and Jeffay's [10] queue monitoring method is quite similar to the way our playout loop manages jitter. As they have found with networked traffic – and as we have found in dealing with IO and compression software – a balance must be found between a stream's jitter and its delivery rate. They prescribe a feedback policy to dynamically adjust of display latency, which supports low-latency conferences with acceptable gap-rates.

A related issue is achieving graceful degradation of service in the event of network congestion. One approach to this problem is for the client to adaptively scale the playback rate by deterministically dropping some of its frames. This is the approach taken in the the Nemesis [7] project, which uses a *predictive prefetch* algorithm to scale a client's input streams. This is also the approach taken in Vosaic [11], which uses its own specialized a real time variant of UDP, and allows the server to scale its transmission by the feedback it receives from the client. As we show in the following section, these techniques are similar to our system's feedback loop – however, since we concentrate on single-client streams, the rate adjustments can obviously be made faster.

Chen and Kandlur describe a player in [2], which, like ours, is a stand alone client station player. However their emphasis is on supporting VCR playback capabilities, such as forward and backward playback for an MPEG encoded video stream. For display, a video is first downloaded from the server in entirety; then during playback the stream is converted to a local form, by separately segregating all P frames and I frames. In this manner, a rough backward playback is simply a matter of displaying a series of I frames.

The system described in [1] scales not only the rate, but also the spatial resolution of a video stream. This is done by packaging three versions of every frame, with each offering a monotonic improvement over the previous one. The first is a 160x120 abstraction of the original picture; the next is the residue term which, when added to the 160x120 image, achieves a resolution of 320x240. The final version is another residue which can be added to the 320x240 image, resulting in full 640x480 resolution. At any point in the process the codec can stop improving the current frame, and proceed to the next. Of course, this flexibility is achieved by using a custom codec, which was designed specifically for this purpose.

Our focus on IO and data paths is echoed in [4], which proposes a means of optimizing the transmission of compressed videos. A *splice* mechanism is introduced, in which an application can associate a kernel-level data source with its sink point; this allows for a direct point-to-point data path between source and sink, obviating unnecessary kernel interference.

3 System Overview

Our simulation engine was designed to *quickly and accurately* predict video playback performance on a range *specific* platform configurations. To achieve this goal, we have to balance several competing objectives. First, our model should be sufficiently abstract to produce quick results, yet sufficiently fine-grained to yield accurate information. Second, we require a way to isolate the roles of the particular CPU type, the SCSI device, and the video characteristics – and reproduce their individual effect on playback performance via simple time-distribution functions. Third, we need a means of combining and scaling these separately generated functions into simulated on-line behavior.

In this section we give an overview of our approach. First, we discuss the datapath and semantics

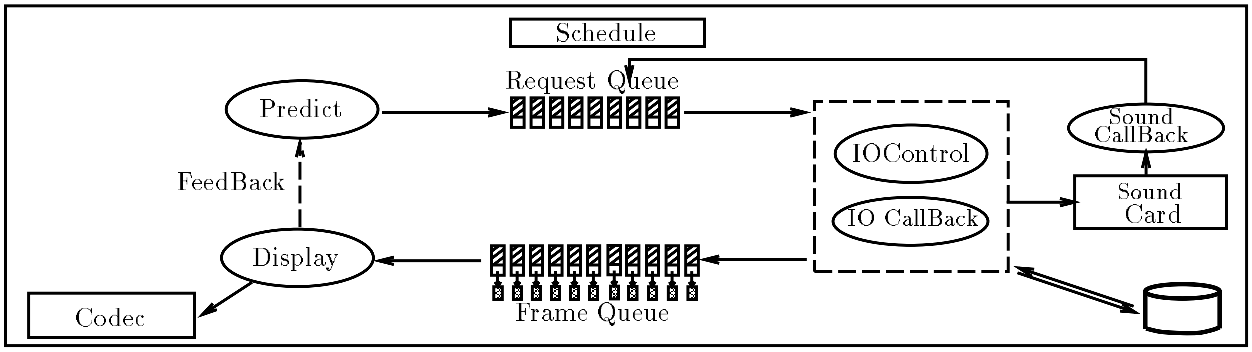

Figure 2: Our software playback system.

of our playout software. Then we discuss our simulation model, and describe how we have structured the time-generation input files.

3.1 The Player Structure

Figure 2 depicts the structure of our playout software, which is composed of three threads and two callback functions. As the figure shows, the system operates as a simple feedback loop. Based on the player's past performance, the Predict thread selects a set of frames to be played in the future, and inserts their IDs into the Request Queue. The IOControl thread removes them, looks up their corresponding file locations, and initiates the appropriate asynchronous IO commands for the SCSI manager. If there are requests for (physically) neighboring frames, the IOControl thread will attempt to bundle as many neighbors as possible within a single IO transfer. By bundling adjacent frames the IOControl thread (and all associated SCSI handlers) can execute less frequently; this CPU time can be better used by activities such as decompression.

When an IO operation completes, its associated callback function preemptively executes, like an interrupt service routine. It inserts a pointer to each transferred frame into the Frame Queue. They subsequently get removed by the Display thread which, according to its real-time movie clock, will either display the associated frames, or just discard them. If the frames have been delivered on time, then they get decompressed by the codec.

Part of the Display thread's job is to update feedback information for the Predict thread, which is done at every time interval Δ. (For the results displayed in the sequel, Δ was set to $\frac{1}{2}$sec.) The feedback is in the form of a predicted playout rate for the next interval, and the Predict thread uses it to dynamically scale its prefetch rate. This scheme is partially aided by the scheduler, which ensures that the Predict thread gets at most most $\frac{1}{2}$ second ahead of the Display thread.

Let t be a multiple of Δ, $\mathcal{R}_{\mathrm{MOVIE}}$ the digitized rate of the video, and let $\mathcal{PR}(t)$ be the predicted playback rate for the time interval $[t, t+\Delta]$. Then if we let $\mathcal{R}(t)$ be the rate that the Display thread *actually* achieved during the interval, $\mathcal{PR}(t+\Delta)$ is calculated as follows:

$$\mathcal{PR}(t+\Delta) = \left[\begin{array}{ll} \alpha \times \mathcal{R}(t) + (1-\alpha) \times \mathcal{PR}(t) & \text{if } \mathcal{R}(t) < \mathcal{PR}(t) \\ min(\mathcal{R}(t) + c, \mathcal{R}_{\mathrm{MOVIE}}) & \text{if } \mathcal{R}(t) = \mathcal{PR}(t) \end{array} \right.$$

where for the experiments reported in this paper, we set $\alpha = .85$ and $c = 1$. In other words, when playback falls behind its predicted rate, we exponentially average the old prediction with the achieved rate. (This is to smooth out sporadically large frame sizes, or abnormally high decompression times.) But when playback meets its prediction, we gradually ratchet up the new prefetch rate, so that eventually the highest potential quality can be realized.

The objective of our design is to let the system achieve a steady state, so that IO and playback are always working in parallel, at their full capacity. This means the Display thread should never have to wait for a frame – the IO should always have prefetched it ahead of time, while the Display thread was processing a previous frame.

Keyframes. Our scheme is complicated by the existence of keyframes (which are analogous to I-Frames in MPEG [9]); e.g., if a keyframe is dropped, then the the interpolated sequence following has to be discarded. Thus, while $\mathcal{PR}(t)$ is the current predicted rate, the Predict thread cannot simply fetch frames at a constant frequency. First a decision is made whether an entire sequence will be avoided. If not, its keyframe is requested, as are selected interpolated frames within the sequence. The Display thread may end up only decompressing – but not playing – the keyframe, so that it can be used to display its dependent, interpolated frames.

Sound. The player software interacts with the sound card via a simple double-buffering scheme. When one buffer is almost finished being played, the sound card triggers a callback routine, and then switches to the other buffer. The sound callback routine places IO requests for sound samples on the Request Queue. Unlike video, sound samples cannot be dropped, and so the IO thread gives them priority.

3.2 The Simulation Engine

The simulation engine basically mimics the control and data paths of the player software itself. The threads are represented by discrete events – called `predict`, `display` and `IOControl` – with time distributions corresponding to their CPU cost per one service, on a specific host. Additional events represent playback activities like decoding (`decode`), SCSI device callbacks (`IOcallback`), and sound card callbacks (`SNDcallback`). The final two events are assigned to the scheduler's activity (`schedule`) and context-switch overhead (`switch`). Events are placed on a "ready queue," and dispatched by the same protocols as their system-level counterparts. For example, the events corresponding to scheduler-controlled actions (e.g., threads) are dispatched in FIFO order, while asynchronous, device-related events (e.g., `IOcallback`, `SNDcallback`) execute via a specific firing time. The simulator's internal clock is updated after an event is completed.

The raw output of a simulation run is a list of all frames processed, accompanied by flags denoting whether the frames were played or dropped, and if played, their simulated display time. From this list, the tool produces several statistics, including the movie's mean playback rate, and its rate variance over 1-second intervals. Also, if a developer wishes, the display-time list can be used as input to a previewer, which allows watching the movie from the context of the simulated workstation.

3.3 Platform Profiling

The profiling tools create hierarchical models of the IO device and CPU platform, which are collected in the simulator's repository. Before a simulation run, the user "assembles" a platform by selecting particular models of CPU and SCSI devices. Then, the simulation engine interacts with these models to update its clock during the run.

Each device model is, in fact, a collection of time generating functions. Figure (3) displays the directory hierarchy which the simulator accesses. The IO model contains a single distribution function, which captures both the deterministic and stochastic properties inherent in the device's transfer time. The CPU model, on the other hand, is composed of distribution functions for the player software itself, in addition to a set of codec-related decompression times for all codecs profiled

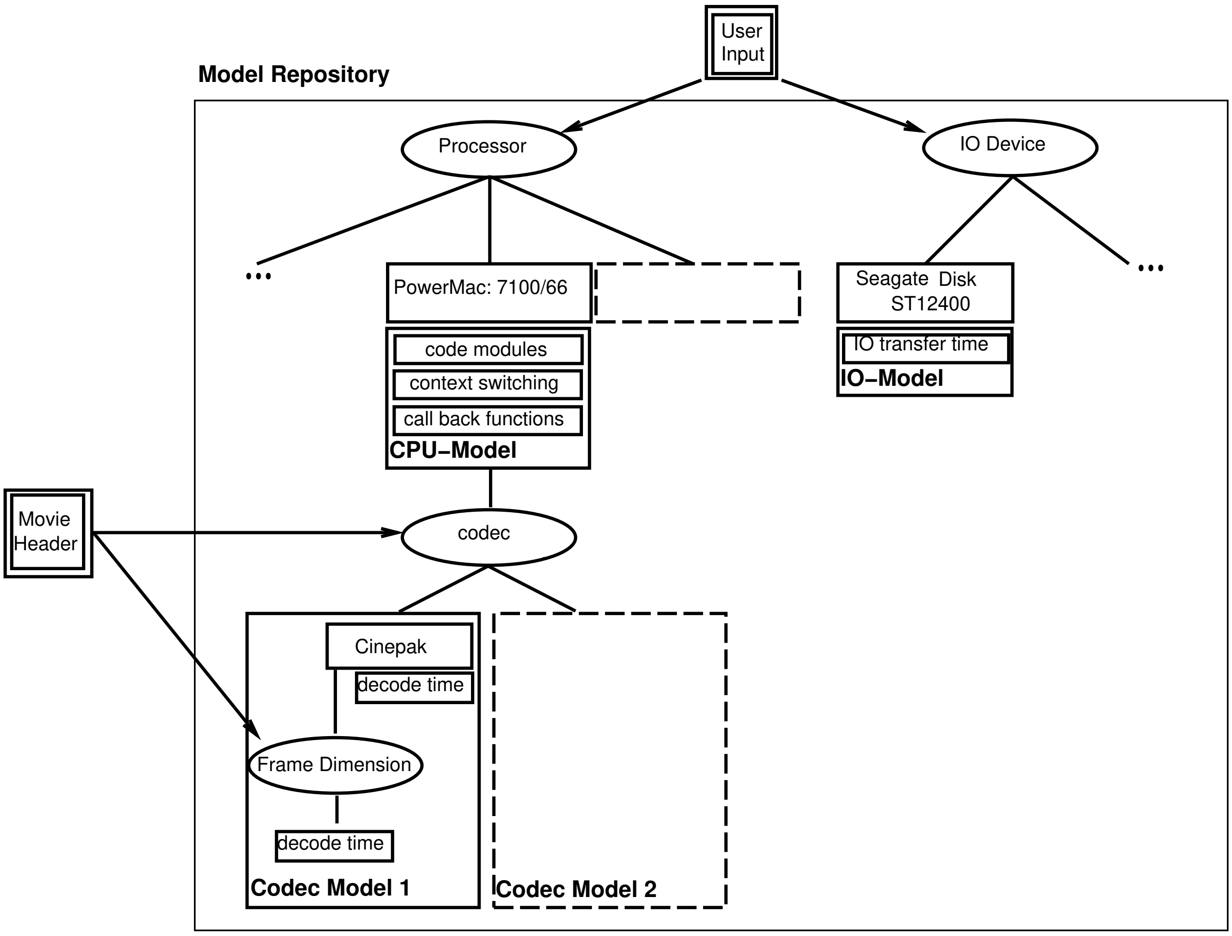

Figure 3: Hierarchy of virtual device models. Users select IO and CPU devices. The simulator extracts codec-related statistics, using the input movie header.

on the particular CPU.

In the next section we discuss these distribution functions in detail, and show how the IO and CPU models are composed to produce a single system model.

4 The Time Distribution Functions

The simulator's eleven time variables (listed in Table 1) range over values produced by their respective time-generating functions. Some of these have deterministic inputs (as well as stochastic residuals), while others are purely stochastic. Table (2) categorizes the time variables by their underlying distribution functions, and lists their deterministic indices if they possess any.

Each function is generated by its corresponding profiler tool, which collects a trace of discrete time-samples, and indexes each sample with its deterministic input (if any). The profilers for CPU-centric variables are basically modified versions of our playback software, which collect the response times of each player component in isolation. The entire process is controlled by an executive, which repeatedly feeds a number of test clips into each profiler, until all of the sample traces have been

Model	Variables
IO	
IO transfer time	T_{io}
CPU	
Display execution time	T_{ply}
Predictor execution time	T_{pre}
IOControl execution time	T_{IOc}
Schedular execution time	T_{sch}
IO callback execution time	T_{io-cb}
Context switching overhead	T_{cnts}
Sound callback execution time	T_{snd-cb}
Movie Header/codec	
KeyFrame decode time	T_{key}
Intermediate frame decode time	T_{intr}
Sound chunk play time	T_{snd}

Table 1: The simulator interacts with these eleven random variables to update time.

constructed. The test clips were produced to cover a range of video characteristics (codec, frame dimension, color depth), and they vary in content (range of motion, color intensity).

The IO profiler, on the other hand, is independent of any playback activity, and simply measures asynchronous SCSI transfer times (indexing the results by the size requested).

Time Generator	Function	Time Variables	Steps	Deterministic Index (x)
Deterministic + Stochastic	$L(x) + R^{-1}(u)$	T_{io}	10^3	Number of bytes transfered
		T_{intr}	10^4	Size of compressed frame
		T_{ply}, T_{pre}, T_{IOc}, T_{io-cb}	10^3	Number of iterations
		T_{snd}	10^3	Size of sound chunk
Pure Stochastic	$F^{-1}(u)$	T_{key}	10^4	none
		T_{sch}, T_{snd-cb}, T_{cnts}	10^3	none

Table 2: Time variables and their distribution functions.

Time sample post-processing is handled in two different ways, depending on whether the underlying variable has a deterministic component. We summarize the two methods here:

Pure stochastic variable: In this case the time-sample list is sorted as a histogram – divided into either 10^3 or 10^4 buckets (depending on the range and variation of the recorded process). Then, the histogram is normalized to the interval $[0, 1]$, which yields a (synthesized) discrete probability distribution function (or pdf) $f(t)$ for the variable, where we now assume that a given outcome is made as a simple Bernoulli decision. I.e., $f(t)$ returns the probability of a sample time t being realized during playback on the device. Next, f's cumulative distribution function $F(t)$ is produced, and the output of the entire process is $F^{-1}(u)$, the CDF's inverse transform, where u is uniformly distributed in $[0, 1]$. The simulator uses this function to generate random response times, in concert with a dedicated random number generator.

Deterministic/stochastic variable: The sample trace is linearized by its deterministic index,

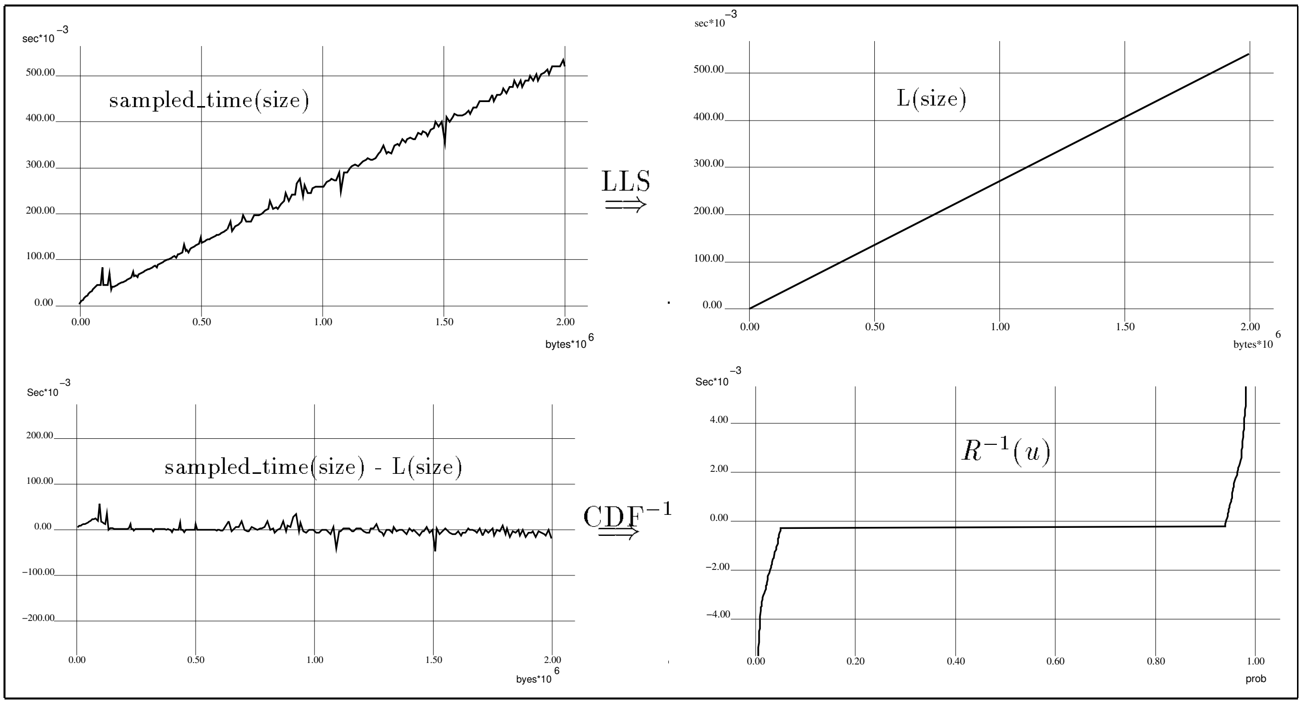

Figure 4: IO time-sample postprocessing, clockwise from top left: (1) Recorded response times for requested sizes; (2) Linearized response times; (3) Residual function; (4) Inverse-transform of residual CDF.

via a least-squares fit. (E.g., in the case of IO transfer, the profiler creates a line capturing the relationship between transfer-time and request-size.) Then, it generates the the residual deviations between the synthesized line and the recorded samples, sorts them in histogram form, and creates the corresponding inverse CDF, as in the stochastic case.

4.1 Profiling and Component Time Distributions

(1) IO Profiling: A device's IO datapath is abstracted using a single variable, indexed by the requested transfer size. This includes all factors such as the device's head latency and rotational delay, its internal caching mechanisms, the SCSI control signals, the SCSI-IO bus latch setup times, the DMA control, all the way to the end-point – which is the final DMA interrupt at the host. While this model may seem somewhat coarse, we have consistently found that the transfers involved in video are large enough to "smooth out" the effects of intermediate hops along the IO datapath. In other words, pure transfer times typically dominate the cost of the device's response overhead – and the intermediate factors can be treated in a stochastic fashion.

Our IO profiler samples sustained transfer times by requesting a variety chunk sizes, which start at 2 Kbytes and increase in 2 Kbytes increments up to 2 Mbytes. A given request starts at a random disk location, and measures the chunk's elapsed response time, from the initial IO call to the the first statement in the callback routine.

Figure 4 illustrates the post-processing that takes place after all samples are collected. First, the least-squares fit $L(size)$ is constructed from the list of response times and their corresponding sizes. Then, a residual sample function is created, i.e., for every time sample $(size_i, time_i)$, the distance r_i from the line is calculated:

$$\forall(size_i, time_i) : r_i = time_i - L(size_i)$$

Then, from the frequency histogram for the r_i, the inverse CDF-transform $R^{-1}(u)$ is generated. Finally, at simulation time, we get our IO time variable as follows:

$$T_{io} = L(size) + R^{-1}(u)$$

where "$size$" is the number of bytes requested by the simulated IO thread, and $u \in [0, 1]$ is obtained via a random-number generator.

(2) CPU Profiling: The CPU-based playout software itself is modeled by the variables T_{ply}, T_{pre}, T_{IOc}, T_{io-cb}, T_{snd}, T_{sch}, T_{snd-cb} and T_{cnts}, which represent execution times for the different segments active during playout. These variables do not cover that wide a range of variation, and their corresponding pdf's are discretized into 10^3 steps each.

Among these variables, T_{sch} (scheduler execution time), T_{snd-cb} (sound card's ISR callback), and T_{cnts} (context-switch overhead) are all modeled in a stochastic manner, and their variation is due mainly to second-order factors like cache affinity, pipeline state, etc. On the other hand, the thread processing times – T_{ply}, T_{pre} and T_{IOc} – have deterministic components, and they depend on the number of activities performed during one scheduled service of the thread. For example, the Predictor may issue anywhere from 1 to 25 frame requests, and this number does have an impact on the CPU time used. There are similar factors which affect the cost of the Display and IO threads.

The remaining CPU-based variables, T_{IOc} and T_{snd}, also have deterministic components. The IO callback's execution time, T_{IOc}, is deterministic in the number of frames it places on the Finished Queue. Also, the sound playout time T_{snd} is proportional to the number of samples in a given sound chunk.

(3) Codec Profiling: While the decode times are also processor-dependent, their distributions are grouped according to the specific codec used, the video's frame dimensions and color depth. The profiler instantiates these variables, and then gathers frame-by-frame decode times for several test videos corresponding to the current instantiation. In these tests, IO and decompression are performed serially, thus isolating decode time as a dedicated process. This is done by repeatedly transferring sequences of frames into memory, then freezing all IO driver activity, and then measuring the frame-decode response times. This continues until all frames in the test clips have been decompressed and measured.

There are two time variables associated with decompression – T_{intr} for intermediate frames, and T_{key} for keyframes – and they have markedly different distribution functions. As for T_{intr}, we have consistently observed that QuickTime's interpolated-frame decode times are directly related to the size of the compressed frame. Hence, T_{intr} is generated in the same manner as T_{io}: The profiler first compiles a list of frame sizes and their corresponding decompression times; then it generates the least-squares fit, and finally it produces the residual's inverse-CDF.

As for T_{key}, its values form a pronounced step-function, indexed by the displayed frame dimensionality – along with some stochastic noise at each step level. Hence, this leads to separate time functions for different frames dimensions (we currently only profile 4:3 dimensions, e.g., 640x480, 320x240 and 160x120). Our profiling tool collects sets of time samples at the three modeled dimensions, and then produces their inverse CDFs to represent the noise at each level. Thus, in Table 2, T_{key} is shown as a purely stochastic variable, since it is indexed in a "global" sense, and used for

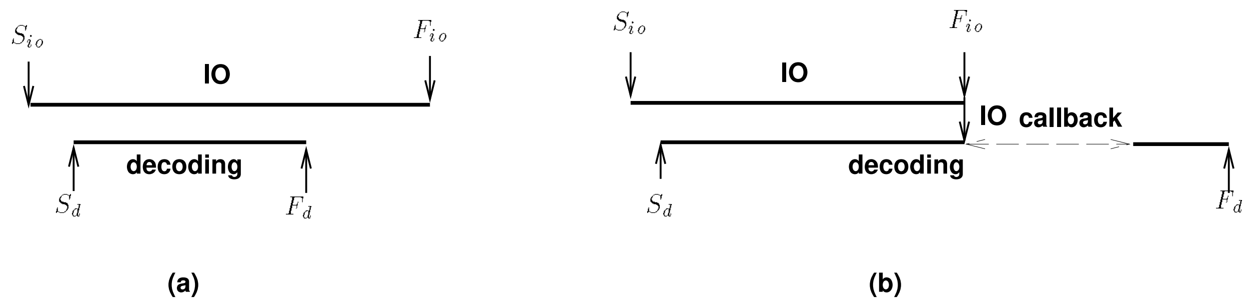

Figure 5: (a) IO completely overlaps decoding; (b) IO completes during decoding; decoding is interrupted by the IO callback function.

the duration of a video run.

4.2 Composing CPU and IO: Interference Modeling

While IO transfers and CPU activities are profiled separately, the simulator has to compose the two models to "build" an abstract platform. For many parameters (such as thread-execution times), this is simply a matter of using the two separately generated distributions in a given simulation run, without adjusting either of them. Unfortunately, this is not the case for decode times, which use most the CPU and memory cycles available – and are thus quite sensitive to interference cause by IO transfers. Recall that most decode activities are pipelined with IO – and these two functions share many of the same resources. For example, often the SCSI driver has to handle multiple block-based DMA interrupts to assemble frames, and the DMA itself steals memory cycles from the codec software. Hence, when decoding and IO requests overlap, these stolen cycles serve to lengthen the codec's actual completion times. During a simulation run, this interference has to be recaptured.

We use the following process: As part of a CPU's profile, we measure the IO interference produced by any SCSI disk attached to it (usually just the internal drive). This interference factor is stored with the CPU abstract model, along with the tested device's transfer rate. Then, when the simulator builds a system model using another IO device, the CPU's interference factor is scaled by the new device's mean transfer rate.

Specifically, let $\beta_{CPU1,IO_{int}}$ be the percentage overhead realized by $CPU1$, when configured with its internal internal drive IO_{int}. We capture $\beta_{CPU1,IO_{int}}$ by running a series of decode tests on $CPU1$ without any IO active, and then re-running the same tests with IO_{int} active throughout every codec activity – which then yields the mean percentage difference between the two series of readings.

Then, to construct the interference model for $CPU1$ configured with some new device IO_{new}, we scale $\beta_{CPU1,IO_{int}}$ as follows:

$$\beta_{CPU1,IO_{new}} = \beta_{CPU1,IO_{int}} \times \frac{\mathrm{E}[Rate_{IO_{new}}]}{\mathrm{E}[Rate_{IO_{int}}]}$$

where $\mathrm{E}[Rate_{IO_{new}}]$ and $\mathrm{E}[Rate_{IO_{int}}]$ are the mean transfer rates on the new device and the old (internal) device, respectively. While this is a fairly coarse abstraction, it produces a sufficient model of interference, and one which proved fairly accurate in our on-line tests. The method works for one basic reason: a CPU (equipped with a set of given drivers) executes its SCSI interactions in a fairly consistent way, regardless of the device connected. Since a constant amount of interference is realized whenever a block is transfered, the percentage overhead is roughly proportional to the data transfer rate into the system. (A fast SCSI device, especially with the same blocking factor, will produce more interference than a slower device.)

As an example, on the PowerMac 7100/66, we measured $\beta_{66,IO_{int}}$ to be 0.057, in a configuration with a local transfer rate of 2530 Kbytes/sec. When we simulated the the 7100/66 with an external drive – which had a transfer rate of 3700 Kbytes/sec – we used $\beta_{66,IO_{ext}} = 0.057 \times (3700/2530)$, or 0.083. When we actually connected the external disk, this interference factor proved roughly correct within 15% accuracy, which sufficed for our purposes.

Now, when the simulator is actually run, there are two interference scenarios which have to be treated separately. These two cases are depicted in Figure 5, where S_{io} and F_{io} denote the start and finish times of the current IO transfer, and where S_d and F_d denote that start and finish times of the simulated decode, respectively. In the first case, decode-time is completely overlapped by IO, while in the second case, decode activity is only partially overlapped by IO – however it is interrupted by the IO callback, which also has to be taken into account. To adjust decompression time for these two scenarios, first T_{decode} is computed from T_{key} or T_{intr}, according to the frame type, and then scaled up to T'_{decode} using the current $\beta_{CPU,IO}$ factor:

Case (a):

$$T'_{decode} = \frac{1}{1 - \beta_{CPU,IO}} \times T_{decode}$$

Case (b):

$$T'_{decode} = \beta_{CPU,IO} \times (F_{io} - S_d) + T_{decode} + T_{IOc}$$

When IO and decompression do not overlap, $T'_{decode} = T_{decode}$.

In theory, while interference applies to all other CPU activities in the player, we note that in practice it has negligible affect. This is due to a simple fact: the durations of the other CPU activities are about an order of magnitude shorter than decoding, and the overhead percentage is relatively small itself. Hence, we have sacrificed a bit of accuracy for the sake of quick simulation results.

5 Simulation Engine and Test Results

The Engine. Table 5 overviews the simulator's internal transitions, and the queuing protocols used for each event. As its initial configuration, the event queue contains two elements: `soundISR` as its first member, and `schedule` as its second. From this point on the events take over, and handle the simulation run until the movie's last frame and sound sample are processed. Dispatching is handled in FIFO order, except in the case of (1) `SNDcallabck`, which re-schedules itself to asynchronously fire again after the current sound chunk is finished; and (2) `IOcallback`, which is scheduled by IO thread to fire when a simulated IO transfer is complete.

Results. In this section we compare some results of the simulator's runs with corresponding performance on our playback software. For these tests we used three Macintosh CPUs, (1) a PowerMac 7100/66, (2) a PowerMac 7100/80 and (3) a PowerMac 7500/100. The external IO devices were a Seagate ST12400 (with read transfer rates of 2800Kbytes/sec) and a Quantum XP34300 drive (with higher transfer rates of 3700Kbytes/sec) (see Table 4).

Event	Dispatch Time	Duration	Events Spawned
`predict`	FIFO	T_{pre}	`switch, schedule`
`display`	FIFO	T_{ply}	`switch, schedule, decode`
`IOcontrol`	FIFO	T_{IOc}	`switch, schedule, IOcallback`
`schedule`	FIFO	T_{sch}	`switch, display, predict, IOcontrol`
`switch`	FIFO	T_{cnts}	-
`IOcallback`	$T_{spawn} + T_{io}$	T_{io-cb}	-
`SNDcallback`	$T_{spawn} + T_{snd}$	T_{snd-cb}	`SNDcallback`
`decode`	FIFO	T'_{decode}	-

Table 3: Relating the time variables to simulator events. The variables denote the duration or relative dispatch time of an event.

For our test videos, we digitized and compressed two scenes from the popular movie "Pulp Fiction." Both clips are approximately one minute long, and both were digitized from clean tape, using a Radius VideoVision M-JPEG board at 30fps, with minimal signal loss. Then the clips were re-compressed into the Cinepak codec, at full 24-bit 640x480, again at 30fps. The keyframe distributions were varied from 1 per 30 frames, up to 1 every frame (i.e., all keyframes). Here we give the results for 1 and 10 keyframes per frame, respectively. A synopsis for the clips is given in Table 5.

In Table 6 we show a combinatorial set of results, ranging over the four videos, the two disk drives, and the three workstation models. Before the simulation phase we built the abstract device models as described above, and we profiled each component in isolation with its own local SCSI disk. (We note that the profiler's test videos do *not* include the Pulp fiction clips; they are much shorter, and range over different codecs and frames sizes.) Then the simulator assembled the six different systems from its repository file – scaling the respective $\beta_{CPU,IO}$'s to suit the two external devices. Each simulation took approximately one second to run on a Sun SPARCstaion 5, and they used the header files from the test clips as as their input.

Note that the differences between the simulator's prediction of playout performance, and the actual monitored performance, range between 0% to 7% – and the predictions may err either on the optimistic or pessimistic side.

On the PowerMac 7500/100, the percentage differences in playout rate are almost zero (and exactly zero in 3 instances). This is due to the fact that the 7500/100 is capable of displaying all video clips at $\mathcal{R}_{\text{MOVIE}}$; likewise the simulator achieves the same rates. On the 7100/80 and 7100/66, the simulator reflects the lower achievable frame rates. The greatest disparity is realized on the 7100/66 with the Quantum disk, during playback of Intercom/1.

CPU Platform	Processor	Bus	$\beta_{CPU,IO}$	Local Disk Transfer Rate
7100/66	PPC 601 - 66MHz	NuBus	0.057	2530 Kbyes/sec
7100/80	PPC 601 - 80MHz	NuBus	0.053	2600 Kbytes/sec
7500/100	PPC 601 - 100MHz	PCI	0.031	3100 Kbytes/sec

Table 4: System characteristics of CPU platforms.

Name	Key Frame Distribution	Total Number of Frames	Size (Mbytes)	Length (secs)
Jack Rabbit Slim's/1	1	1950	145.7	65
Jack Rabbit Slim's/10	10	1950	91.1	65
Intercom/1	1	1920	141.1	64
Intercom/10	10	1920	94.8	64

Table 5: Video characteristics of test movies. All movies are full-frame, cinepak videos digitized at 30fps.

CPU	Video	IO Devices			
		Seagate		Quantum	
		Simulator	Player	Simulator	Player
7100/66	Jack Rabbit Slim's/1	17.77	17.18	19.08	18.58
	Jack Rabbit Slim's/10	22.66	23.68	25.17	24.69
	Intercom/1	17.38	16.39	16.64	17.98
	Intercom/10	21.38	22.12	22.69	21.86
7100/80	Jack Rabbit Slim's/1	21.98	21.92	21.85	21.97
	Jack Rabbit Slim's/10	27.83	28.49	28.12	28.75
	Intercom/1	22.02	21.88	21.75	21.97
	Intercom/10	25.33	25.97	25.75	25.97
7500/100	Jack Rabbit Slim's/1	29.98	29.98	29.98	30.00
	Jack Rabbit Slim's/10	29.95	30.00	29.97	29.98
	Intercom/1	29.98	29.98	29.98	29.98
	Intercom/10	29.89	30.00	29.92	30.00

Table 6: $\mathcal{R}(t)$ of the video clips, generated from the player and the simulator.

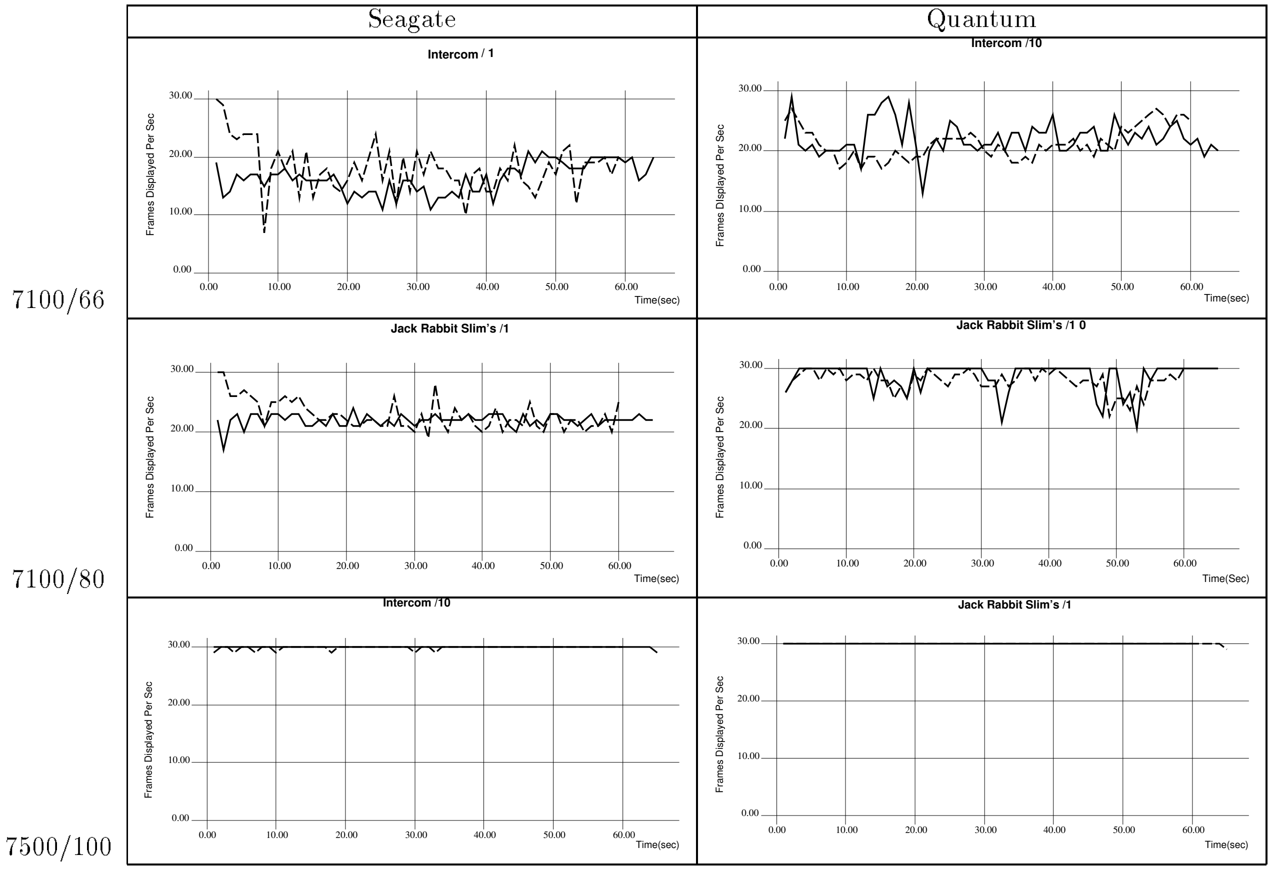

Figure 6: Comparison of the player and the simulator. Dashed lines denote the simulator and solid lines denote the player.

While the simulator is quite good at capturing average rates – over the course of an entire clip – it is less accurate on a per second basis. Figure (6) displays second-by-second playout performance for a subset of the experiments. Note that on the 7100/80 and 7100/66 (which are forced to drop frames in both simulation and true playback), there seem to be large variations in several trials. But this is to be expected. While we have modeled many of the time variables as pure Bernoulli decisions, they do, after all, have some deterministic causes. And in fact, the playout software can never even repeat its own second-by-second pattern, when when tested multiple times with the same clip. The differences lie in the system's inherent nondeterminism, which includes (1) varying behavior of the SCSI handlers, (2) differences in the callback firing times, (3) the pattern of IO and decode pipelining, as well as many other factors.

As a final fidelity test, we fed simulator's display-time lists to a previewer, and used the 7500/100 as a "viewing platform" to "watch" the timelines generated by the simulated 7100/66 and 7100/80. Esthetically, actual playback and simulated playback were visually equivalent, and that the disparity displayed in our second-by-second graphs was not visually discernible.

6 Conclusion

In this paper we described our prototype simulation package, which allows developers to quickly estimate performance of video clips running on different target platforms. The advantages of such a system are fairly obvious: it gives a reasonably accurate representation of video playout traces, which can then be used for extracting performance statistics, and for "viewing" the target's simulated behavior on a (superior) video production system. As long as the target models are in the simulator's repository, one can forgo actually buying the actual system, or even any of its component devices. This scheme, of course, relies on someone having profiled the components, but this need only be done once, and the component models can then be shared.

Note that there is a fairly simple reason why we can achieve a high level of accuracy: while many of the time variables have stochastic parameters, the simulator's main input – i.e., the movie's header itself – is completely determined in advance. Since the header contains the essential characteristics of every frame, these can be used as deterministic inputs for the time generating functions. In addition, we also note that video playout itself is significantly more deterministic than almost any other type of computer workload.

We are extending this work in several directions. First, we are expanding the device models to include various multi-spin CD-ROMs, which are now capable of delivering full-frame video at reasonable rates. Second, we are expanding the codec models to include higher resolutions, as well as different codec types (including software-only MPEG). Finally, we plan to extend our method to include hardware-codecs as well; we believe that the same profiling and simulation methods can be applied in a fairly straightforward manner.

References

[1] Navin Chaddha, Gerard A.Wall, and Brian Schmidt. An End to End Software Only Scalable Video Delivery System. In *Proceedings of the Workshop on Network and Operating Systems for Digital Audio and Video (NOSSDAV 95)*, 1995.

[2] Ming-Syan Chen and Dilip D.Kandlur. Downloading and Stream Conversion: Supporting Interactive Playout of Videos in a Client Station. In *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 73–80, 1995.

[3] Apple Computer Corporation. *Inside Machintosh: Quicktime*. Addison Wesley, 1994.

[4] Kevin Fall and Joseph Pasquale. Improving Continuous-Media Playback Performance with In-Kernel Data Paths. In *Proceedings of the First International IEEE Conference on Multimedia Computing and Systems*, pages 100–109, 1994.

[5] D.James Gemmell, Harrick M.Vin, Dilip D.Kandlur, P.Venkat Rangan, and Lawrence A.Rowe. Multimedia Storage Servers: A tutorial. *IEEE Computer*, pages 40–49, May 1995.

[6] Richard Gerber and Ladan Gharai. Experiments with Digital Video Playout. In *ACM Sigmetrics*, pages 210–221, 1996.

[7] Howard P. Katseff and Bethany S.Robinson. Predictive Prefetch in the Nemesis Multimedia Information Service. In *ACM Multimedia Proceedings*, pages 201–209, 1993.

[8] A.L. Narasimha Reddy and James C.Wyllie. IO Issues in a Multimedia System. *IEEE Computer*, pages 69–74, March 1994.

[9] Ralf Steinmetz. Compression Techniques in Multimedia Systems. Technical Report 43.9307, IBM European Networking Center, Vangerowstrabe 18, 69020 Heidelberg, Germany, 1993.

[10] Donald L. Stone and Kevin Jeffay. An Empirical Study of Delay Jitter Management Policies. *Multimedia Systems*, 2(6):267–279, 1995.

[11] Roy H. Campbell Zhigang Chen, See-Mong Tan and Yongcheng Li. Real Time Video and Audio in the World Wide Web. In *Proceedings of the Fourth International World Wide Web Conference*, 1995.