Achieving Predictable Timing via Cooperative Polling

Charles Krasic
Department of Computer Science
University of British Columbia

Ashvin Goel
Electrical and Computer Engineering
University of Toronto

Anirban Sinha
Department of Computer Science
University of British Columbia

Abstract

Today, high-quality media streaming and rich-media applications have become available on desktop computers, and mobile devices are increasingly video enabled. Although computing power is improving over time, user expectations of video quality are also increasing, and thus the resource requirements of these applications is keeping pace with computing improvements. Hence, these applications must adapt gracefully based on resources such as available CPU and network bandwidth to maintain their timing constraints. Unfortunately, current operating systems provide little support for time-sensitive applications whose adaptation mechanism can interact poorly with the kernel's scheduling mechanism leading to poor performance.

In this work, we present an event-driven application model called cooperative polling that is aimed at supporting adaptive, time-sensitive applications. Cooperative polling reduces unpredictable timing by minimizing involuntary preemption, and it facilitates cooperation between applications by sharing event information such as deadlines and priorities across applications. This approach together with a simple deadline-based scheduling policy achieves overall predictable timing, and also enables application-centric adaptation during overload. Our evaluation with a streaming video player shows the significant benefits of this model.

1 Introduction

The distinction between general-purpose and real-time computing has become blurred as rich media applications such as audio, video, and animated computer graphics have become an integral part of everyday computing. These applications have become increasingly demanding as CPU and network capacity has improved over time. For example, desktop computers are being used for complex media applications such as picture-in-picture, video editing, remote surveillance, and multi-party conferencing. At the same time, improvements in video sources and sinks such as cameras and displays have kept pace with improvements in other computing resources so that data rates of high quality video often correlate with or surpass the limits of various resources such as CPU, local storage or wide-area networking. For instance, with high definition video, the compression algorithms in current video standards such as MPEG-4 and H.264 have computational demands that can saturate the fastest processors currently available.

Media applications are characterized by three requirements. They have timing constraints that must be satisfied for correct operation, the resource requirements of these *time-sensitive* applications can vary dramatically over time, and they must be able to *adapt* their behavior to avoid overloading the bottleneck resources. These adaptive, time-sensitive applications require *both* short-term responsiveness and long-term throughput. This combination of requirements is a challenge for general-purpose operating systems which use simple scheduling heuristics that are mainly optimized for throughput-oriented applications. For example, a scheduler may use a large scheduling quantum to reduce context switching costs but this choice can affect timing response.

Traditionally, applications in a general-purpose environment are written by different programmers and use separate processes across applications to provide address space isolation. However, these processes are scheduled using a preemptive discipline which introduces unpredictable timing behavior, especially in the presence of issues such as priority inversion, lock preemption, livelock, deadlock, etc. As a result, applications that are time-sensitive may not run at the desired time, get the desired

allocation, or may miss important deadlines, and these problems are hard to understand or debug.

Unpredictable timing in general-purpose systems is a symptom of a more fundamental problem that time-sensitive applications are not aware of the timing requirements of other time-sensitive applications. If such applications were aware of the timing requirements of other applications, they may be able to accommodate the others, and vice versa. Furthermore, by sharing timing information, these applications may be able to more effectively adapt their behavior while still preserving timeliness during overload.

In this paper, we focus on providing support for adaptive, time-sensitive applications in general-purpose operating systems. We present an event-driven application model that we call *cooperative polling* and a simple extension to the operating system interface that is aimed at facilitating cooperation between time-sensitive applications.

Applications in our cooperative polling model use events that have deadlines or priorities associated with them that serve as the basis for intra-application timing and inter-application cooperation. Events in our model are either triggered by deadlines or are best-effort. Deadline events have deadlines associated with them and they execute after their deadline has occurred, while best effort events have priorities associated with them and they execute in priority order.[1] The application will typically generate a mix of the two types of events during the course of its execution. For example, a simple video decoding application will generate deadline events for display and best-effort events for decoding. This type of model is often called reactive, because many events are generated in reaction to external activities such as arrival of data from the network.

The cooperative polling model has three characteristics that make it suitable for adaptive, time-sensitive applications. First, it uses a cooperative scheduling mechanism that aims to run events to completion. This approach reduces involuntary preemption which can cause unpredictable timing. Second, it uses a simple deadline-based scheduling policy of giving precedence to deadline events over best-effort events so that the timing constraints of the deadlines events can be met. Finally, the model supports inter-application cooperation by sharing event information such as deadlines and priorities across applications. This information is used within a time-sensitive application to voluntarily yield at appropriate times. Since our applications use an event model, voluntary yielding does not impose additional burden on the programming model. Through reciprocal cooperation, the model achieves overall predictable timing, minimizes involuntary preemptions and enables application-centric adaptation during overload.

A cooperative-polling application specifies its event information to the kernel that then conveys this information to other similar applications. The kernel also performs policing (i.e., preemption) when applications fail to yield at the appropriate times and hence the kernel ensures that applications get their desired allocations. Furthermore, these applications are run as processes and get the same benefits of address space isolation as traditional applications.

We have implemented a video streaming application called QStream that uses cooperative polling and adapts video quality based on available CPU and network bandwidth [15, 14]. Our evaluation shows the benefits of using cooperative polling to achieve predictable timing with single as well as multiple media applications both in underload and overload.

The following sections describe our approach in detail. Section 2 discusses related work in the area. Section 3 provides motivation for our approach, and then Section 4 presents the cooperative polling model. Section 5 describes the implementation of the model and discusses some aspects of the QStream application. Section 6 presents our evaluation, and Section 7 presents our conclusions.

2 Related Work

Traditionally, operating systems provide a process interface that isolates the different execution contexts, similar to virtual machines. However this isolation does not provide any timing guarantees. The cooperative polling model supports time-sensitive applications by facilitating cooperation between these applications via sharing of their internal deadlines or priorities.

Our model is most closely related to split-level scheduling [11] which aims to correctly prioritize user-level threads in different address spaces while minimizing user/kernel interactions. The main difference is that the higher priority classes in split-level scheduling can preempt the lower priority classes, while our model aims to avoid preemption altogether via cooperative yielding.

Both scheduler activations [3] and our model aim to avoid the ill-effects of preemption by informing the user level about the scheduling decisions made by the kernel. However, the main difference is that activations are upcalls that inform the application when a new scheduling decision is made while our model uses application-level polling to synchronize with the kernel's scheduling decisions. This difference is partly a result of the different application domains: activations are mainly designed for

[1] Although we use the term deadline, note that the event dispatcher in our model never skips or drops events. It is the responsibility of the application to adapt to delayed events.

throughput-oriented applications where the upcall model is easier to use, while cooperative polling is mainly designed for time-sensitive applications where polling is a commonly used method to meet timing requirements. Also, with activations, the user level only informs the kernel when it yields the processor,[2] while with cooperative polling, applications also inform the kernel about their deadlines or priorities.

While cooperative polling is implemented by applications, soft timers [4] is a kernel-based polling approach. It uses trigger states such as kernel entry points to efficiently schedule events (e.g., packet transmission). Both approaches aim to avoid unnecessary preemption or interrupts.

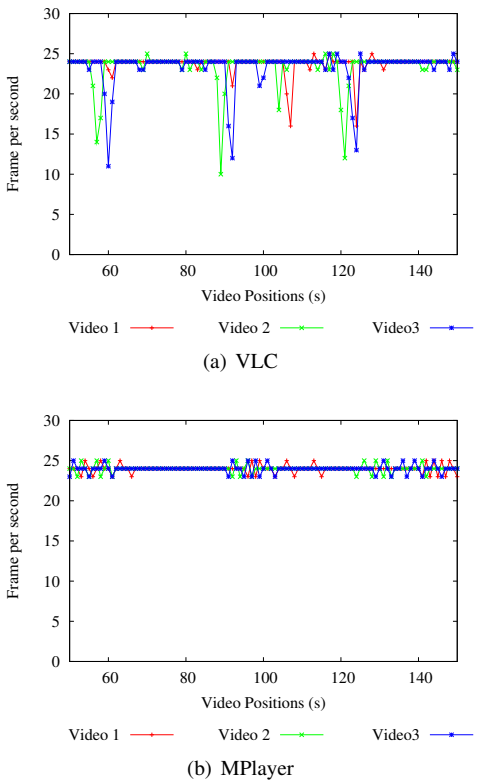
Although the relative merits of event-driven and multithreaded architectures remain contentious [16, 23, 30, 1, 29], generally events are considered to offer better performance while threads are considered to offer ease of programing. Our model uses events because they appear to be a natural match for time-sensitive applications that are designed to quickly respond to external input. For example, the worst-case execution time (WCET) of a job or a response is an important metric for time-sensitive applications. This metric is trivial to instrument with events. Nevertheless, we believe that it is possible to use non-preemptive threads libraries such as Pth [7] as an alternative for implementing cooperatively-polled applications.

Currently, our model supports applications that are primarily single threaded. Zeldovich et. al. [31] provide multiprocessor support for event-driven programs. Their approach would directly apply to our work.

While events have been used extensively, most event systems have focused on the efficiency and scalability advantages of events rather than predictable timing. As a result, most event systems do not distinguish between deadline and best-effort events. The PulseAudio [24] sound server uses events that are closest to our model. It uses separate event types corresponding to our deadline and best-effort types, but unlike our model, applications cannot control the order of best effort events.

There exists a large body of work that aims to provide operating system support for multimedia and real-time applications [18, 12, 13, 20, 28, 5, 25, 10]. Based on our experiences, we have found that preemption is one of the most serious concerns when implementing time-sensitive applications. As a result, our model avoids using preemptive kernel threads and requires minimal support, in the form of sharing timing information, from the kernel. Our user-level scheduler is simple and gives precedence to deadline events [17] over best-effort events. The reason this approach works well is that deadline events are typically short and do not overload the CPU in our applications, while best-effort events can overload the CPU but are designed to be adaptive.

[2]With multi-processors, the user level can also ask for more processors.



(a) VLC

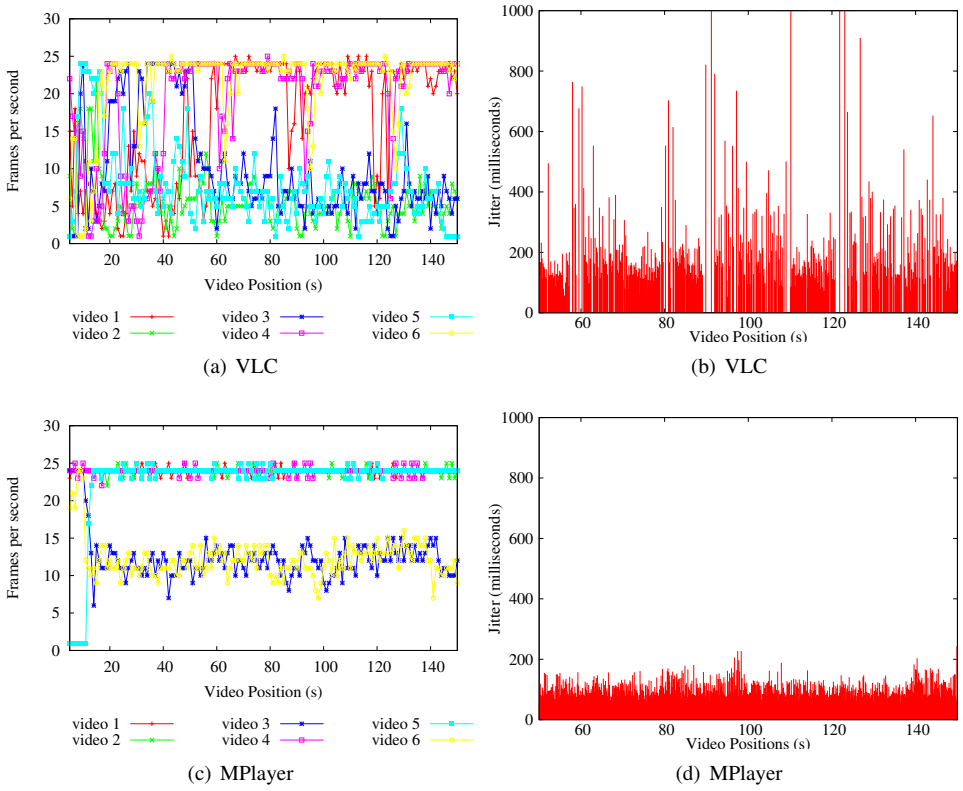
(b) MPlayer

Figure 1: Performance of the VLC and MPlayer video players in underload

Our model focuses on time-sensitive applications that can adapt during overload. Our QStream video streaming application uses a technique called Priority-Progress to adapt to available network bandwidth [15] and CPU availability [14]. This technique was inspired by several other works on quality-adaptive streaming [26, 8, 27]. Seda [30] provides a framework for performing overload management primarily for throughput-based applications such as Internet services.

3 Motivation

The main motivating applications for this work are video streaming over mobile devices and high-quality streaming. A variety of video-capable mobile devices such as smart phones, PDAs, Internet tablets, iPods, *etc.* have become available today. These mobile devices have modest CPU speeds because of the need to conserve power usage and maximize battery life. Our previous work on CPU adaptation [14] is part of a project to port our QStream software to two such devices, the Series 60 (S60) smart phones [22] and the Nokia 770 Internet Tablet [21].



(a) VLC (b) VLC

(c) MPlayer (d) MPlayer

Figure 2: Performance of the VLC and MPlayer video players in overload.

At the other end of the spectrum, high-quality streaming is highly resource intensive, and it has significant concurrency and tight timing requirements. It must concurrently handle network processing, audio and video compression and I/O to several devices (audio, video, storage). Rich media with multiple streams increases concurrency even further. In addition, audio and multiple video streams are time constrained and must be synchronized before presentation. The timeliness requirements of these streams are typically in millisecond timescales. These requirements are at least an order of magnitude tighter than the response requirements of typical throughput-oriented applications such as web servers, and they are challenging for general-purpose CPU schedulers that have historically been biased towards improving throughput.

Our previous work [14] measured the performance of two of the most popular open source video players, MPlayer [9] and the VideoLan Client (VLC) [2]. As a motivation for this work, we present these results here again. We chose two players because they have substantially different architectures: MPlayer is event-driven while VLC is multi-threaded, and we chose open source players because they could be instrumented easily. Both these players are mature and of high quality and support frame-rate adaptation during CPU overload. They have been implemented by large teams of talented and dedicated developers and have a large user base.[3]

We measured the performance of MPlayer and VLC by playing 3 and 6 videos (the same video) simultaneously. Each video is played in a separate player. The CPU usage remains just below saturation with 3 videos while the CPU is fully saturated with 6 videos. These experiments were performed on a Dell Inspiron 1300 laptop PC, with a 1.8 GHz Pentium-M CPU and 512 MB memory running the Ubuntu Linux 6.06 distribution. The video is a movie taken from a DVD and converted to MPEG-4 format. We measured the frame rate and jitter of the videos. Frame rate is better for expressing the overall smoothness of video, while jitter is the inter-frame display time and it captures noticeable pauses better.

Figure 1 shows the frame rates of the three videos for the VLC and MPlayer video players during underload. Both players maintain close to the full frame rate of 24, and subjectively, the videos play with normal smoothness and no noticeable pauses.

Figure 2 shows the performance of VLC and MPlayer during overload (CPU usage is pegged at 100%) when the

[3]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.

frame-rate adaptation mechanism was active in both the players. The graphs on the left show that the frame rates of the videos varies dramatically. Both players exhibit bimodal fairness with many of the videos experiencing zero or low frame rates and some that have almost full frame rate. Figure 2(b) shows that jitter reaches up to one second for VLC, which experiences several pauses. MPlayer is able keep jitter below 200 ms.

To conclude, these players maintain acceptable performance during underload but not during overload. Interestingly, the event-driven MPlayer is more consistent than the threaded VLC player even though both players adapt quality similarly during overload. While we have not analyzed this difference in performance in detail, we believe it results mainly as a result of the interaction between the adaptation mechanism and the kernel's scheduling mechanism.

```
submit(EventLoop *l, Event *e);
cancel(EventLoop *l, Event *e);
run(EventLoop *l);
stop(EventLoop *l);
```

Figure 3: Basic event API

```
struct Event {
  enum { BEST_EFFORT, DEADLINE } type;
  Callback callback;
  TimeVal deadline;
  int priority;
  ...
};
```

Figure 4: Event type definition

4 Cooperative Polling

The cooperative polling model aims to provide support for applications that require timeliness and that adapt during overload. This model consists of an event-driven cooperative scheduling mechanism and a deadline-based scheduling policy. In addition, the model enables inter-application cooperation by sharing event information across applications. We discuss these aspects of the model below.

4.1 Cooperative Scheduling

The cooperative scheduling model is based loosely on the principles of reactive programming described below:

1. The model is event-driven with a per-thread event dispatcher that operates independently of event dispatchers in other threads. Program execution is a sequence of events or function invocations that are run non-preemptively or cooperatively.

2. Events must avoid actions that can block or sleep.

3. Events should avoid long running computations.

Adaptive, time-sensitive applications have some computations that have deadlines while others are best effort. Our model avoids using preemptive kernel threads by scheduling these computations non-preemptively at the user level. The lack of preemption implies that reactive programs are generally free of locking and synchronization primitives required in multi-threaded programs.

The second rule against blocking is generally challenging to satisfy in practice. However, we have implemented an asynchronous I/O subsystem described in Section 5 that eases programming significantly. The third rule may seem the most counter intuitive. Obviously, long computations may be inherent to the task at hand (e.g. decompressing video). However, most long computations use loops and this rule simply means that reactive programs must divide the iterations of long running loops into separate events. The focus on short non-blocking events promotes a environment that allows software to quickly respond to external events when they occur and hence the name reactive.

Figure 3 lists the key primitives in our scheduling model. The application calls `submit` to submit an event for execution. To initiate dispatching of events, the application calls `run`, which normally runs for the lifetime of the application. The application must submit at least one event before calling run, and it calls `stop` from within one of its events to end the dispatching of events. The application can also call `cancel` to revoke an event it had previously submitted.

4.2 Scheduling Policy

The scheduling policy component of our model aims to provide predictable timing by reducing scheduling latency. It distinguishes between deadline-based and best-effort events and gives precedence to deadline events so that their timing constraints can be met.

Figure 4 shows the type definition of an event. An application specifies each event as either a best-effort or a deadline event. The `callback` specifies the function that will handle the event and any data arguments to be passed. The `deadline` field specifies an absolute time value. Deadline-based events are not eligible for execution until the `deadline` time has passed. Once eligible, deadline events take priority over all best-effort events.

The `priority` field is used by best-effort events. It is up to the application to use priorities to control execution

```
run(EventLoop l) {
  do {
    if (head_expired(l.deadline_events)) {
      e = q_head(l.deadline_events);
      callback_dispatch(l, e);
      cancel(l, e);
    } else if
      (q_not_empty(l.best_effort_events)) {
      e = q_head(l.best_effort_events);
      callback_dispatch(l, e);
      cancel(l, e);
    } else {
      yield(l);
    }
  } while (l.stop != True);
}

yield(EventLoop l) {
  if (q_not_empty(l.deadline_events)) {
     sleep until next deadline;
  } else {
     l.stop = True;
  }
}
```

Figure 5: Event dispatcher algorithm

order. For example, in a video application it is important to keep sound uninterrupted because users are sensitive to audio glitches. Hence, the application would assign a high priority to events related to servicing the audio output device. When best-effort events have the same priority, we overload the deadline field and use it as a secondary sort key for ordering best-effort events. For the rest of the paper, we call this combination the priority of the best-effort event.

Figure 5 shows the the event dispatch algorithm. The deadline and best-effort events are stored in the `deadline_events` and the `best_effort_events` priority queues, and the `submit` and `cancel` operations are realized by insertion and removal from these queues. These operations are idempotent and have no effect if the event is already submitted or canceled, or is a null event.

The dispatcher simply services *all* events as provided by the application even when events arrive faster than they are dispatched. This approach can cause the queue fill levels to increase, perhaps unboundedly, if overload is persistent (e.g. the CPU is just too slow for the given application). However, we chose this approach because it makes the scheduling policy and the dispatcher simple and predictable, and also because we believe that effective overload response requires application-specific adaptation. Our QStream video client implements such adaptation by reducing the generation of certain events and invoking `cancel` for some existing events to skip the less important steps of video decoding as necessary to maintain timeliness [14].

```
yield(EventLoop l) {
  cancel(l, l.coop_deadline_event);
  cancel(l, l.coop_best_effort_event);
  if (q_non_empty(l.deadline_events) ||
      q_non_empty(l.best_effort_events)) {
    // coop_poll sleeps until next deadline
    coop_poll(q_head(l.deadline_events),
              q_head(l.best_effort_events),
              &l.coop_deadline_event,
              &l.coop_best_effort_event);
    l.coop_deadline_event.callback =
    l.coop_best_effort_event.callback =
       yield;
    submit(l, l.coop_deadline_event);
    submit(l, l.coop_best_effort_event);
  } else {
     l.stop = True;
  }
}
```

Figure 6: Dispatcher support for inter-application cooperation

4.3 Inter-Application Cooperation

We enable cooperation between time-sensitive applications with one new primitive, `coop_poll`. This function yields the processor to another cooperatively polled thread that either has the next expired deadline or has the highest priority best-effort event. It takes two IN and two OUT parameters. The two IN parameters specify the most important deadline and best-effort events in the current thread. These values are used to wake up the thread at its next deadline or when its best-effort event has the highest priority among all threads. When the `coop_poll` call returns, the two OUT parameters are set to the most important deadline and best-effort events across all other threads. This information is used by the current thread to yield the processor as well as by the kernel to preempt the thread if the thread fails to yield. Section 5 discusses an implementation of this primitive.

Figure 6 shows the use of the `coop_poll` call in a modified `yield` function that enables inter-process cooperative scheduling. The `run` routine remains unchanged from Figure 5. This `yield` function is designed so that events are executed across threads in the same order as events in the single-threaded dispatcher function shown in Figure 5.

The first two arguments in the call to `coop_poll` export the thread's own most important deadline and best effort events. To enable sharing, we add two proxy events

to the event loop state, `coop_deadline_event` and `coop_best_effort_event`, that act on behalf of other applications. The deadline and priority values of these proxy events are set by `coop_poll` to reflect the most important deadline and best effort event of all the other applications. After the `coop_poll` call, the proxy events are submitted to their respective event queues in the current thread. The callback function for these events is set to `yield` so that the current thread yields voluntarily to other applications in the `callback_dispatch` routine shown in Figure 5.

The `cancel` calls at the beginning ensure that the event queues contain only events internal to the current process. This in turn prevents yield from spinning where a thread transitively yields to itself.

5 Implementation

The implementation of cooperative polling consists of three main components, the coop_poll function, an event-driven polling library and an asynchronous I/O library that eases programming. We describe these components in turn. We also briefly describe the QStream application that we will use to evaluate our model.

5.1 Coop_Poll

The `coop_poll` primitive has a relatively straightforward implementation. It uses two priority queues, one for deadline events and another for best-effort events. The first two arguments of the call (see Figure 6) are inserted into the queues. The next thread is scheduled based on the head event in the deadline queue, or if it is empty then the best-effort queue. This event is removed and the last two arguments of the call are filled with the new head events of the two queues.

When the deadline queue is empty, `coop_poll` sets the deadline to a maximum timeslice value as determined by the process scheduler. This approach is designed to allow scheduling of regular best-effort jobs when only cooperatively-polled best-effort jobs are present in the system [11, 12].

Currently, we have implemented `coop_poll` in our user-level polling library. With this implementation, the kernel does not preempt threads that fail to yield. We plan to implement this call in the kernel with the Bossa scheduling framework [19]. This implementation will allow experimenting with other scheduling policies that provide more control for interactive and other regular best-effort jobs.

5.2 Cooperative Polling Library

Although there are programming languages designed specifically for reactive programming, our event-driven model is implemented in C, so the principles of reactive programming are implemented by idiom, rather than enforced by the language or runtime.

Our cooperative polling library is called `libqsf`, and it has several similarities with other event libraries. Although it was initially implemented as part of the QStream framework, it is generic and intended to be used by other time-sensitive applications. The core of our scheduling model is implemented with an event dispatcher that uses a pair of priority queues, one for each type of event. For efficiency, these queues are implemented using a heap data structure.

5.3 Asynchronous I/O

Besides the core event dispatch primitives, `libqsf` also provides a complementary set of primitives for performing asynchronous I/O. The use of asynchronous I/O (AIO) is an integral part of the overall reactive model, because it helps ensure the rule that events should never block or sleep. The AIO API in libqsf is similar to the synchronous I/O API provided by POSIX standards. Figure 7 lists some of the primitives. The meaning of most arguments is analogous to counterparts in the traditional socket API. The extra `callback` argument reflects the asynchronous semantics. These primitives always return immediately and the callback is dispatched when the requested operation is complete. This API is implemented using a combination of our event mechanism and the standard socket API in non-blocking mode. The AIO facility takes care of tracking the progress of I/O operations and managing the details of invoking kernel notification facilities such as the `poll()` system call. We believe that these services greatly reduce the burden imposed on the programmer by the non-blocking rule of our reactive model.

Even with the AIO facility, it is not always feasible to implement entire applications using our cooperative polling model. The reason is that much existing software is designed only for the synchronous, multi-threaded paradigm. For instance, POSIX functions such as `stat` (provides details about a file such as its size and modification date) and `getaddrinfo` (translates DNS names to IP addresses) provide the only way to accomplish the respective tasks, and there is no easy way for an application to prevent them from blocking or sleeping. Similarly, useful libraries such as Berkeley DB (among other things, it provides a robust and mature implementation of B-trees) often only provide synchronous APIs.

We handle these situations by reverting to the common strategy of employing helper threads. The helper thread

```
aio_accept(Aio *listen_fd, AioAddress *addr, Aio *fd, Callback *callback)
aio_connect(Aio *fd, AioAddress *addr, Callback *callback);
aio_read(Aio *fd, void *buf, size_t length, Callback *callback);
aio_write(Aio *fd, void *buf, size_t length, Callback *callback);
...
```

Figure 7: Asynchronous I/O API in `libqsf`

performs the synchronous operation on behalf of the main thread and communicates with the main thread using AIO (e.g., via a local socket), thereby isolating the main thread from the blocking operation. In theory, this approach reintroduces preemption and non-determinism that we have sought to avoid with the reactive model. However, in practice it has not caused a significant problem for our applications. So far, the demand for worker threads has only arisen for operations that are I/O dominated and computationally light. Hence, these threads spend most of their time sleeping and have a negligible scheduling effect on the main program thread.

To ease the process of isolating synchronous APIs, `libqsf` provides primitives that handle the details of worker thread creation, creation of an optional private event dispatcher within the helper thread and communication with the main thread. In the future, we plan to use a promising and elegant technique called Lazy Asynchronous I/O [6] that converts synchronous system calls to asynchronous calls.

5.4 The QStream Video Application

As part of our broader work, we have implemented a complete adaptive video streaming system called QStream. QStream is a substantial application consisting of over 100,000 lines of code, mostly written in C. It uses the `libqsf` library and is implemented entirely using the cooperative polling model. QStream uses a scalable video compression technique called Priority-Progress to adapt to available network bandwidth [15] and CPU availability [14].

The development of QStream led to the design and refinement of our cooperative polling model. The implementation of cooperative polling is entirely contained in the `libqsf` library. As a result, we were able to convert the QStream video streaming application to a cooperative polling application simply by re-linking against a modified version of libqsf.

6 Evaluation

In this section, we evaluate the performance of cooperative polling using our QStream video streaming system. For all the results in this section, the experimental setup consists of a pair of PCs connected by a LAN, one acting as a video server and the other as the player. The server machine is also responsible for collecting the measurement data produced by our player. The video player machine is a white box desktop PC with a 3.0 GHz Intel Pentium 4 and 1 GB memory running the Fedora Core 5 Linux distribution.

Similar to Section 3, we play multiple, concurrent streaming videos as the workload. Each video is different, and has both variable bit rate and variable processing requirements. We evaluate scheduling performance by measuring the timeliness and the quality of the videos.

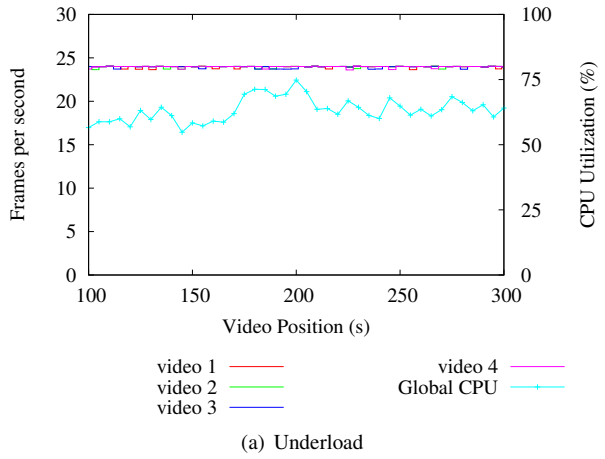
Below, in Section 6.1, we first use the basic reactive event model as described in Sections 4.1 and 4.2 to establish baseline performance. Then, in Section 6.2, we examine the performance of the cooperative polling event model as described in Section 4.3.

6.1 Single Player Performance

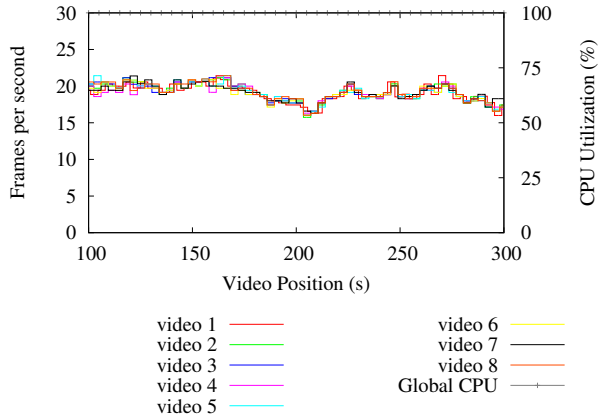
For the baseline performance, we play multiple videos all within a single application thread. Similar to the measurements presented in Section 3 with MPlayer and VLC, we measured the performance of QStream across a range of load levels by varying the number of videos played simultaneously. From these loads, we selected the number of videos just below the point where CPU saturation occurs, and then we doubled that amount to induce heavy CPU saturation. For the machines used in our experiments, the number of videos is four and eight for the underload and overload cases respectively.

Figure 8 shows the frame rate and the total CPU utilization of all the videos in the QStream application in the underload and overload cases. Figure 8(a) shows that that QStream timeliness in underload, similar to the MPlayer and VLC results, is excellent and all videos play at the maximum frame rate. In contrast, Figure 8(b) shows that QStream's performance in overload degrades much more gracefully than MPlayer and VLC. QStream is able to adjust video quality fairly across videos, avoiding major timing interruptions, despite the fact that the CPU is saturated (100% utilization) the entire time.

The comparison above takes advantage of a feature unique to QStream in that it plays all the videos within a

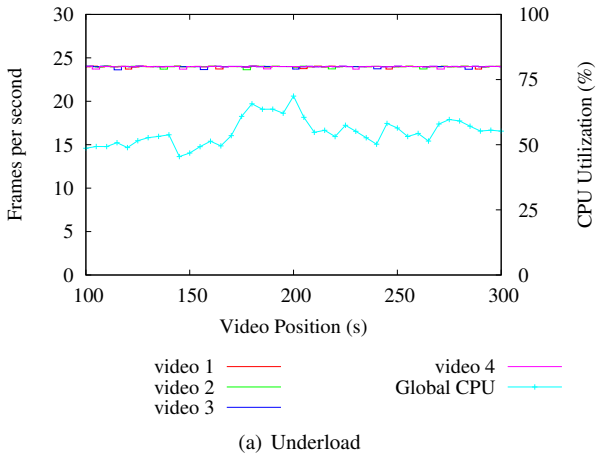


(a) Underload

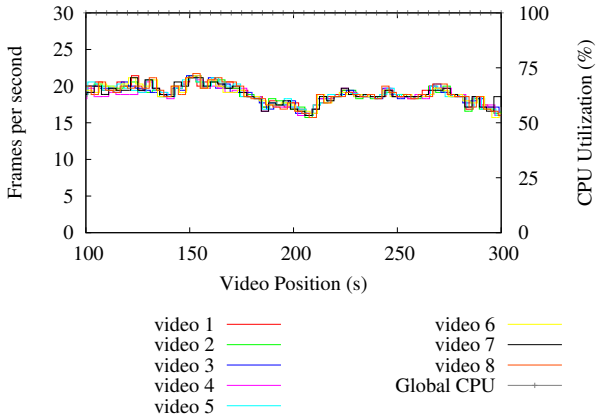


(b) Overload

Figure 8: Monolithic QStream Video Player in underload and overload



(a) Underload



(b) Overload

Figure 9: Cooperative QStream Video Player in underload and overload

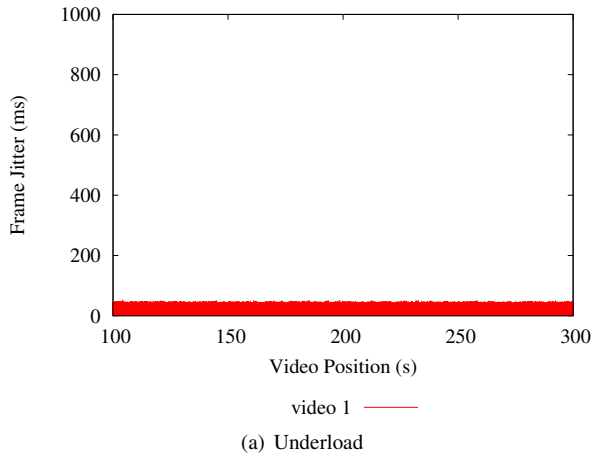
single OS thread in these tests, an approach we call monolithic playback. Monolithic playback allows QStream maximum control over its own timing. The performance advantages enjoyed by QStream are a result of a combination of its adaptation method, and a single-threaded approach which is a work-around to kernel scheduling issues that hampered the other players. Nevertheless, the performance of QStream's monolithic playback provides us a useful baseline for how well we can expect to perform with multiple time-sensitive applications.

6.2 Multiple Player Performance

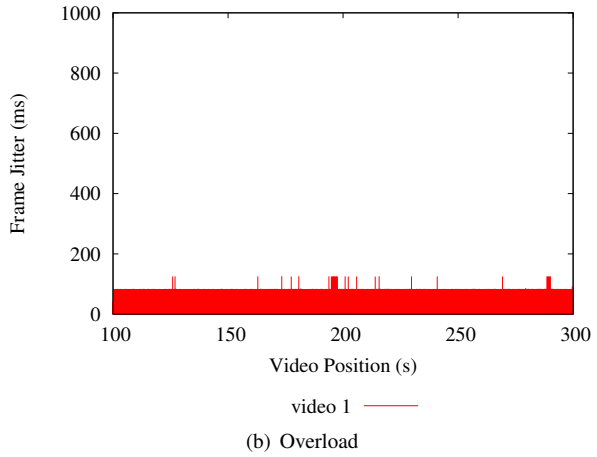
This section repeats the same workloads of the previous section, but instead of the monolithic playback approach, we now start a separate QStream player process for each video (4 for underload and 8 for overload), and allow these processes to communicate via our cooperative polling mechanism. Figure 9 shows the results of this experiment. Figure 9(a) shows that the QStream player maintains the maximum frame rate for all the videos in underload, similar to MPlayer and VLC. Compared to these players, the strength of our approach lies in the overload case shown in Figure 9(b). All videos have the same consistent frame rates in overload. The cooperative polling mechanism is able to maintain the advantages of the monolithic baseline both in underload and overload.

Frame rate is better for expressing the overall smoothness of video, while jitter is the inter-frame display time and captures noticeable pauses better. Figure 10 shows frame jitter for the frames displayed in the underload and overload workloads respectively. QStream avoids any major frame jitter spikes, which subjectively translates into consistent temporal quality for the viewer.

There is a slight divergence between the monolithic and cooperative polling results in Figure 8(b) and Figure 9(b) despite the fact that we have designed cooperative polling to execute events in an order that is as close

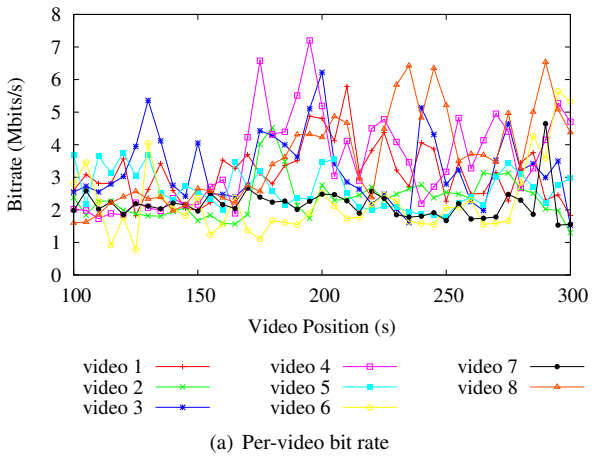


(a) Underload

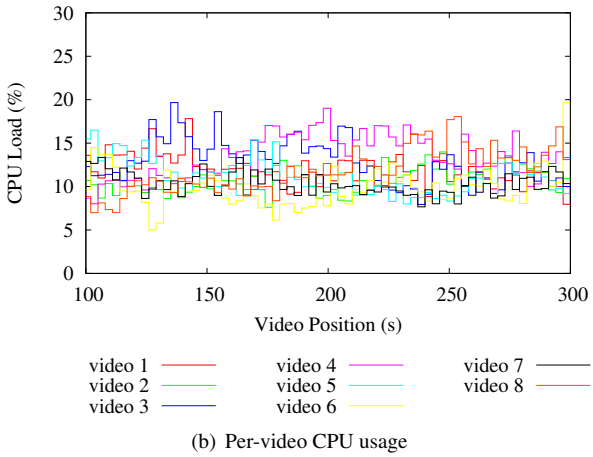


(b) Overload

Figure 10: Frame jitter for QStream with cooperative polling in underload and overload



(a) Per-video bit rate



(b) Per-video CPU usage

Figure 11: Dynamics of video playback for the cooperative QStream players during overload

to the monolithic case as possible. There are numerous potential sources of non-determinism that could account for this difference, but we suspect the biggest contributing factor is polling for asynchronous I/O completions. In the monolithic case, polling occurs for all pending I/O of all the videos in one shot, whereas in the cooperative case, each process only polls for completion events relating to its own video. We are considering possible solutions to mitigate this discrepancy, but leave them to future work.

To better understand the performance of cooperative polling, we present several related measurements. Figure 11 shows the challenges inherent in continuous media workloads. Figure 11(a) shows the bit rate over the timeline of the experiment for each of the eight videos. The bit rate is highly variable, which is generally considered a formidable CPU scheduling challenge. Figure 11(b) shows the CPU usage for each of the QStream video players that make up our workload. While we achieve our desired goal of same video quality as shown in Figure 9(b), the corresponding CPU allocations is Figure 11(b) are highly variable. These results indicate that application-specific quality adaptation using conventional heuristic-based kernel CPU scheduling is unlikely to yield good results.

The predictable timing and the address space isolation of the co-operative polling approach comes with nominal overhead. We instrumented our cooperative poll implementation and measured the rate of voluntary context switches. Although these numbers are high, we find that the impact on performance is relatively low. For the underload workload (four videos), the number of context switches is roughly 1252 per second, while for the overload workload (8 videos) the context switch rate is about roughly 3630 per second. Figure 12 shows the aggregate frame rate across all videos in underload and overload respectively. In Figure 12(a), the total frame rate is constant at 96 fps because all the four videos play at the maximum frame rate. In overload, the average throughput is

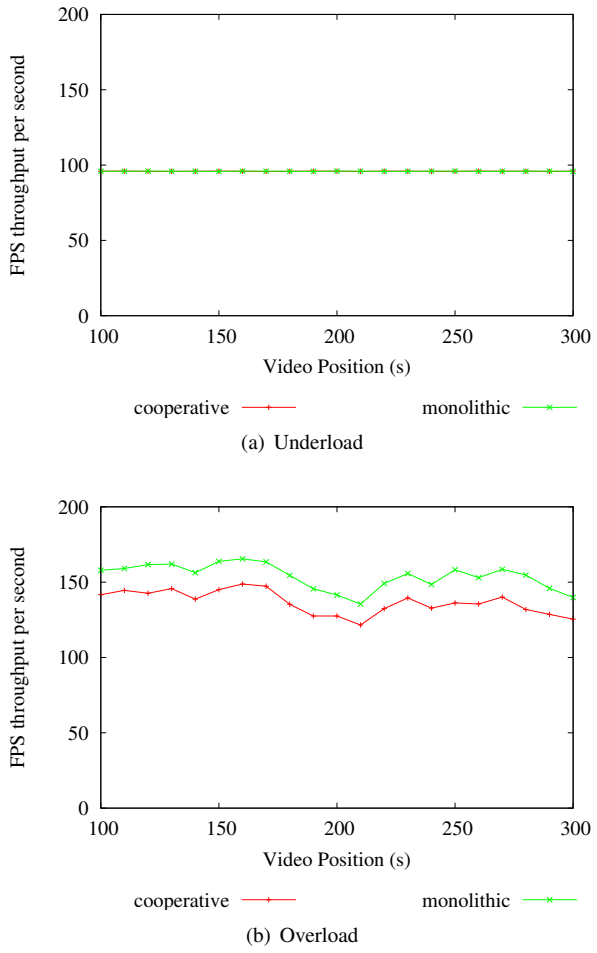


Figure 12: Overall throughput: monolithic vs cooperative

150.2 and 133.5 for the monolithic and cooperative cases respectively. With eight videos running simultaneously, the cooperative case has a 12.5% overhead in terms of frame rate. This overhead is lower with fewer videos because the context switch rate goes down.

The last aspect of performance concerns the timeliness of our approach in absolute terms, independent of our video application. Our event dispatcher is instrumented to measure the *dispatch latency*. Recall from Section 4.2 that our event dispatcher only invokes the callback for deadline events after the deadline has passed. The dispatch latency is the difference between the time when a deadline event is actually serviced and the deadline value specified in the event by the application.

Figure 13 shows the dispatch latency for one of the videos through the course of the experiment. Each spike in the figure shows the worst case dispatch latency for one or more events that occur close together. This graph shows the timeliness that can be expected from the cooperative polling approach. We believe that these results

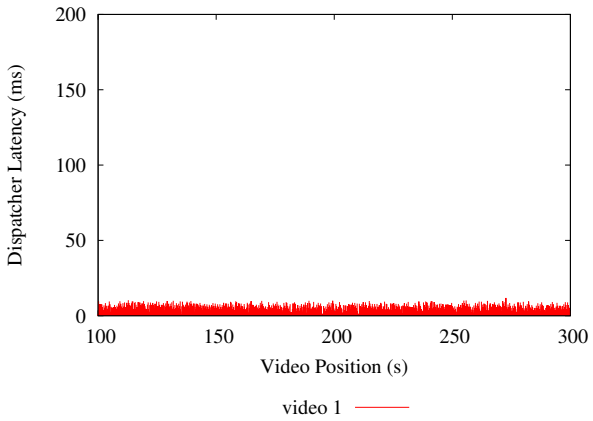


Figure 13: Dispatch latency for one cooperative QStream player during overload

show that our cooperative polling approach holds great promise in supporting a variety of time sensitive applications in general-purpose operating systems.

7 Conclusions

An important goal while developing our QStream video streaming application has been to gain a deeper understanding of the requirements of large-scale, time-sensitive applications. Our motivating applications have been video-enabled mobile devices and high-quality media streaming, both of which are resource intensive and require adaptive applications. This combination of time constraints, heavy resource demands and adaptation is challenging for current systems. This limitation led to the design and implementation of our event-driven cooperative polling model.

Cooperative polling aims to reduce unpredictable timing by minimizing involuntary preemption, and it facilitates cooperation between applications by sharing event information such as deadlines and priorities across applications. Our evaluation has shown that this approach together with a simple deadline-based scheduling policy achieves overall predictable timing, and it allows applications to make adaptation decisions during overload that cooperate with rather than get overwhelmed by the kernel's scheduling policy.

The QStream application as well as the cooperative polling framework is open source software and is available at `http://qstream.org`.

References

[1] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without

manual stack management. In *Proceedings of the USENIX Annual Technical Conference*, June 2002.

[2] L. Aimar et al. VideoLan. `http://www.videolan.org/`.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Efficient kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(3):53–79, Feb. 1992.

[4] M. Aron and P. Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3):197–228, Aug. 2000.

[5] S. Childs and D. Ingram. The Linux-SRT integrated multimedia system: Bringing QoS to the desktop. In *Proceedings of the IEEE Real Time Technology and Applications Symposium (RTAS)*, May 2001.

[6] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous i/o for event-driven servers. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[7] R. S. Engelschall. The GNU Portable Threads. http://www.gnu.org/software/pth/.

[8] N. Feamster, D. Bansal, and H. Balakrishnan. On the interactions between layered quality adaptation and congestion control for streaming video. In *Proceedings of the 11th International Packet Video Workshop (PV2001)*, pages 128–139, April 2001.

[9] A. Gereoffy et al. The FFMpeg Project. `http://www.mplayerhq.hu`.

[10] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. Supporting time-sensitive applications on a commodity OS. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180, Dec. 2002.

[11] R. Govindan and D. P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 68–80, Oct. 1992.

[12] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, Oct. 1996.

[13] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–211, Oct. 1997.

[14] C. Krasic, A. Sinha, and L. Kirsh. Priority-progress CPU adaptation for elastic real-time applications. In *Proceedings of the Multimedia Computing and Networking Conference (MMCN)*, Jan. 2007. To appear.

[15] C. Krasic, J. Walpole, and W. chi Feng. Quality-adaptive media streaming by priority drop. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 112–121, June 2003.

[16] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[17] C. L. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan 1973.

[18] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

[19] G. Muller, J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *Proceedings of the High Assurance Systems Engineering Conference (HASE)*, pages 56–65, Oct. 2005.

[20] J. Nieh and M. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 184–197, Oct. 1997.

[21] Nokia. Nokia 770 Internet Tablet. `http://www.nokia.com/770`.

[22] Nokia, Symbian, et al. Series 60 Smartphone Software Platform. `http://www.s60.com/`.

[23] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[24] L. Poettering, P. Ossman, S. E. King, et al. PulseAudio Sound Server. `http://0pointer.de/lennart/projects/pulseaudio/doxygen/`.

[25] J. Regehr and J. A. Stankovic. Augmented CPU reservations: Towards predictable execution on general-purpose operating systems. In *Proceedings of the IEEE Real Time Technology and Applications Symposium (RTAS)*, pages 141–148, May 2001.

[26] R. Rejaie, M. Handley, and D. Estrin. Quality adaptation for congestion controlled video playback over the Internet. In *Proceedings of the ACM SIGCOMM*, pages 189–200, October 1999.

[27] D. Sisalem and F. Emanuel. QoS control using adaptive layered data transmission. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, June 1998.

[28] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 145–158, Feb. 1999.

[29] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for Internet services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 268–281, 2003.

[30] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[31] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the USENIX Annual Technical Conference*, pages 239–252, June 2003.