Priority-Progress CPU Adaptation for Elastic Real-Time Applications

Charles Krasic[a] and Anirban Sinha[a] and Lowell Kirsh[b]

[a]University of British Columbia, Vancouver, Canada

[b]Amazon.com, Seattle, WA

ABSTRACT

As multimedia-capable, network-enabled devices become ever more abundant, device heterogeneity and resource sharing dynamics remain difficult challenges in networked continuous media applications. These challenges often cause the applications to exhibit very brittle real-time performance. Due to heterogeneity, minor variations in encoding can mean a continuous media item performs well on some devices but very poorly on others. Resource sharing can mean that content can work for some of the time, but real-time delivery is frequently interrupted due to competition for resources. Quality-adaptive approaches seek to preserve real-time performance, by evaluating and executing trade-offs between the quality of application results and the resources required and available to produce them. Since the approach requires the applications to adapt the results they produce, we refer to them as *elastic real-time* applications. In this paper, we use video as a specific example of an elastic real-time application. We describe a general strategy for CPU adaptation called *Priority-Progress* adaptation, which compliments and extends previous work on adaptation for network bandwidth. The basic idea of Priority-Progress is to utilize priority and timestamp attributes of the media to reorder execution steps, so that low priority work can be skipped in the event that the CPU is too constrained to otherwise maintain real-time progress. We have implemented this approach in a prototype video application. We will present benchmark results that demonstrate the advantages of Priority-Progress adaptation in comparison to techniques employed in current popular video players. These advantages include better timeliness as CPU utilization approaches saturation, and more user-centric control over quality-adapation (for example to boost the video quality of selected video in a multi-video scenario). Although we focus on video in this paper, we believe that the Priority-Progress technique is applicable to other multimedia and other real-time applications, and can similarly help them address the challenges of device heterogenity and dynamic resource sharing.

Keywords: Quality-Adaptation, Continuous Media, Performance Measurement, CPU Scheduling

1. INTRODUCTION

As multimedia-capable, network-enabled devices become ever more abundant, device diversity (heterogeneity) and dynamics due to resource sharing remain difficult challenges networked continuous media applications. These challenges often cause the applications to exhibit very brittle real-time performance. Due to heterogeneity, minor variations in encoding can mean that a continuous media item performs well on some devices but very poorly on others. Resource sharing can mean that content can work for some of the time, but real-time delivery is frequently interrupted due to competition for resources. Quality-adaptive approaches seek to preserve real-time performance, by evaluating and executing trade-offs between the quality of application results and the resources required and available to produce them. Since the approach requires the applications to adapt the results they produce, we refer to them as *elastic real-time* applications.

The elastic real-time example we use throughout this paper is based on CPU adaptation for (distributed) video playback. Any of the basic resources of processing, network, and storage could potentially be the limiting factor on real-time performance of video. Our exploration of CPU adaptation is motivated by the general principal that the only way to make video software truly general purpose (hence viable for today's distributed systems) is to incorporate support for graceful adaptation to any and all potential resource bottlenecks.

With the rapid improvements in processors over the years, the CPU requirements of video are now typically very much secondary to the main limit of available network bandwidth. A typical desktop PC, with CPU such as an 3.0GHz Intel P4, only requires as little as 5–20% of its available cycles to play fullscreen standard definition (SD) video (720x480@30fps). With these figures, it would be easy to dismiss the CPU adaptation as uncessary, but that would be premature. There are

Further author information: (Send correspondance to Charles Krasic): E-mail: krasic@cs.ubc.ca

other instances where the balance of resources and video requirements shift, and the CPU time remains scarce. To illustrate the motivation for CPU adaptation, we consider three examples: mobile video devices, high definition (HD) video, and rich multi-video applications.

The variety of video-capable mobile devices such as smart phones, PDAs, Internet tablets, iPods, *etc.* are growing, and these mobile devices have very modest CPU speeds because of the need to conserve power usage and maximise battery life. Indeed, the adaptation work presented here is part of a project to port our adaptive streaming software to such types of device. Our initial targets are the Series 60 (S60) smart phones[19] and the Nokia 770 Internet Tablet.[18] The S60 software platform consists of Symbian OS and Nokia developed middleware. S60 is a major mobile platform and the market leader of the worldwide smart phone market, over 20 million S60 units were sold in 2005. For S60 development, we are using various models of phones by Nokia. They have all ARM based CPUs with speeds in the range 110-200Mhz. Support for high speed wireless networking in these devices is improving, most smart phones have Bluetooth wireless networking support, and more recently some models have 802.11 wireless as well (e.g. some third generation S60 devices, among others). The 770 is a new class of device released by Nokia in late 2005, and uses the Maemo software platform[17] which is based on Linux. The initial version of the 770 has a TI OMAP CPU (ARM instruction set) with speed of 250Mhz. The Nokia 770 supports 802.11b/g networking. The high speed networking combined with the relatively high quality the 770 display (800x480 resolution) make it very compelling target for wireless video streaming. However, unlike most PCs, the CPU speed of these devices is much more of a limit on maximum video quality than network bandwidth (even considering those with smaller screen sizes). We believe that the diversity of devices in the mobile space on the whole will make the concept of mobile video very unwieldy for content providers without graceful adapation mechanisms.

Somewhat at the opposite end of the spectrum from mobile video, there are high end video applications which will require much more CPU cycles. Improvements in large display technology are helping drive demand for high definition (HD) video, yet only the fastest processors available can even begin to cope with upper profiles of HD (e.g. 1080i, 1080p). In our lab, tests with the same type of PC mentioned above, which can easily play SD video, but using HD video content instead, show that it can only play 1080p HD video (1920x1080@30fps) at about half the full frame rate (15fps), fully saturating the CPU. Finally, there are rich continuous media applications that utilize multiple simultaneous videos, such as multi-party video conferencing, remote video surveillance, video authoring, tele-immersion *etc*. Developing these applications in a general purpose fashion so that they maintain good timeliness in the face of heterogeneity and dynamic resource sharing remains an open problem.

In this paper, we describe a general strategy for CPU adaptation called *Priority-Progress* adaptation, which compliments and extends our previous work on adaptation for network bandwidth. The basic idea of Priority-Progress is to utilize priority and timestamp attributes of the media to reorder execution steps, so that low priority work can be skipped in the event that the CPU is too constrained to otherwise maintain real-time progress. We have implemented this approach in a prototype video application. We will present benchmark results that demonstrate the advantages of Priority-Progress adaptation in comparison to techniques employed in current popular video players. These advantages include better timeliness as CPU utilization approaches saturation. In the multi-video case, the method allows controlled adapation across the videos, for example to keep quality even or to deliver higher quality for the video(s) specified as having higher importance. Although we focus on video in this paper, we believe that the Priority-Progress technique is applicable to other multimedia and other real-time applications, and can similarly help them address the challenges of device heterogenity and dynamic resource sharing.

The organization of the rest of this paper is as follows. In Section 2, we will elaborate on the CPU requirements of video playback, and discuss adaptation in two existing popular players. Section 3 will describe video adaptation in more detail, and present the design of our Priority-Progress CPU adaptation approach. Section 4 will present results from experiments with our prototype, showing its improvement over existing approaches. Related work is discussed in Section 5. Section 6 contains our conclusions and outlines possible future work.

2. CPU ADAPATATION IN EXISTING VIDEO PLAYERS

As described in Section 1, we believe the challenges of heteregeneity and resource sharing will make it inevitable that there will be circumstances where a video application has to attempt to maintain temporal fidelity even though there may be periods when the CPU is saturated. In this section we review basic characteristics of CPU requirments of video, and present some measurements from existing popular video players.

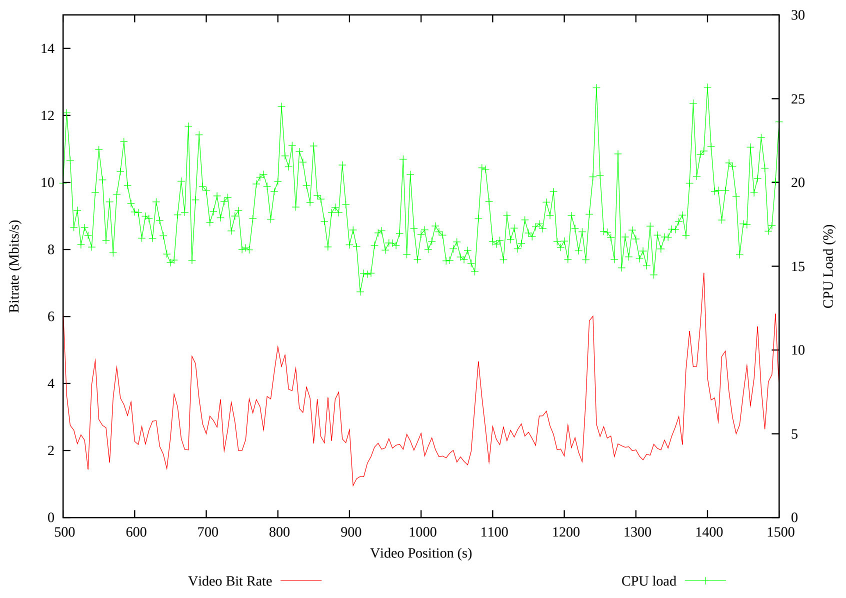


Figure 1. CPU utilization and bitrate measured while playing a single video. The values are smoothed over 5 second intervals.

2.1. On The CPU burstiness of Video

It is well established that the bitrate requirements of efficiently compressed video are highly variable over time.[7] Figure 1 shows the video bitrate and CPU usage while playing typical video, in this case an MPEG-4 conversion of a movie from a DVD. For readability, the values have been smoothed (to 5 second intervals). The main point we wish to illustrate with Figure 1 is that the CPU requirements are highly volatile over time, in strong correlation with bitrate. This suggests that a substantial amount of over-provisioning relative to the average will be necessary to avoid episodes of CPU saturation. It also suggests that if over-provisioning is not feasible, then effective CPU adaptation is a formidable challenge.

2.2. Benchmarks from Popular Video Players

To help provide context for understanding the performance our CPU adaptation approach, we first perform some measurements on two of the most popular open source video players: *MPlayer*[8] and the VideoLan Client (*VLC*).[1] We chose these programs partly because they are open source*, which made it easy for us to add instrumentation and take measurements. We also chose them because they are of very high overall quality, the result of very large teams of talented and dedicated developers. Due to the free nature of these programs it is difficult to get a precise idea of how widely they are used, but there are good indications that their user communities are substantial. For instance at the time of writing, the download page for VLC reports over 10 million downloads, and the mailing lists for each of the projects receive hundreds and even thousands of postings per month. Between the two, we think it likely that they rival many commercial video players in overall popularity.

One of the main reasons we chose to evaluate both MPlayer and VLC is because they have substantially different software architectures, representing two comparably predominant schools of thought with regard to real-time application architecture: namely event-driven vs multi-threaded. Given their popularity and maturity, we suggest that these two programs are representative of the best of breed for the respective architecture. Although the relative merits of event-driven and multi-threaded architectures remain contentious between advocates of either,[13,20,25] it is safe to say that the event-driven side generally claims better performance, while the multi-threaded side claims better ease of programming. Reflecting somewhat it's origins in eastern Europe (where fast computers are somewhat more scarce), and it's slightly older age, the MPlayer project has always had a strong emphasis on overall speed, and the ability to achieve real-time performance on

*Both programs use the GPL license.

even the lowliest of hardware. MPlayer is single threaded, and is written mainly in the C language and also contains significant amounts of hand coded assembly. VLC originated with a team of teachers and students at Ecole Centrale Paris, and arguably reflects a more academic approach emphasising more code modularity and readability. VLC is written mainly in `C++` [†], and it uses a multi-threaded architecture.

To characterize the real-time performance of a video application, there are many possible subjective and objective timeliness metrics. In our evaluation of MPlayer and VLC, we will focus on aspects of temporal fidelity, that is, how natural and smooth motion in the video appears. Audio-video synchronization is another important aspect of video timeliness, but we leave that outside the scope of this paper.

To quantify temporal fidelity, we have instrumented the two players to produce trace output for each frame displayed. The trace indicates for each frame the time it was displayed. Then for each frame, we compute the time since the previous frame was displayed, which we call the frame jitter. We also calculate the frame rate average over time. We use these two metrics to quantify fidelity. The most popular frame rates for video are between 24 and 30 frames per second, which represent nominal frame jitter values of 41.7 and 33.3 ms respectively. Although the frame jitter and frame rate are derived from the same value, they convey different aspects of temporal fidelity. The frame rate is better at expressing the general notion of motion smoothness. The frame jitter is better at indicating worst cases, notably user-perceivable pauses, which the averaging in the frame rate value may conceal.

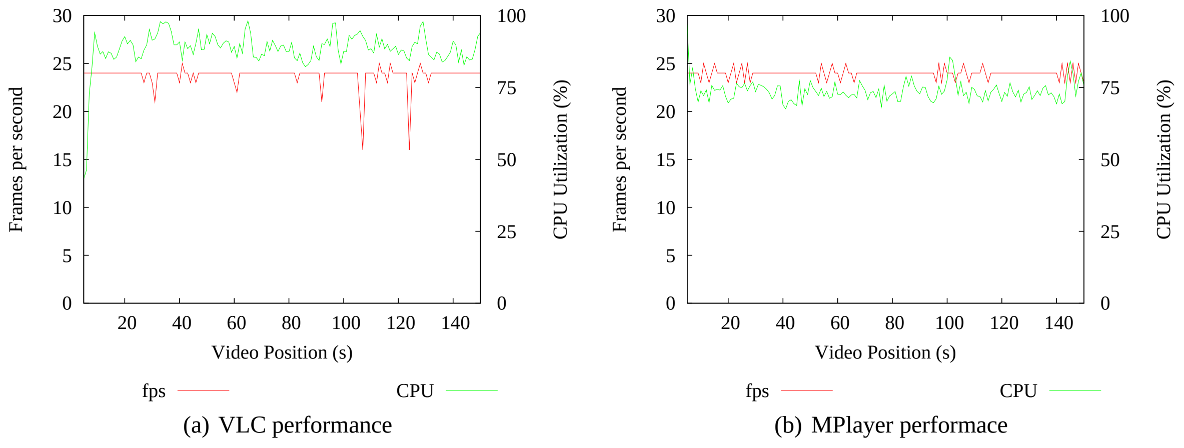
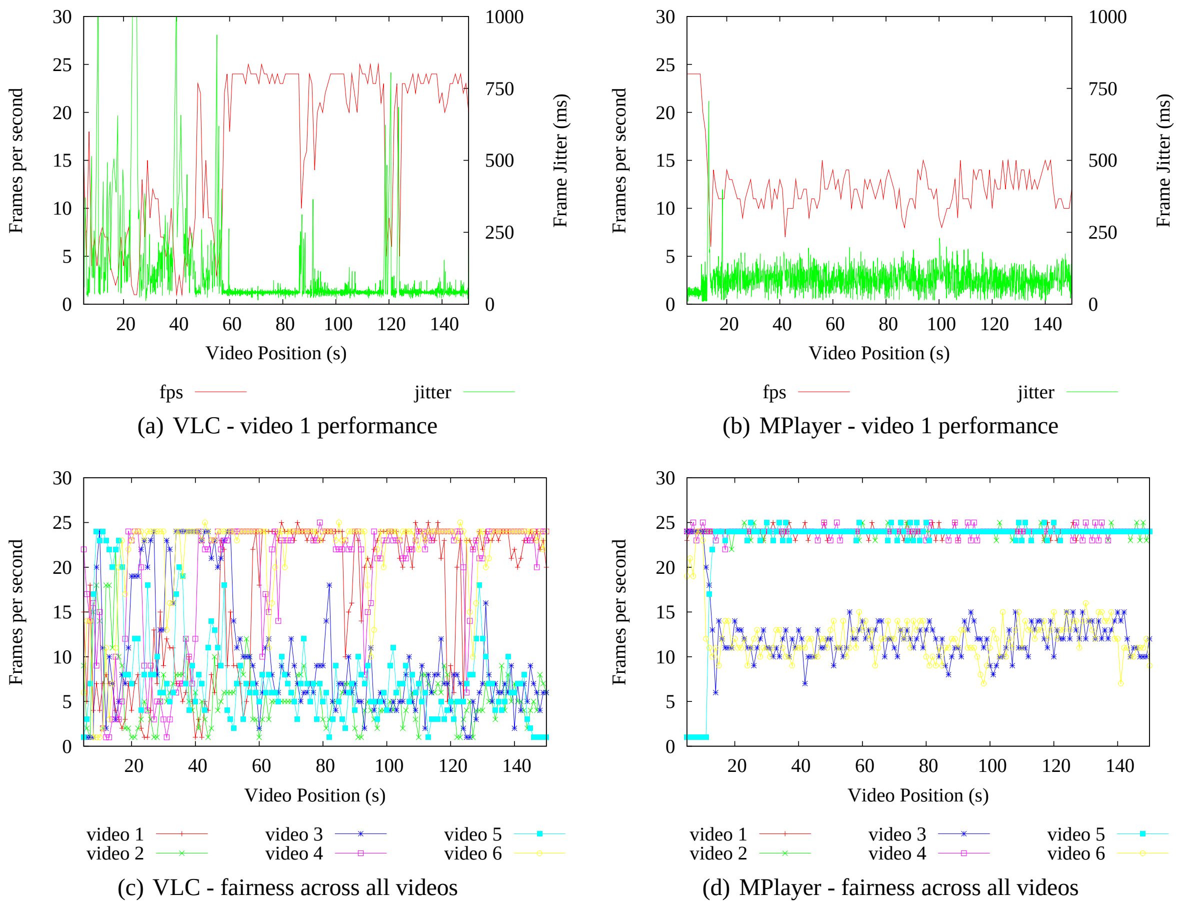


Figure 2. Playing 3 videos. Left y-axis shows frame-rate, right y-axis CPU usage. Both players maintain full FPS (24) most of the time, and are free of major pauses.

As far as CPU quality-adaptation goes, frame rate adaptation is the only mechanism supported by MPlayer and VLC (and most other players). We will describe some of the complexities of frame rate adaptation with compressed video in Section 3. For the moment, we turn our attention to the performance of MPlayer and VLC under increasing CPU load. The experiments in this section were run on a Dell Inspiron 1300 laptop PC, with a 1.8 GHz Pentium-M CPU and 512 MB of ram. The operating system is Ubuntu Linux 6.06. The video is a movie taken from a DVD and converted to MPEG-4 format. To vary the CPU load, we ran a sequence of experiments, each plays n simultaneous videos (the same video), for increasing values of n . We did this for both VLC and MPlayer. From the set of experiments, we selected two values of n of interest, which we denote n_{sat} and $n_{2\times sat}$, where n_{sat} is the largest value of n where CPU usage remains just below full utilization, and $n_{2\times sat}$ is double that value, which we chose to force the CPU to be heavily saturated.

In our setup, n_{sat} was 3 for both players. Figure 2 shows the frame rate and CPU utilization for the first of the three videos, Figure 2(a) for VLC and Figure 2(b) for MPlayer. Subjectively, in both cases the videos play with normal smoothness, and no noticable pauses (we omit plotting frame jitter because the values were all close to the nominal value). Although, the same video was used, and MPlayer and VLC use the same MPEG-4 codec, CPU utilization for MPlayer was about 14 % less than with VLC (74% vs 88%).

[†]It still uses substantial amount of library code such as the FFMpeg codecs[3] that are written in C and hand generated assembly.



(a) VLC - video 1 performance (b) MPlayer - video 1 performance (c) VLC - fairness across all videos (d) MPlayer - fairness across all videos

Figure 3. VLC and MPlayer performance. Forcing saturation with 6 simultaneous videos. The top graphs show peformance of one video selected from the group. The bottom graphs show frame rates of all videos.

When we double the number of videos, forcing CPU saturation, the frame-rate adaptation mechanism of each player becomes necessary. In this case we are interested in the timeliness of individual players, as well as fairness across the different videos. Figure 3(a) shows the performance of VLC for one of the videos. This particular video had several pauses in the first 60 seconds. Also, frame rate is very poor in the first 60 seconds, although later the stream recovers. Figure 3(c) shows the frame rate of all six videos for VLC . We note that fairness is very bi-modal, with most of the videos experiencing very low frame rates, and a few have almost full frame rate. Figure 3(b) shows one of the six videos when the plaeyer is MPlayer. Unlike VLC, MPlayer's adaptation seems to drop the frame rate more consistently. Notice also that MPlayer is more free of large frame jitter spikes. Figure 3(d) shows that across all six videos, MPlayer also exhibits bi-model fairness, albeit it seems more consistant than VLC.

One conclusion we draw from these results is that MPlayer has an edge in performance and adaptation over VLC. Given that they use the same MPEG-4 codec,[3] we attribute the bulk of this to the architectural differences between the two programs. We think it is more difficult to control timing and provide graceful degradation with the multi-threaded approach. Our prototype player which we describe in the next section also takes the event-driven approach. When we consider fairness, neither MPlayer or VLC seems to adapt satisfactorily though. In part, we attribute this to the fact as separate programs, the kernel scheduler has significant influence. In our prototype, we will take the purported timeliness advantages of the event-driven approach to a next step.

3. PRIORITY-PROGRESS CPU ADAPTATION

As desribed in Section 1, the goal of our work is to help make real-time applications more elastic, meaning that they more gracefully adapt to the effects of heterogeneity and resource sharing. In previous work,[11] we introduced our the Priority Progress Streaming (PPS) algorithm and its implementation in QStream, our prototype video streaming software.

3.1. Priority-Progress Review

We call our CPU adaptation Priority-Progress also because it builds on and compliments the same approach we used to adapt to available network bandwidth in our PPS streaming algorithm. We provide a brief overview of Priority-Progress as used for network adaptation here, more details are available in.[10, 11] Priority-Progress is based on three principles: incremental quality, prioritized data, and priority data dropping.

The incremental quality assumption means that the target application can be architected and designed so that its main results can be produced in an incremental fashion, where successive increments improve the quality of results. For video, we used the idea of layered compression to achieve this incremental property. Our compression format is called SPEG, which we derived from MPEG by adding a form of fine grained SNR scalability. Initially, SPEG was based on MPEG-1 and had four SNR layers per frame.[11] The current version of SPEG, used in this paper, is based on MPEG-4 and has eight SNR layers per frame. We also implemented temporal (frame rate) scalability with SPEG.

The second principle of Priority-Progress is prioritized data. In QStream, the process of prioritizing data is performed by a component we call the *mapper*. The purpose of the mapper is two-fold. The first has to do with the fact that we expect quality-adaptation to be multi-dimensional, meaning that there is more than one way to adapt the quality-resource trade-off, and there needs to be a way to orchestrate the mix of adaptations. In QStream, the two adaptations are temporal quality (frame rate) and spatial quality (SNR layers). The mapper algorithm is parameterized by policy specifications, in the form of user-controllable utility functions, that are utilized to guide the priorities the mappers assigns to units of layered media. The other purpose of the mapper is modularity, we wanted to minimize the amount of code that was exposed to low-level details of video. The networking part of PPS only use timestamps and priorities (provided by the mapper), and does not need to understand SPEG bitstream format.

The final principle in Priority-Progress priority-order execution. We use timestamps and priorities to adapt quality while maintaining basic timeliness. The timestamps are used to subdivide the data into a series of intervals, we call adaptation windows. Within a given adaptation window, data are re-ordered by priority, and processing proceeds from high to low priority. The timestamps define a natural deadline for the adaptation window, if the processing does not complete by that deadline, the low-priority data are dropped. In this way, the application continuously adapts to available resource, while maintaining timeliness. This is the basis for network adaptive streaming in QStream, which is very effective.[11] In this paper, our intent was to re-use most of the concepts toward CPU adaptation, the rest of this section will describe the differences between the network and CPU case, and adjustments we made.

3.2. Priority-Order Decoding

In our approach to CPU adaptation, it is necessary to understand the division of time among the various computation steps involved in the overall real-time process. Part of the reason we believe the Priority-Progress idea can be applied to CPU adaptation is the well known observation that most applications spend their time in a small fraction of their code, the so-called critical paths. If the critical path is well defined enough, then we surmise that we need only (re-)design the critical path to be CPU adaptive to make the overall application elastic. The first step therefore, is to understand the performance and identify the critical paths.

In video software, the processing requirements are highly concentrated in the activities surrounding video compression. Profiling of the MPlayer and VLC, and our QStream video player (using the `oprofile` tool for Linux[14]) shows that they usually spend from 80–95% of their time in the video codec (i.e. video decompression algorithm). Aside from the codec, all other processing activities tend to be orders of magnitude less demanding, although this can be somewhat hardware dependent.

Therefore, we have chosen to incorporate Priority-Progress style dropping into the video decoding process. Similar to the network case, we group video data into time intervals, called adaptation windows, and then process (decode) the data in priority order, dropping low priority data if we run out of time. Initially, our intention was to re-use the priority assignment produced for streaming (PPS) directly, but instead we found it necessary to modify the priorities, for reasons which we now elaborate.

3.2.1. Data Dependencies and Frame Ordering in Compressed Video

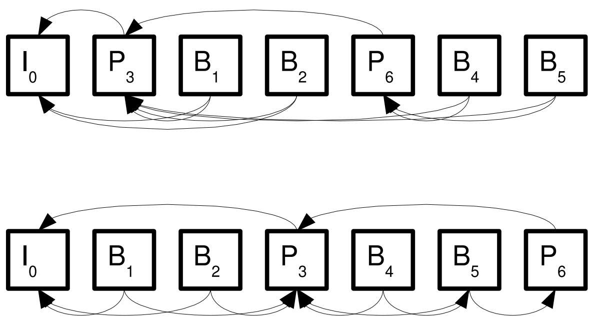


Figure 4. An example of decode versus display order in MPEG. Frames are decoded in the order of upper sequence (left to right), and re-ordered for display as in the lower sequence.

Our goal is to adapt by decoding frames in QStream's priority order, which differs from the normal order of decoding. However, a standard video decoder does not support re-ordering of video data as easily as network code supports re-ordering of messages. Due to predictive coding (I, P, B frames) in MPEG, the decoder has its own re-ordering logic that expects input frames in so-called decode order, and re-orders frames internally to produce output in display order. Figure 4 depicts the concepts of decode and display order used by MPEG. Furthermore, many decoder APIs including the one we use (based on the XViD codec[12]), expect their input to be provided in whole frames. Even though the XViD code is of a very high standard, we would characterize it as being something like a formula one race car: it is extremely intricately tuned and even minor changes are in a lot of danger of disrupting its stability. Diagnosing and correcting mistakes is often maddeningly difficult and time consuming. Therefore, although some changes to the video code were inevitable to realize priority-order decoding, our intent was to start by keeping the changes as minimal as we could.

In light of our desire to minimize tampering with video code, we decided that supporting priority-order decoding of SPEG spatial (SNR) layers would be too ambitious. In the network case, the PPS priority ordering often means that the base layer (highest priority) data from a whole group of frames are sent before subsequent spatial layers. For priority-order decoding, this would require that we abandon the whole frame model of the XViD codec API. Instead we would need to be able decode part of a one frame, but defer decoding spatial enhancements until after working on other frames. This is particularly hard in the SPEG4 decoder, because within a frame, all of the spatial layers are processed on a macroblock at a time basis. The macroblock at a time logic means there is only one macroblock object, whereas layered decoding would require a whole frame's worth of macroblock objects to store incremental results. Furthermore, we would have significant 'end of frame' processing (iDCT, dequant, etc.), and it isn't entirely obvious when it would be best to execute. Therefore, we chose to restrict our Priority-Progress adaptation to units of whole frames, and leave spatially adaptive decoding to future work.

To restore the whole frame model, we added a new step to the QStream client that aggregates SPEG data units that belong to the same frame (but different spatial layers). In PPS, These data units do not generally share the same priority value. When combined, we compute an overall frame priority, by taking the highest priority of the constituent parts. This preserves an important property of the priorities generated by the mapper, which is that frame dropping is spaced as evenly as the GOP pattern (mix of I, P, B frames) will allow. In other words, the priorities are assigned so frames are dropped in a pattern that will minimize frame jitter.

Even with whole frames, we still require changes to the codec due to the fact that the frames are now supplied in priority-order as opposed to the standard MPEG decode order. In particular, this effects the treatment of references frames by the the motion compensation component of the decoder. The standard solution for an MPEG decoder is to retain copies of the most recent two reference frames (I or P frames) internally for the purpose of motion compensation. Processing of B-frames (bidirectionally predicted) will also mean that output of reference frames may be delayed by the decoder, so that output order will equal the display order of frames. Given the above mentioned delicacy of video code, we decided that the safest solution would be to move management of reference frames outside the video codec.

External reference frame management works by expanding the codec API to include the reference frames explicitly, and effectively disables the decoder's logic for re-ordering between decode and output order, so that every input frame reveals the output frame immediately. For example, when we supply a B-frame to the decoder, we explicitly include the two (previously decoded) reference frames along with it. The application (QStream) is responsible for keeping track of which reference frames are required for each frame supplied to the decoder, and for resorting output frames into display order. This scheme also requires the application to be careful about memory management, since it is imperative that the application not release the memory for a reference frame until all dependent frames are decoded. To ensure this, we use reference counting. In the end, our approach kept the bulk of code to support priority order decoding outside of the video codec. In all, the size of changes to the codec amounted to less than 50 lines of code.

3.3. CPU Adaptation Windows

To adapt to available CPU, the last step we need is the one that drops low priority frames. Again, we take the priority-progress approach which groups data (frames) into an adaptation window, and work from high to low priority. For decoding we call the windows Priority-Progress Decode (PPD) windows. Although they share the same basic purpose as PPS windows, PPD's differ from the PPS windows because of memory considerations. Just as the PPS receiver accumulates data while receiving, PPD accumulates output frames, and the final display of the output frames does not begin until the entire window is complete[‡]. However, in comparison to the compressed SPEG data, these output frames are very large, so the feasible duration of PPD windows is much more constrained than PPS windows. To put this in perspective, DVD quality video will require about 1-10 Mbit of memory per second of PPS window, whereas the raw output from that same video requires about 165 Mbits (approx. 15 Mbytes) per second of PPD window. In practice, the PPS streaming algorithm in QStream will allow windows to reach sizes on the order of 30 seconds or more, whereas for the CPU case, we test adaptation windows on the order of hundreds of milliseconds up to a maximum of 2 seconds.

3.4. Inter Stream Adaptation

We encountered one final issue with CPU adaptation, about fair adaptation in the presence of multiple concurrent streams. We chose to use playback of multiple simultaneous videos in many of our experiments, because it represents an extremely challenging scenario for elastic real-time applications. When we first attempted such tests, we found that the Priority-Progress mechanism was subject to quality oscillations across the streams. After some diagnostic effort, we found the problem was related to the fact that the adaptation windows of the different video windows were unsynchronized, meaning they did not start and end at the same times, and they did not all have the same duration. Due to the unsynchronization, short duration windows were susceptible to starvation if high priority frames of another window (from another video) had already started processing. To solve this issue, we added a secondary sort key to the adaptation algorithm. In addition to sorting frames by priority, we use secondary key of frame's display deadline to order frames that have the same priority. Subsequent to this modification, our QStream implementation was able to perform very well in the multiple video case.

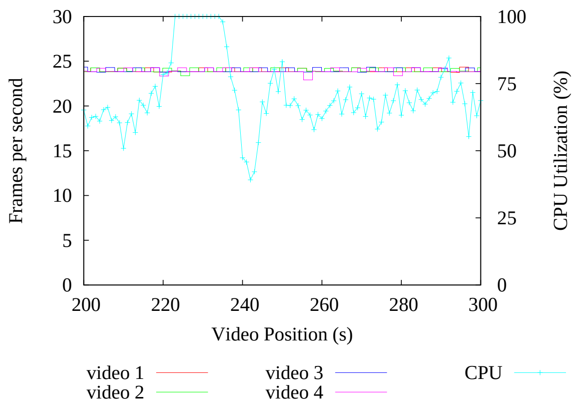
4. EXPERIMENTS WITH SPEG AND QSTREAM

In this section we present measurements of Priority-Progress adaptation using our QStream prototype. All of the experiments in this section play videos on a desktop PC (white box) with a 3.0 GHz Intel Pentium 4, and 1G RAM. For the QStream experiments, a QStream streaming server runs on a separate PC. Except where noted, values are smoothed on 1 second intervals in all plots in this section.

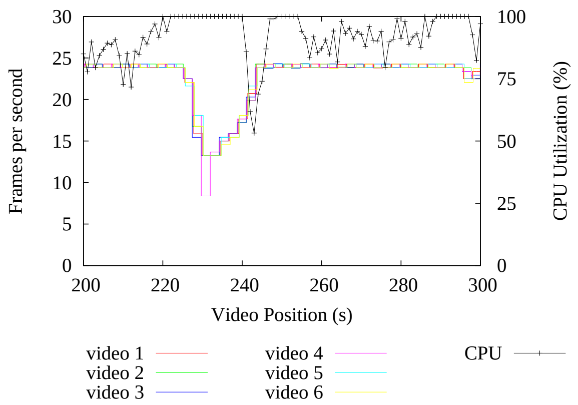
Figure 5 shows the QStream player's performance as CPU load is increased. During periods of CPU saturation, there are reductions in frame-rate. Noting the CPU usage in Figure 5(a), we can see that the periods of reduced frame rate in all of Figures 5(a)–5(d) are quite well correlated with peaks in CPU requirements of the video. In general the fairness across videos is extremely close.

Figure 6 shows the frame jitter for the first video of each each experiment, recall we define frame jitter as the time difference between each successive frame displayed, for the same runs. While the frame rate plots are calculated based on one second intervals, the plots in Figure 6 show every frame individually, so they provide a better understanding of worst-case. Subjectively, we find the frame jitter metric captures well the notion of stutter or pausing that one notices when video frame rate becomes unacceptably low. Figure 6 confirms that in these experiments, QStream avoided interruptions even as the CPU became heavily saturated.

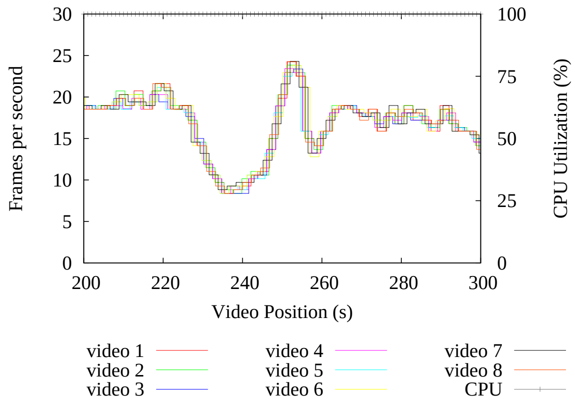
[‡]Consider that it is possible that the lowest priority frames in priority order could be among the earliest frames in display order.



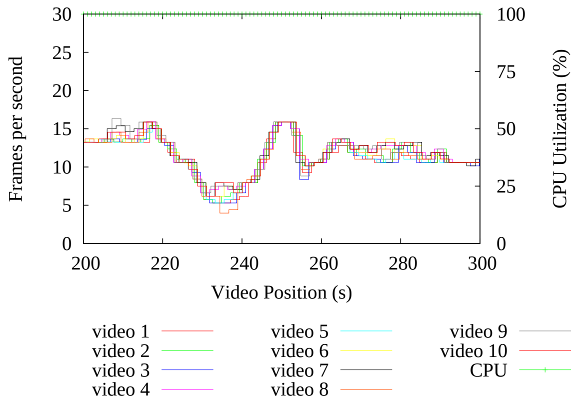
(a) QStream - 4 SPEG videos



(b) QStream - 6 SPEG videos



(c) QStream - 8 SPEG videos



(d) QStream - 10 SPEG videos

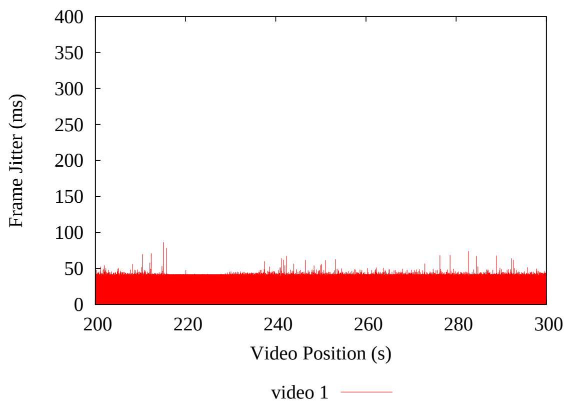
Figure 5. QStream performance with increasing load. Frame rate drops as load increases, but frame rate is kept even across all videos.

In the previous experiment, all videos were prioritized with the same policy. Recall from Section 3, QStream includes the *mapper* component that derives data priorities based on explicit adaptation policies, in the form of utility functions. The mapper is run online in the QStream prototype, and it includes interactive controls which allow these utility functions to be changed on a per-video basis. In the the next experiment, we select one of the videos to received preferred treatment by using a different utility function from the others, specifying that the utility of the preferred video across the range of frame rates is higher than for the other videos. This kind of adaptation might be useful in a multi-party video conference, where one video at any given time is considered the foreground (speaker). Similar scenarios might occur in other multi-video applications (surveillance, authoring, *etc.*). Figure 7 shows the results with minimal saturation, and with heavy saturation. In Figure 7(a), there is mild CPU saturation, but in the one period where frame rates drop, the preferred stream stays at full frame rate. In Figure 7(d), their CPU is always saturated, but the higher utility video clearly achieves proportionately higher quality than the others.

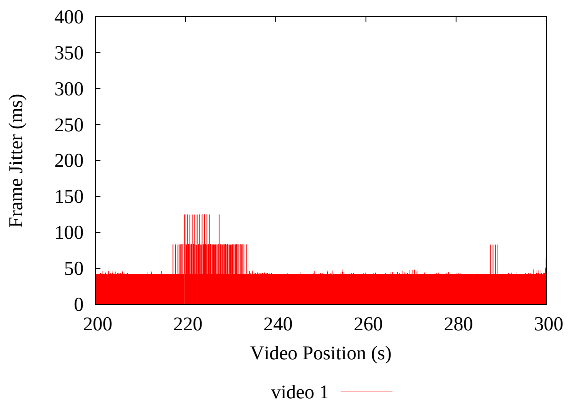
5. RELATED WORK

Quality-adaptation for multimedia is not a new concept. Many others have proposed techniques for video adaptation. This work traces back to much of the pioneering work on quality adaptive multimedia in the Quasar Project.[5,24] The Priority-Progress CPU adaptation we propose is based on the technique we proposed for adaptive video streaming,[11] which in turn was inspired by several other works on quality-adaptive streaming.[6,21,22]

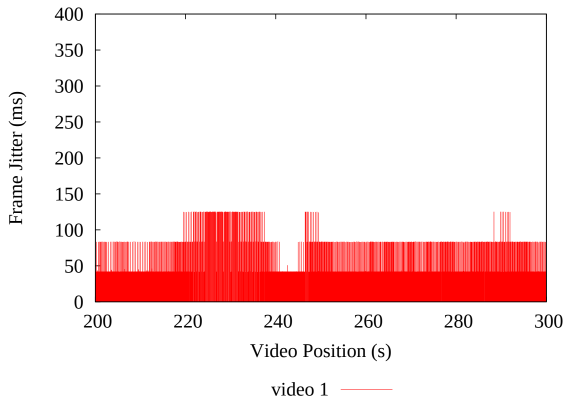
The classic formulation of quality-adaptation is as a feedback control loop, in which a controller is used to adjust trade-offs between application quality (video fidelity) and resource usage (computation time) in order to meet the timeliness



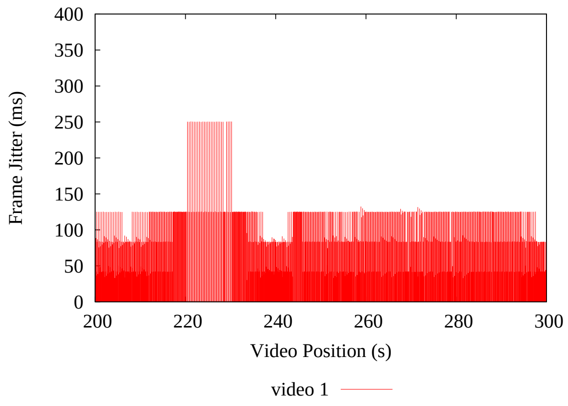
(a) QStream - 4 SPEG videos



(b) QStream - 6 SPEG videos



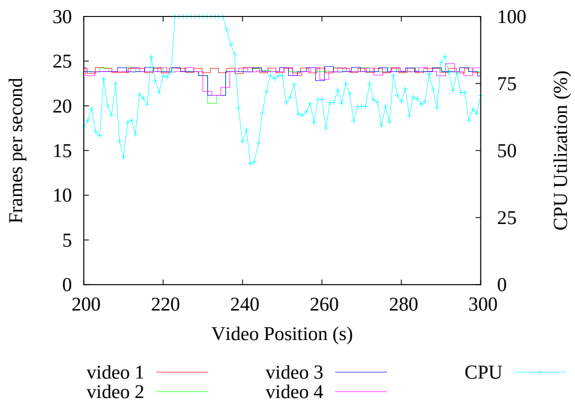
(c) QStream - 8 SPEG videos



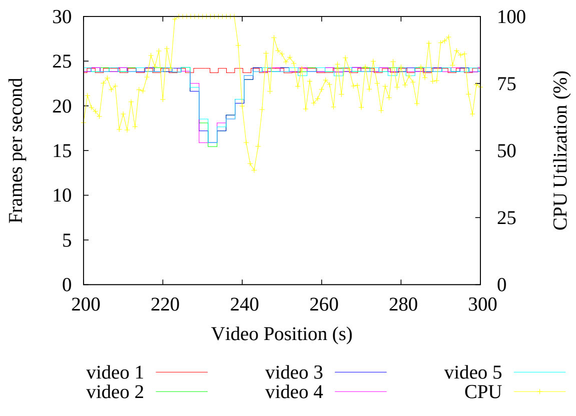
(d) QStream - 10 SPEG videos

Figure 6. Frame Jitter: at lower frame rates, QStream still avoids severe stutter or pauses.

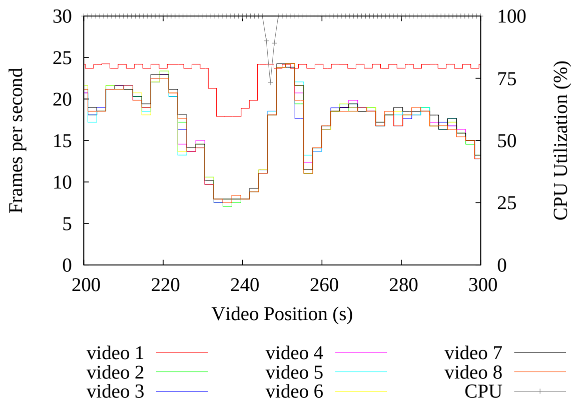
objectives inherent to the task.[4,9,15,16,23] This approach is based on the premise that the feedback loop can monitor some aspect(s) of progress in the recent past in order to make control decisions and actuate parameters of quality-adaptive mechanisms accordingly. The general concept of feedback control is formalized in control theory, which is used successfully in a wide range of electronics, and increasingly applied in software. However, we feel that the application of feedback loops in quality-adaptation is of dubious utility. Most classic quality-adaptation controls, whether implemented by ad-hoc logic, or by a more formal application of principles of control theory, assume (either implicitly or explicitly) that there is a reliable way to monitor progress and that there exists a model that can estimate the correct control decisions for the near future, based on monitored values. For CPU adaptive video, this means that the adaptation mechanism must be able to estimate the CPU time required to process video data, estimate the amount CPU time that will be available, and from these derive control decisions that will lead to the best final video quality. There have been many analytical studies concerning resource dynamics of video (bandwidth requirements, computational cost) which in large part motivated in understanding how to design reliable control mechanisms.[2,7] Our position is that the high complexity of proposed models for video and the difficulty of estimating resource availability in best-effort environments, undermines the effectiveness of feedback based quality-adaptation. One of the major motivations of the Priority-Progress approach is to take an alternative approach to the classic feedback loop, wherein estimates of resource requirements and availability are not needed to accomplish proper adaptation. For CPU adaptive video, this means that Priority-Progress does not need to know or predict how much CPU time is required to process each video image, nor does it need to predict how much CPU time will be made available to it by the CPU scheduler.



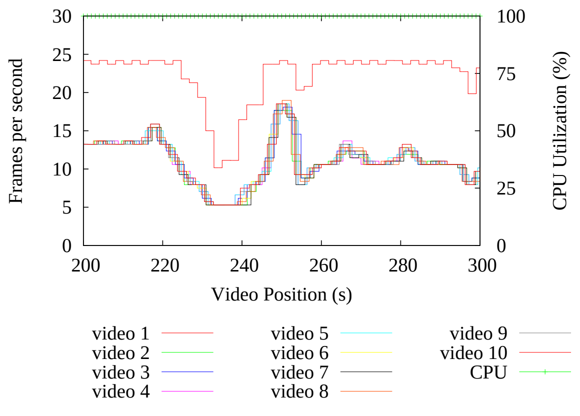
(a) QStream - 4 SPEG videos



(b) QStream - 5 SPEG videos



(c) QStream - 8 SPEG videos



(d) QStream - 10 SPEG videos

Figure 7. Adaptation control: Video 1 is given higher importance than the others.

6. CONCLUSIONS

We have presented the design and evaluation of the Priority-Progress CPU adaptation, an approach to quality-adaptive scheduling in elastic real-time applications. Our approach is quite novel in that it preserves timeliness in conditions of persistent CPU overload, which is generally an anathema to conventional real-time approaches. In addition, it allows control over sharing in quality-centric terms (frame rate), rather than resource centric terms (task priority, nice value, proportion, etc) that most scheduling mechanisms provide. In future work we plan to explore more fine-grained adaptation mechanisms (spatial adaptation). The implementation of this work is part of a fully functional streaming system, QStream, which is open source and is available at our website: `http://qstream.org/`.

7. ACKNOWLEDGMENTS

We are grateful to Ashvin Goel who provided valuable feedback on this work.

REFERENCES

1. L. Aimar et al. VideoLan. `http://www.videolan.org/`.
2. A. C. Bavier, A. B. Montz, and L. L. Peterson. Predicting mpeg execution times. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 131–140, New York, NY, USA, 1998. ACM Press.
3. F. Bellard, M. Niedermayer, et al. The FFMpeg Project. `http://ffmpeg.sf.net/`.

4. S. Cen. *A Software Feedback Toolkit and its Application In Adaptive Multimedia Systems*. PhD thesis, OGI, October 1997.
5. S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A Distributed Real-Time MPEG Video Audio Player. In *Network and Operating System Support for Digital Audio and Video*, pages 142–153, 1995.
6. N. Feamster, D. Bansal, and H. Balakrishnan. On the Interactions Between Layered Quality Adaptation and Congestion Control for Streaming Video. In *11th International Packet Video Workshop (PV2001)*, pages 128–139, Kyongiu, Korea, April 2001.
7. M. W. Garrett and W. Willinger. Analysis, modeling and generation of self-similar VBR video traffic. In *SIGCOMM*, pages 269–280, 1994.
8. A. Gereoffy et al. The FFMpeg Project. `http://www.mplayerhq.hu`.
9. A. Goel, J. Walpole, and M. Shor. Real-rate scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 434–441. IEEE Computer Society, 2004.
10. C. Krasic. *A Framework for Quality-Adaptive Media Streaming*. PhD thesis, OGI School of Science and Engineering at OHSU, 2003.
11. C. Krasic, J. Walpole, and W. chi Feng. Quality-adaptive media streaming by priority drop. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 112–121, New York, NY, USA, 2003. ACM Press.
12. C. Lampert, M. Militzer, P. Ross, et al. The XViD Project. `http://www.xvid.org/`.
13. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.
14. J. Levon, P. Elie, et al. OProfile. `http://oprofile.sourceforge.net/`.
15. B. Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, September 1999.
16. B. Li and K. Nahrstedt. "impact of control theory on qos adaptation in distributed middleware systems". In *In Proceedings of the 2001 American Control Conference (ACC 2001)*, Arlington, Virginia, 2001.
17. Nokia. Maemo Project. `http://www.maemo.org/`.
18. Nokia. Nokia 770 Internet Tablet. `http://www.nokia.com/770`.
19. e. a. Nokia, Symbian. Series 60 Smartphone Software Platform. `http://www.s60.com/`.
20. J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.
21. R. Rejaie, M. Handley, and D. Estrin. Quality Adaptation for Congestion Controlled Video Playback over the Internet. In *Proceedings of ACM SIGCOMM '99 Conference*, pages 189–200, Cambridge, MA, October 1999.
22. D. Sisalem and F. Emanuel. QoS Control using Adaptive Layered Data Transmission. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, Austin, Texas, June 1998.
23. J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, page 59, Washington, DC, USA, 2001. IEEE Computer Society.
24. J. Walpole, R. Koster, S. Cen, C. Cowan, D. Maier, and D. McNamee. A Player for Adaptive MPEG Video Streaming Over the Internet. In *Proceedings 26th Applied Imagery Patter Recognition Workship AIPR-97*, Washington, DC, October 1997. SPIE.
25. M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM Press.