Fair and Timely Scheduling via Cooperative Polling

Charles Krasic Mayukh Saubhasik
Anirban Sinha
Department of Computer Science
University of British Columbia

Ashvin Goel
Electrical and Computer Engineering
University of Toronto

Abstract

Advances in hardware capacity, especially I/O devices such as cameras and displays, are driving the development of applications like high-definition video conferencing that have tight timing and CPU requirements. Unfortunately, current operating systems do not adequately provide the timing response needed by these applications. In this paper, we present a hierarchical scheduling model that aims to provide these applications with tight timing response, while at the same time preserve the strengths of current schedulers, namely fairness and efficiency. Our approach, called cooperative polling, consists of an application-level event scheduler and a kernel thread scheduler that cooperate to dispatch time-constrained application events accurately and with minimal kernel preemption, while still ensuring rigorously that *all* applications share resources fairly. Fairness is enforced in a flexible manner, allowing sharing according to a mixture of both traditional resource-centric metrics and new application-centric metrics, the latter being critical to support graceful application-level adaptation in overload. Unlike traditional real-time systems, our model does not require specification or estimation of resource requirements, simplifying its usage dramatically. Our evaluation, using an adaptive video application and a graphics server, shows that our system has event dispatch accuracies that are one to two orders of magnitude smaller than are achieved by existing schedulers. At the same time, our scheduler still maintains fairness and has low overhead.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Managemnent—Scheduling

General Terms Design, Measurement, Performance

Keywords Fairness, Timeliness

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'09, April 1–3, 2009, Nuremberg, Germany.
Copyright © 2009 ACM 978-1-60558-482-9/09/04. . . $5.00

1. Introduction

Over the years, there have been impressive improvements in computing capacity. Today, common personal computers rival the supercomputers of the not too distant past, in measures such as processor speed, storage capacity, communication bandwidth, etc. These advances are allowing the development of desktop applications with demanding timing requirements. For example, high-resolution video displays and cameras have become commonplace today, and video conferencing applications using these devices are increasingly being used to reduce the need for face-to-face meetings involving long-distance travel. These applications perform numerous activities such as capture and output of audio and video, echo cancellation and network streaming, all of which have stringent timing requirements. For example, packets need to be received by the client-side application with minimal delay to reduce overall lag or frame skipping. Another class of applications with similar requirements is gaming. Video gamers prefer frame rates upwards of 100 fps (10 ms per frame) due to rapid motion and close viewing conditions, and so gaming displays are often advertised as having under 5 ms response time. Therefore, gaming applications have challenging requirements because they need to perform all their other activities, including I/O and computation, within 5 ms for each frame.

An emerging trend is a class of applications that have both tight timing and heavy CPU requirements. For example, digital multi-track audio processing software used by musicians, has tight timing constraints, and also substantial CPU needs due to sophisticated graphical visualization. Similarly, high resolution I/O devices such as 24 inch displays with 1920x1080 resolution are easily affordable today (they cost less than $400). These displays support full HD quality video (i.e., 1080p), but the data rates and the de/compression needs of HD continuous media applications routinely push the computational limits of available processors. In addition, these applications are increasingly being designed to support diverse environments, from ultra high-resolution for tele-immersion [Yang 2007] to constrained resolution for mobile platforms. An appealing idea is "encode once, run everywhere", wherein data is encoded and

stored once scalably, spanning the entire range of low to high resolutions [Schwarz 2007]. The applications adapt the data on the fly, at one or several stages of processing, based on availability of resources such as CPU capacity, storage, network bandwidth or display size. This *adaptive* approach can also be used to serve the same content to different clients simultaneously, and multiple independent content to the same client (e.g., real-time editing of video sports feeds). These adaptive applications are time-sensitive *and* they can saturate resources because they aim to provide the highest possible quality based on resource constraints.

Current Approaches While hardware technology has advanced exponentially and these sophisticated applications have become available, unfortunately, modern operating systems do not provide the timing response needed by these applications. Poor timing, especially under heavy load, results in quality artifacts such as stutters, skips, hangs, and auditory and visual glitches, limiting the use of these applications. Traditionally, commodity operating systems aim to provide fairness and high throughput, at the expense of timing response. For example, Unix-based feedback schedulers use low CPU usage as a heuristic to give priority to interactive applications [Corbato 1962]. However, continuous media applications may have high resource requirements as described above, and thus may not be run at the desired times.

Consider various options for providing improved timing. The quantum of the scheduler can be reduced so that the OS can respond to application events more rapidly, but that contradicts with the throughput goal, due to increased context switching. Alternatively, a time-sensitive application can be strictly prioritized at the expense of other applications but that contradicts with fairness. In the worst case, a tight-loop bug in a real-time application can effectively lockup the entire system. Another alternative is to use reservation schemes [Mercer 1994, Leslie 1996, Jones 1997] that provide each thread an allocation, consisting of a certain proportion of the CPU over a certain period of time, ensuring that the deadlines of real-time tasks are satisfied. Best-effort tasks can coexist because they have a certain minimum allocation to avoid starvation. These schemes require specifying resource requirements, which is non-trivial in general-purpose environments. It is possible to estimate requirements [Steere 1999, Lu 2000, Yang 2008], but a more fundamental problem with reservations is that the resource can be allocated anytime during the period. If an application has tight timing requirements (e.g., packets should be delivered to the application within one ms after they are received), the period has to be short, which can dramatically increase context switching overhead, since a single application with a short period prevents the scheduler from allowing any other application to run for a long duration. As a result, reservations have been mainly used to provide coarse-grained timing response (e.g., 30 ms granularity for displaying video frames).

Our Approach We believe that it is important for commodity operating systems to provide accurate timing and fairness to applications, without requiring resource specifications such as CPU allocation. Our key observation is that time-sensitive applications have some computations that are time synchronous (e.g., video display) and others that are best effort and can be adapted (e.g., video decoding), and these two types of computations need to be clearly identified so that their needs can be met independently. In particular, the time-synchronous computations should specify their timing needs to the OS scheduler so that it can dispatch them with low latency, and yet all applications, including time-sensitive applications, should share resources fairly with each other. Based on this observation, we present a programming model for designing these applications and a scheduler that meets their needs.

By default, our scheduler enforces max-min fairness among threads in terms of the measured amount of CPU time each executes. Thread weights may be adjusted to alter the default sharing. We refer to this as *resource fairness* since it is defined based on a resource metric, *i.e.* CPU time. Alternatively, we also provide a thread grouping mechanism that allows *application fairness*. Threads may be grouped, and their overall allocations subdivided among group members in application defined terms, for instance multimedia applications may determine fairness based on user-centric quality. In multimedia applications, our notion of resource fairness relates to the common notions Quality of Service (QoS), whereas application fairness relates to Quality of Experience (QoE).

Our approach provides timing accuracy (up to roughly one ms), fairness across all applications, has low overhead and requires no resource specification. It consists of a two-tiered, cooperative application and kernel-level scheduling system. The application-level scheduler dispatches non-preemptively, prioritizing the time-synchronous computations (timer events) over the best-effort computations (best-effort events). The kernel scheduler is preemptive and ensures fair sharing of resources across all applications. Connecting the two schedulers is a bidirectional interface called *cooperative polling* by which the application scheduler can specify its timing needs to the kernel scheduler, and the kernel scheduler can provide the timing needs of other applications (e.g., the earliest timer event of any other application) to the application scheduler.

A key aspect of cooperative polling is that the kernel expects that a time-sensitive application will yield voluntarily so that the timing needs of other time-sensitive applications can be satisfied. For example, the application is expected to yield by the next earliest timer event. Cooperative polling reduces unnecessary kernel preemption, which can cause application-level priority inversion and unpredictable timing. It also makes applications aware of the timing needs of other applications, allowing them to accommodate others,

and vice versa. Furthermore, by sharing timing information, these applications are able to more effectively adapt their behavior while preserving timeliness during overload.

Our kernel-level scheduler provides preferential treatment to the cooperative application by ensuring that its timer events are dispatched with low latency. This cooperative fair-sharing approach minimizes dispatch latency, but the scheduler still *ensures* that applications share resources fairly, i.e., uncooperative applications lose preferential treatment. Furthermore, our scheduler provides both resource-based and application-specific fairness.

Our approach is similar to work on split level scheduling such as scheduler activations [Anderson 1992], in that we provide a way for application-level and kernel-level schedulers to cooperate. However our approach differs from scheduler activations in two important ways. Firstly, the semantics of cooperative polling are much simpler, mainly because the kernel component of cooperative polling does not have any role in the creation or deletion of kernel threads (activations), while such a role is integral to the more complex semantics of scheduler activations. Secondly, whereas scheduler activations use upcalls, we piggyback timing information onto the polling operations performed by time sensitive applications. This aspect of our approach and its benefits are similar to how soft and firm timers use in-kernel polling (triggers) to provide high resolution timing without frequent interrupts [Aron 2000, Goel 2002].

We have implemented cooperative fair sharing in the Linux kernel. We have developed a video streaming application called QStream that adapts video quality based on available network, CPU and storage bandwidth [Krasic 2007; 2008]. We have modified QStream to use cooperative polling and we have retrofitted cooperative polling into the X11 server. Our evaluation shows the benefits of our approach for supporting resource-intensive, time-sensitive applications.

The following sections describe our approach in detail. Section 2 describes the application programming model, and then Section 3 presents the cooperative application-level and kernel-level scheduling system. Section 4 describes the implementation of our system, and provides an overview of some adaptive applications we have developed. Section 5 presents our evaluation, Section 6 discusses related work in the area, and Section 7 presents our conclusions.

2. User Level Programming Model

Our event-driven programming model is inspired by the principles of reactive programming [Berry 1992]. It is designed for computations that can be run non-preemptively and are short-lived. Non-preemptive scheduling avoids unpredictable timing that can be caused by preemption. It also helps avoid the use of locking and synchronization primitives required in multi-threaded programs.

```
submit(Event *e);
cancel(Event *e);
run();
stop();
```

Figure 1. Application scheduler API.

```
struct Event {
  enum { TIMER, BEST_EFFORT } type;
  Callback callback;
  TimeVal release;
  TimeVal user_virtual_time;
  ...
};
```

Figure 2. Event type definition.

Short-lived events avoid blocking or sleeping *and* run for short periods of time, helping ensure that timer events can be dispatched with low latency. Asynchronous I/O is favored to avoid blocking. Having only short-running events may sound counter intuitive, since long computations seem inherent to continuous media applications (e.g., video decompression). However, most long computations use loops, and each iteration can be divided into a separate event. Another well-known technique is to break up the execution of a long computation into smaller units through the use of coroutines [Adya 2002]. This focus on short, non-blocking events promotes an environment that allows software to quickly respond to external events, such as the arrival of data from the network, hence the name *reactive* programming.

2.1 Application Scheduler API

In our programming model, cooperative applications use a per-thread application scheduler that operates independently of application schedulers in other threads. Program execution within each thread is a sequence of events (function invocations) that are run non-preemptively.

As our actual implementation is written in C, we will illustrate our model in this section using C-like pseudo code. Figure 1 lists the key primitives in the application 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 an event to end the dispatching of events. The application can also call `cancel` to revoke an event it had previously submitted.

Figure 2 shows the type definition of an event. An application specifies each event as either a timer or a best-effort event. The `callback` field specifies the function that will handle the event and any data arguments to be passed. The `release` field specifies an absolute time value. Timer events are *not* eligible for execution until the `release` time has passed. Once eligible, timer events take priority over all best-effort events. The core application scheduler never skips or drops any timer events, including delayed events in our model. It is the responsibility of the application to

```
recv_video_frame(player, frame) {
  frame.decode_event = {
    type = BEST_EFFORT,
    user_virtual_time = decoder_get_virtual_time(frame),
    callback.fn = decode_video_frame };
  submit(frame.decode_event)
  frame.expire_event= {
     type = TIMER,
     release = decoder_get_release_time(frame),
     callback.fn = expire_video_frame };
  submit(frame.expire_event);
}
decode_video_frame(player, frame) {
   cancel(player.loop, frame.expire_event);
   if (decompress(frame) != DONE) {
     submit(frame.decode_event);
     return;
   }
   frame.display_event = {
     type = TIMER;
     release = player.start + frame.pts;
     callback.fn = display_video_frame };
   submit(frame.display_event);
}
expire_video_frame(player, frame) {
   cancel(frame.decode_event);
}
display_video_frame(player, frame) {
   put_image(player.display, frame.image);
};
```

Figure 3. Example of adaptive video player.

adapt to delayed events. The `user_virtual_time` field is explained in the next section.

2.2 Application Example

Figure 3 shows an example of how applications are written in our programming model. It presents an outline of the core logic of an adaptive video player (for a more detailed explanation, see [Krasic 2007]). The entry point in this example is the `recv_video_frame` function which is called upon arrival of compressed video frame data (e.g., from network or disk). It schedules two events for the frame, a decode and an expire event. The decode event is a best-effort event for decoding the frame. It executes the function `decode_video_frame`. In our adaptive player, video frames may arrive in order of importance, and if the CPU is busy, multiple best-effort frame decode events may be pending at the same time. The `user_virtual_time` attribute of the best-effort events is used to determine the order in which to decode frames. The functions `decoder_get_virtual_time` and `decoder_get_release_time` (not shown) use application-specific semantics to assign values of `user_virtual_time` and release, reflecting the relative importance of individual frames to perceived quality.

Less important frames may have their `expire_event` timer fire before they can be decoded, and they will be dropped. Note that this adaptation is time dependent and does not require estimating resource availability. For more important frames, the `decode_event` will occur first and decompress the frame data. Since decompression is CPU intensive, it may invoke several iterations of the decode event for full decompression. Finally, when decompression is done, a timer event is scheduled to display the image at the correct frame presentation time.

2.3 Assumptions and Limitations

The programming model above does not not require any specification of resource requirements. However, we assume that the CPU requirements of timer events alone will not saturate the system, which is likely because most computation within adaptive applications is not time sensitive. Also, cooperative applications will be designed (ideally) to adapt their best effort events. For example, in a video or audio player, the code responsible for decoding frames is the most CPU intensive and typically accounts for over 75% of the execution time. However, buffering and other adaptation methods allow flexibility as to when decoding is performed. In contrast, the code for presenting an already decoded frame onto the display or a sound device requires limited processing but has tight timing requirements [Krasic 2007]. We believe most continuous media applications such as games, graphical simulations, visualization, etc. share similar characteristics.

We also assume that events are non-preemptable within an application thread, and that programs will be written with short-lived events. The non-preemptive and short-lived computation requirements match well with event-based applications. For example, GUI systems generally use short event handlers for responsiveness. We believe that it should be possible to modify existing event-driven applications to use cooperative polling without significant restructuring, as we show later for the X11 server. It should be possible to use non-preemptive thread libraries such as Pth [Engelschall 2006] to implement our model. Another alternative is the TAME system [Krohn 2007] that offers the programmability advantages of threads.

3. Kernel Level

Our overall scheduling approach combines application-level event scheduling, described in the previous section, with a kernel scheduler that we call a cooperative fair-share scheduler. Connecting the two schedulers is a new `coop_poll` system call that enables inter-thread cooperation by sharing an thread's timing and progress information with the kernel and with other threads, the result being improved timing response. In this section, we describe `coop_poll` and then present our kernel scheduler.

3.1 The `coop_poll` System Call

The `coop_poll` call voluntarily yields the CPU and communicates timing and fairness information to the kernel as shown below:

```
coop_poll(Group gid, Event *timer, Event *best_effort);
```

This call takes a thread group parameter and two event parameters. A thread group in our system consists of threads that share resources cooperatively. The values of the event parameters specify 1) the timer event with the earliest release time, and 2) the best-effort event with the smallest `user_virtual_time` in the current thread to the kernel scheduler. We call these values the thread's release time and importance. These event values are used to wake up the thread at its next release time, or when its best-effort event is most important among all threads within its thread group.

When `coop_poll` returns, the kernel sets the release time in the timer event to the earliest release time among timer events, submitted via `coop_poll`, across *all* other threads. Similarly, the `user_virtual_time` value in the best_effort event is set to the least user-virtual time among the best-effort events of other threads within the *same* thread group. Our kernel expects that threads will cooperate by calling `coop_poll` in the future according to these return values. In exchange for this cooperation, the kernel scheduler gives the thread preferential treatment in the form of protection from preemption and accurate timer dispatch, as described later in Section 3.2. Thus these event parameters represent a *quid-pro-quo* quality of service agreement. We define threads as being *cooperative* when they use the `coop_poll` system call and adhere to cooperative behavior described above.

Our fair-share scheduler provides resource-centric fairness across thread groups and application-specific fairness within a thread group. Threads within a group pool their resources and then subdivide them according to application-specific policy as expressed by the `user_virtual_time` value of the `best_effort` event parameter. The kernel selects the thread with the least `user_virtual_time` within a thread group. For example, a video application can set this value to the number of frames processed, denoting video quality. Then two video threads running in the same thread group (e.g., multi-party video conferencing) would have the same frame rate (video quality) even though the CPU is allocated differently to the threads. If the threads in a group are from heterogeneous applications, they must adopt a commonly understandable notion of `user_virtual_time`, e.g., based on cumulative measures of utility. Thread groups can be used to schedule the threads of a single application, cooperating adaptive applications, or all applications of a user.

3.2 Kernel Scheduler

We have designed a kernel scheduler that aims to provide better timeliness and fairness than current best-effort schedulers by taking advantage of the cooperative polling model. Our kernel scheduler uses a variant of weighted fair-queuing (WFQ) to accomplish fair sharing. Below, we provide an overview of this algorithm before describing our cooperative fair-share scheduler.

Fair-Share Scheduling Our fair-share scheduler uses the notion of virtual time. As each thread executes, our scheduler updates the virtual time, in weighted proportion to the actual running time of the thread. As shown in Figure 4, the scheduler uses a fair-share run queue sorted by minimum virtual time, with threads in a group sharing their virtual time. The run queue keeps track of minimum virtual time.When a thread is added to the run queue, its virtual time is set to at least the minimum virtual time. This happens when a thread is first created, and also when a thread wakes up. This adjustment ensures that new threads and threads that sleep cannot accumulate CPU allocation that would subsequently allow them to starve other threads. This "use it or lose it" approach helps accommodate the sporadic needs of IO-bound threads. Standard integer arithmetic techniques help ensure that virtual time wrap-around works correctly.

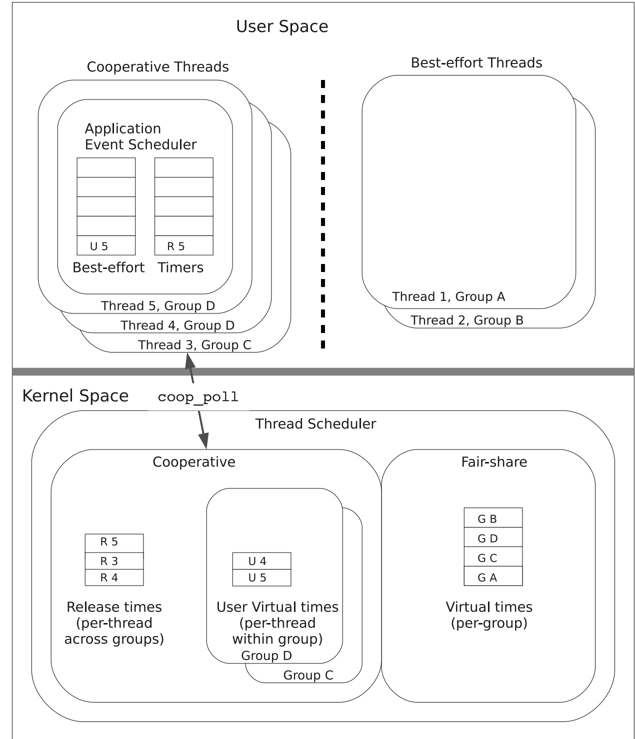


Figure 4. Hierarchical architecture of cooperative polling.

The scheduler also computes how long the next thread should run, i.e., the timeslice of the thread, as $ts = period/N$. The *period* is a global scheduling parameter (a typical value is 20 ms), and N is the number of runnable threads. Each thread gets a chance to run every period, and hence smaller values improve short-term fairness. However, the scheduler enforces a minimum timeslice (e.g., 100 us) to prevent livelock. Note that the period parameter does not affect the timing response of the cooperative scheduler described below.

Cooperative Fair-Share Scheduling The cooperative fair-share scheduler provides enhanced timing support by combining fair sharing with information available from thread release times. A thread issuing the `coop_poll` system call is inserted into a release queue, sorted by earliest release time,

as shown in Figure 4. When there are cooperative threads in the release queue, either runnable or blocked, the scheduler uses the earliest release time among all the cooperative threads to adjust the timeslice of the thread that is chosen to run. There are two cases depending on whether the earliest release time is in the future or in the past. When the release time is in the future, the scheduler selects the thread with the smallest virtual time, and sets its timeslice $ts = min(release - now, period/N)$. Otherwise, the scheduler selects the thread with the earliest release, and sets its timeslice $ts = 0$. On exit from `coop_poll`, the release time of the timer parameter will be set based on the timeslice, informing the application when it should next yield. Not visible to the application, the kernel extends the true timeslice by a grace period during which the application has the chance to yield before being preempted (also to ensure the minimum timeslice), this will be explained further in the next section.

In the first case, when $(release - now) > period/N$, the scheduler uses the fair-share scheduling algorithm described earlier. Otherwise, it uses earliest-release scheduling because the thread is expected to execute until the next release time. In this case, the earliest cooperative thread runs with its timeslice set to zero, allowing it to run for a minimal amount of time. As a result, the application scheduler of a cooperative thread will execute timer events with release time in the recent past, and then it will yield back to the kernel immediately (since its timeslice is 0) via `coop_poll`. This is the expected behavior of cooperative threads.

Policing Misbehaving Threads Cooperative threads receive preferential treatment because they are scheduled soon after their release times are due (earliest-release scheduling), but our scheduler includes a *policing* mechanism to ensure that they do not gain long-term advantage by misusing `coop_poll`. Specifically, our policing mechanism demotes a cooperative thread to a best-effort thread so that the thread is subject to fair sharing exactly as other best-effort threads. Unlike cooperative threads, the kernel does not maintain release times for best-effort threads, so they are unable to run at specific times. Policing is temporary and threads regain cooperative status each time they call `coop_poll`. The scheduler performs policing for three reasons described below: 1) running beyond timeslice, 2) non-cooperative sleep, and 3) exceeding a virtual time threshold.

When a cooperative thread is selected to run (i.e., it had called `coop_poll` when it blocked), a kernel timer is scheduled after the timeslice plus a grace period. This period is a scheduling parameter called `coop_slack` (1ms by default). Cooperative threads are expected to yield soon after their timeslice deadline, but if they fail to do so within the slack period, they are preempted and demoted. Second, based on our programming model, cooperative threads are normally expected to sleep by calling `coop_poll`. However, if a thread blocks in the kernel for any other reason, then it will not have a release time and is demoted. In regards to overhead, we note that there is only a single instance of the above timer required (per cpu), for the currently running task. In Section 5.6 we report that overall overhead of our scheduler is small, hence we believe the overhead of this timer is negligible.

As a final defense against misbehavior, the kernel uses the virtual time of the thread to ensure fairness. Recall that `coop_poll` inserts the thread's release time into the release queue. This insertion is not done if the thread's virtual time exceeds the run queue's minimum virtual time by more than a certain threshold. A thread issuing many timer events in a short period can increase its CPU share in the short term, but this will cause its virtual time to advance faster than other threads. The threshold (order of ms) ensures that cumulative unfairness is bounded in the long term. A malicious application can employ a large numbers of threads but higher-level containment mechanisms are more appropriate for defending against such attacks.

Scheduler Usage The weights of the fair-share scheduler are adjustable via the standard POSIX `nice` system call. Adaptive applications will be able to maintain their timing even in overload, but the weight can be used to influence quality. For non-adaptive cooperative applications, the weight can be used to compensate if the normal fair share is not sufficient. Since our scheduler is fully work-conserving, weights for non-adaptive applications can be set conservatively without risking starvation of other applications.

4. Implementation

Our core implementation consists of an application-level and a kernel-level scheduler. The application scheduler implements the API shown in Figure 1 and calls the `coop_poll` system call to invoke the kernel scheduler. We briefly describe our implementation of the kernel scheduler and two applications, the QStream video application and the X11 server, that we have modified to support cooperative polling. We will use these applications to evaluate our approach.

4.1 Kernel Scheduler

Our current prototype scheduler is a based on the official Linux kernel repository (version 2.6.26.5). Several recent infrastructural improvements in Linux help the performance or the implementation of our scheduler. These include a preemptable kernel, high-resolution timers and process accounting, tickless idle mode, and the replacement of a priority scheduler with the CFS (Completely Fair) Scheduler.

In prior work [Goel 2002], we described the benefits of a preemptable kernel on event dispatch latency. With low-latency kernel preemption, the dominant remaining component of event dispatch latency is scheduling latency. The contributions of this paper are targeted directly at scheduling latency. We use high-resolution timers as a straightforward way to enforce timeslices, although CFS still relies on a periodic timer 'tick' to decide when to end the current timeslice.

High-resolution process accounting simplifies virtual time computations.

Our implementation does have multiprocessor support, but we do not evaluate multiprocessor operation or workloads in this paper. We plan to do so in future work. Like CFS, our scheduler maintains a run queue per processor. In our case, each run queue contains per-processor instances of fair-share virtual time, release time, and best-effort user virtual time queues. Non `coop_poll` tasks are load balanced in the same way as CFS. For tasks using `coop_poll`, we disable thread migration—i.e. we currently assume that `coop_poll` applications are multiprocessor aware, and they take responsibility to distribute their work among their (CPU pinned) threads as they see fit. We have implemented such a scheme in QStream to utilize multiprocessors for video processing. We are also exploring alternative methods for supporting event-driven programs on multiprocessors [Zeldovich 2003].

The CFS scheduler in Linux arrived concurrently with the development of our own scheduler. Our kernel scheduler is similar to CFS, and it should be possible to re-target the `coop_poll` implementation to CFS.

4.2 The QStream Video Application

As part of our broader work, we have implemented a complete adaptive video streaming system called QStream [Krasic 2007]. QStream is a substantial application consisting of over 100,000 lines of code, mostly written in C. QStream uses scalable video compression along with a resource adaptation algorithm called Priority-Progress to adapt to available network bandwidth, storage throughput, and CPU availability. QStream includes a `libqsf` event library that implements our application-level scheduler. QStream uses the event API shown in Figure 1, and it required no changes except relinking with a modified `libqsf` to use cooperative polling.

4.3 X11 Server

The X.Org X11 server (X server) forms the core graphical interface for most Unix based operating systems, and is thus crucial for time-sensitive Unix applications requiring real-time visualization. The X11 architecture uses a socket to communicate between applications and the X server. Socket communication is subject to scheduling delays, so application requests are handled in a best-effort manner.

The X server is an event-driven application [Packard 2000]. It provides an extension called X Synchronization [Glauert 1991] that allows applications to communicate their timing needs to the server. The main primitive, `XSyncAwait`, provided by this extension allows specifying a wait condition and provides a barrier-like functionality. Any X11 requests made by an application subsequent to calling `XSyncAwait` are deferred until the condition becomes true. Although this extension is present in X11, its implementation does not include high resolution timers. We modified the X11 server to use `coop_poll` inside its event loop, and enabled precise timer conditions in the synchronization extension. Our modifications consist of 917 lines out of a total of 236,221 lines in X11 (approximately 0.3%). We also modified the QStream video player to call `XSyncAwait` before each call to XPutImage to communicate the desired display time for each video frame.

5. Evaluation

In this section, we evaluate the timeliness, fairness and overhead of our cooperative polling model. The workload in our experiments consists of a varying mix of best-effort and time-sensitive applications. Each workload is run for 5 minutes with some additional time for startup and tear down.

We use a kernel compilation job to represent an intensive best-effort application. We set the number of best-effort threads in the system by parallelizing the kernel build process, using the `make -j` parameter. We use two time-sensitive applications, the QStream video streaming player (adaptive) and the X11 server (non-adaptive). Similar to best-effort threads, the adaptive workloads can fully saturate the system, utilizing 100% CPU. We control the number of adaptive threads by streaming different videos concurrently. Each video has a highly variable bit rate and variable processing requirements. This adaptive application is an extremely challenging target for the scheduler. Our results show that the bursty nature of these video processes exhibits pathological behavior for any heuristic that attempts to predict future requirements based on the recent past. We also performed experiments that show the benefits of incorporating cooperative polling into the X11 server.

We compare our kernel scheduler against the Completely Fair (CFS) and the real-time priority Linux schedulers. We use CFS in its “low-latency desktop mode” configuration.

We evaluate timeliness by measuring tardiness (or dispatch latency), the difference between the requested release time and the time when a timer event is dispatched. This value is obtained by instrumenting the application schedulers in QStream and X11. We evaluate fairness by comparing CPU allocation across threads, and we evaluate overhead by measuring video throughput in terms of its frame rate.

The setup for all the experiments consists of three PCs connected by a LAN, two as video servers, and the other as the client desktop. The two servers load balance the server work. All the measurements are taken on the desktop, and in all cases, we ensure that the servers and the LAN do not limit desktop performance. The desktop machine is a generic desktop PC with a 3.0 GHz Intel Pentium 4 (1MB L2 cache), 1 GB of RAM, NVidia NV43 GPU (driver version 173.08 beta), running the Ubuntu 7.10 Linux distribution.

5.1 Timely Response - Baseline

The workload for this experiment consists of a single adaptive thread sharing the processor with best-effort threads. This experiment provides a useful baseline for the perfor-

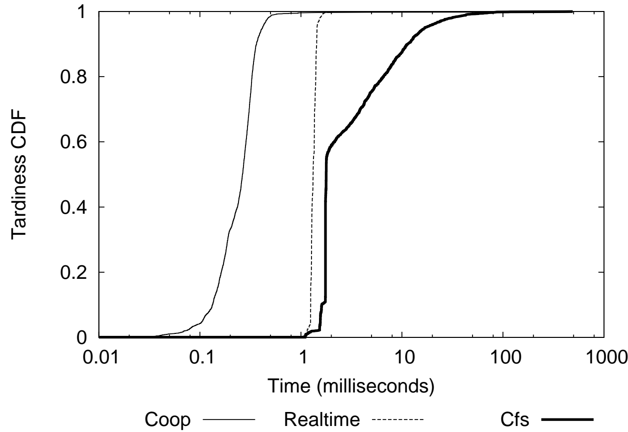


Figure 5. Cumulative distribution of tardiness with a single adaptive thread.

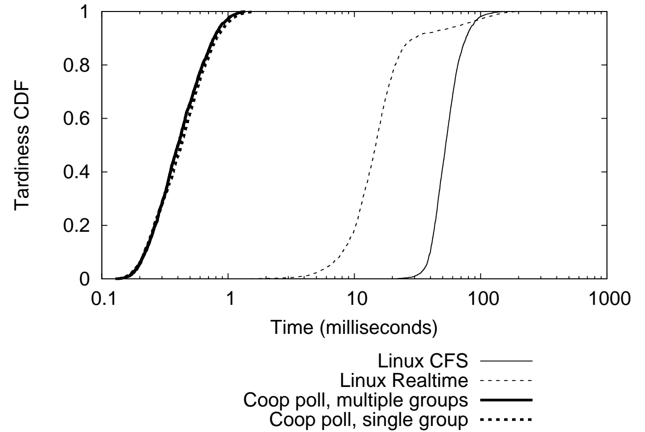


Figure 6. Tardiness with multiple adaptive threads.

mance that can be expected with multiple adaptive applications. We use four concurrent invocations of gcc for the kernel build workload.

Figure 5 shows the cumulative distribution of tardiness in QStream with each of the schedulers. The worst case tardiness occurs when the lines reach the top of the y-axis. The shape of the lines gives an indication of the distribution of the tardiness of the events, with lines to the left indicating lower tardiness. This figure shows two main results. First, the Linux real-time scheduler and our cooperative scheduler have similar worst case tardiness, on the order of 1ms. This number represents close to optimal tardiness, given the granularity of events within our QStream implementation. In particular, we have verified that the worst-case execution times for best-effort events in QStream are in the under 1 ms range. Note that the real-time scheduler has poorer average tardiness because, in the absence of `coop_poll()`, QStream uses the `poll()` system call which does not use high-resolution timers in Linux. Otherwise, we expect the numbers would be similar.

The second result is that the Linux CFS scheduler fares poorly in terms of tardiness. Many events are dispatched over 20 ms after their release. This tardiness is unacceptable for video conferencing and various audio application, specially echo cancellation [Valin 2007]. Without explicit information from applications, conventional schedulers use heuristics to categorize applications as interactive, and favor them in terms of tardiness. This experiment shows that when best-effort threads drive a system into overload, the scheduler interactivity heuristics tend to fail, and applications with timing constraints do not work well.

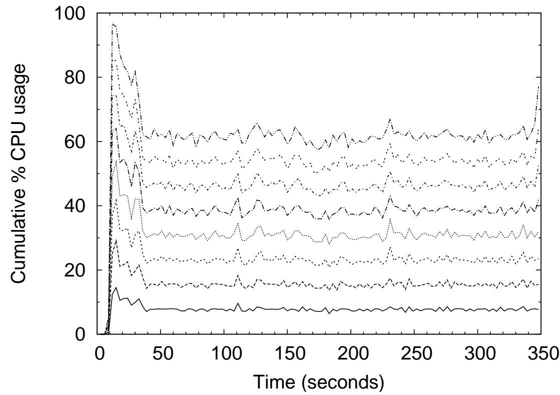
5.2 Timely Response - Complex Workload

In this experiment, we employ multiple adaptive threads in our workload. We use a mix of 8 adaptive threads, and 4 best-effort threads. Roughly 7 adaptive threads saturate the processor, so this experiment runs at 100% CPU load. We are not aware of any real-time systems that attempt to provide tight timing at 100% capacity while maintaining fairness. We run the cooperative scheduler in two scenarios, threads in the same thread group, and threads in separate thread groups (see Section 3.1). In the single-group experiment, the adaptive threads aggregate their CPU allocations and use the best-effort event `user_virtual_time` parameter of `coop_poll` to synchronize frame rates. In the multiple-group experiment, only the timer event information is shared, and the kernel allocates CPU fairly.

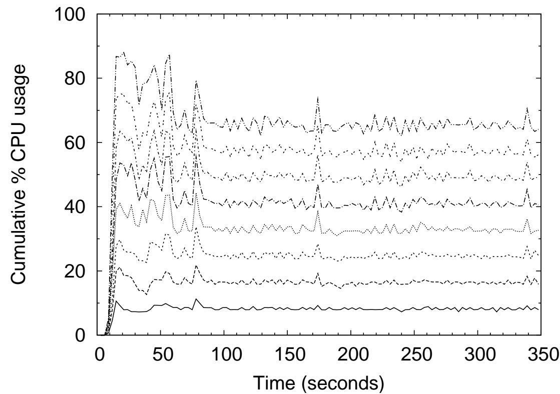
Figure 6 shows the cumulative distribution of tardiness in QStream with the three schedulers. Both the single and multiple group cases of the cooperative scheduler achieve comparable tardiness, in the 1ms range. With Linux real-time priorities, all the adaptive threads run with equal priority in the round-robin real-time class. As compared to the previous experiment, both the CFS and the real-time cases have much worse performance. With CFS, almost all events are dispatched with over 20 ms tardiness, and with Linux real-time, the tardiness values are now well above 10 ms. The worst-case performance of both the schedulers is over 100 ms. With multiple threads, the real-time scheduler uses timeslices that are too coarse grained.

5.3 Fairness

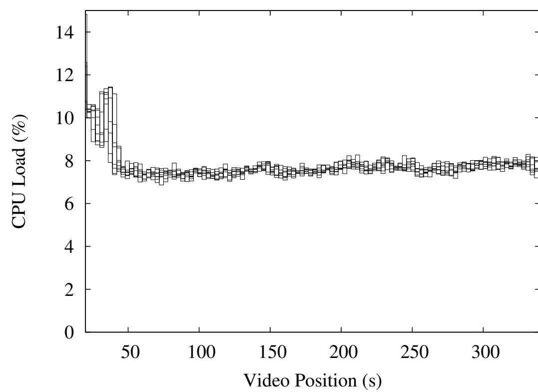
Figure 7 shows fairness results by plotting the CPU usage of the 8 adaptive threads for two cases: CFS and cooperative fair share with threads in separate groups. Figures 7 a) and b) show the sum of CPU usages per adatpive thread, while Figures 7 c) and d) show the CPU usages individually. The kernel build jobs run several short compilations, and we do not plot their CPU usage directly. However, the overall usage of these best-effort threads is the time remaining in the Figures 7 a) and b), since the total CPU usage is 100% in all these experiments. Since there are 12 running threads, the ideal result is an allocation of 8.3% per process and 66.6% to the 8 adaptive threads.



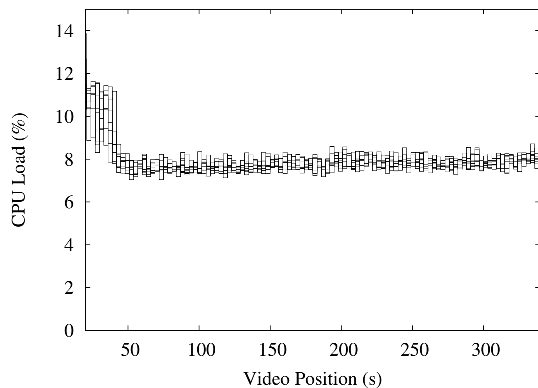
(a) Linux CFS



(b) Coop Poll, Multiple Groups

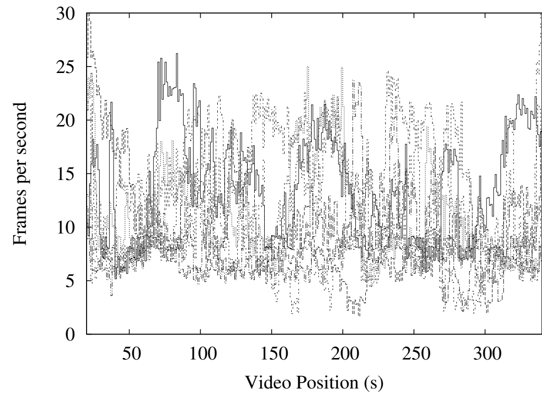


(c) Linux CFS

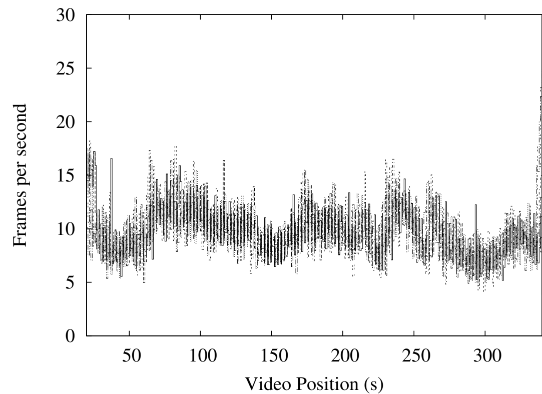


(d) Coop Poll, Multiple Groups

Figure 7. CPU fairness with multiple adaptive threads.



(a) Multiple Groups



(b) Single Group

Figure 8. Video quality in frames per second.

To help quantify the data in the graphs, we compute according to a resonably popular metric known as the Jain fairness index [Jain 1984], which is defined by the following equation: $fairness = \frac{(\sum x_i)^2}{(n \sum X_i^2)}$. This index ranges from $1/n$ (worst case) to 1 (best case). The Linux CFS scheduler and our scheduler provide comparable fairness to adaptive threads, the Jain index for both cases in Figures 7 is 0.98. However, as can be seen comparing the top areas of Figures 7 a) and b), our scheduler provides slightly less allocation to best-effort threads because these threads lose allocation when they sleep during I/O operations. This difference is most pronounced during the first phase of the experiment, when the kernel build is highly I/O intensive (make build and make dep stages). Later, the kernel build becomes CPU intensive spending most of its time in gcc.

The Linux real-time scheduler starves the gcc processes, as expected (figure not shown). Also the allocation between real-time threads is highly uneven, because the round-robin algorithm does not track CPU usage tightly.

Application-Specific Fairness Although the fairshare schedulers in the previous section provide uniform resource fairness, they do not provide uniform application fairness.

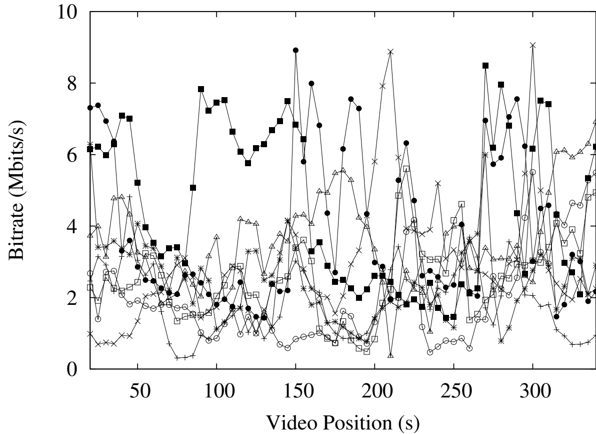


Figure 9. Video Bitrates.

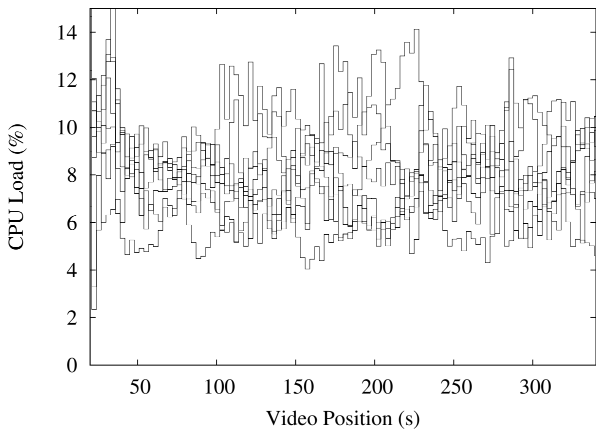


Figure 10. CPU fairness with multiple adaptive threads in a single thread group.

Figures 8a) and 8b) show the video frame rate when the adaptive threads run in different thread (Jain index 0.86) groups or the same thread group (Jain index 0.97). Notice the tight grouping of frame rates in Figure 8b). Recall, adaptive threads may specify application-level virtual time to coop_poll, allowing them to subdivide their combined allocation according to application specific criteria. As a result, the single thread group here achieves extremely fair quality (frame rates), in contrast to fair CPU allocation that delivers highly unfair video quality. Note that each video plays a different clip, Figure 9 shows the bitrates of each of them. Such highly non-uniform bitrates are suggestive of the correspondingly bursty relationship between video quality and CPU requirement of each thread. Hence the variation in the frame rate in Figure 8a) is not unexpected.

A smooth CPU rate translates to a bursty video quality, and vice versa. Confirming the latter, Figure 10 shows the individual CPU allocations (Jain index 0.92) of the adaptive threads corresponding to Figure 8b) . We believe that a purely kernel scheduling solution, lacking the application specific information, *cannot* achieve the level of application fairness shown in Figures 8b).

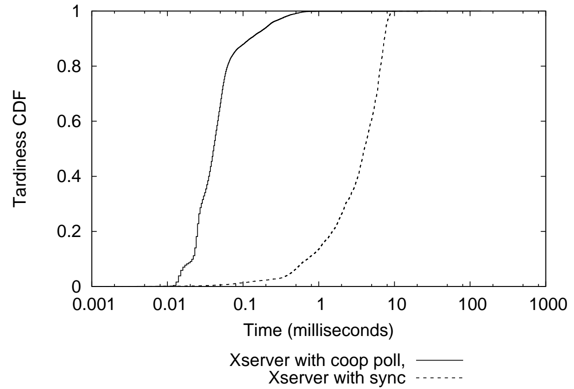


Figure 11. Tardiness of X11.

5.4 Cooperative but Non-Adaptive

The experiments of Sections 5.1 and 5.2 measured the tardiness of the QStream player. In those experiments, we had disabled frame display to the X11 server, to isolate any effects due to X. In this section, we enable frame display and measure the performance of the X11 server. The workload in these experiments is the same as in the previous sections (8 adaptive threads and `make -j4` kernel build). We only use our cooperative scheduler to run this experiment. We run two experiments that compare the performance of the X11 server with and without cooperative polling. The first experiment uses the X11 synchronization extension (see Section 4.3) to measure frame display tardiness, which is not possible with an unmodified server because it does not know about display release times. In the second experiment, the synchronization information is used by the X11 server to supply timer events to the kernel via coop_poll, and to measure tardiness.

Figure 11 shows the cumulative distribution of tardiness of frame display events in X11. When using cooperative polling, the X11 server has tardiness in the 1 ms range, even under the heavy load imposed by this experiment. Without cooperative polling, the tight timing achieved by the QStream adaptive threads (see Figures 5 and 6) is lost due to unpredictable timing in the X11 server. Note that this experiment uses the synchronization extension that is designed to meet the timing requirements of frame display, but the X11 server is unable to run in time due to CPU load, and hence the increased tardiness. We should emphasize that the adaptive threads use cooperative polling, which indirectly helps X11 even when it does not use cooperative polling. The X11 tardiness would be close to the Linux CFS CDF shown in Figure 6 if the adaptive threads were running under CFS.

5.5 When Cooperative Tasks Misbehave

Section 3.2 described how our scheduler polices coop_poll threads. In this section, we evaluate our scheduler with a

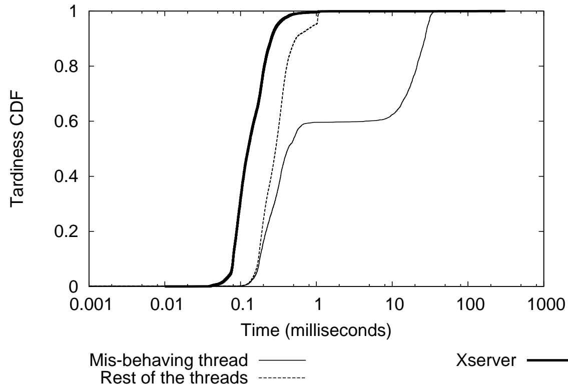


Figure 12. Tardiness with a misbehaving cooperative task.

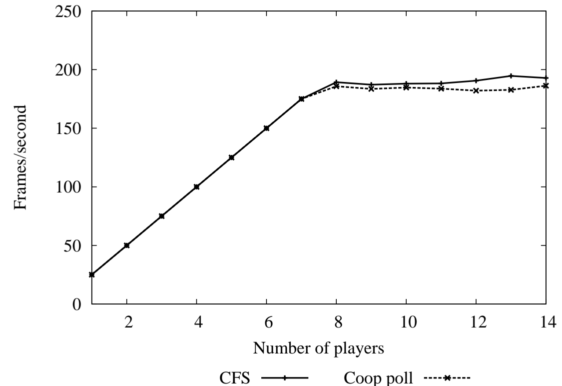


Figure 13. Overhead of Cooperative Polling vs. Linux CFS.

misbehaving application, by modifying a QStream application that deliberately delays its yields. The delay is randomly chosen from the range of 0-10 ms, and the probability of delay is configurable. Aside from one misbehaving player, which delays yielding with a probability of 0.01, this workload is the same as in the previous section. Figure 12 shows that the tardiness of only the misbehaving player is affected. It gets policed repeatedly causing it to have high tardiness up to 40 ms, while the rest of the players and the Xserver are unaffected.This experiment shows that `coop_poll` not only provides better timing to cooperative applications, but also does not harm them when others are uncooperative!

5.6 Scheduler Overhead

In this section, we describe micro- and macro-benchmark experiments that measure the performance overhead of our cooperative fair-share scheduler compared to the Linux CFS scheduler. We ran the context-switch latency micro-benchmark from the lmbench benchmark suite [McVoy 1996], and the results for the two schedulers are the same.

We also ran a macro-benchmark experiment comparing cooperative fair sharing with CFS. The workload in these experiments consists only of adaptive threads, from 1 to 14 videos. This range helps evaluate our scheduler's overhead across a range of workloads from under-load to heavy overload. In this experiment, we use the same video in all the threads. This video is specially prepared with a uniform bit rate-quality relationship (I frame only, fixed quality, content with no movement), so that the video frame rate provides a good estimate of the scheduler overhead.

Figure 13 shows the average as the number of videos is increased. Since 7 videos saturate the processor, both schedulers have the same frame rate until 7 videos. Beyond that, the throughput of our scheduler is lower due to increasing context switching. The actual slowdown is modest (6% at 14 players). Overall, our cooperative scheduler is competitive with CFS, both in fairness and throughput, yet we achieve as much as two orders of magnitude improvement in timing.

5.7 Limits of Periodic Scheduling

We believe that cooperative polling provides tight timing with low overhead because it uses the direct timing information provided by threads to inform its scheduling decisions. We compared our approach with a purely periodic approach in which the scheduler simply gives a short timeslice to each thread, simulating round-robin or the period parameter of reservation based scheduling.

We performed this experiment with 8 adaptive applications, and we progressively decreased the global period (described in Section 3.2) of our scheduler, but the applications did not use cooperative polling. We summarize our results here. We were able to achieve a worst-case tardiness of 5 ms, and a context switch rate of 9348/second, when the global period is 1 ms. Decreasing the period further had no benefits on tardiness. With cooperative polling, the equivalent workload has a worst-case tardiness of 1ms and a context switch rate of 2211/second. Thus, cooperative polling achieves a 5x improvement in tardiness with a 4x reduction in context-switch rate relative to periodic scheduling methods.

6. Related Work

The SMART scheduler [Nieh 1997] and the borrowed virtual time (BVT) scheduler [Duda 1999] share our goal of providing support for time-sensitive and best-effort applications. Both use virtual-time based fair sharing across all applications, similar to our scheduler. However, SMART uses reservations for real-time threads, requiring estimation of resource requirements. BVT uses a user-specified *warp* value to temporarily bias the virtual time of time-sensitive applications, but the warp value is not directly related to the application's timing needs.

Our approach would benefit from the integrated support for CPU, memory and disk scheduling in the Redline system [Yang 2008]. However, Redline requires specifying reservations and estimating resources, and its use of reservations limits timing accuracy that can be achieved without introducing significant overhead.

Our model is closely related to split level scheduling [Govindan 1992] in which threading is implemented via hybrid user and kernel-level scheduling. The aim is to correctly prioritize user-level threads in different address spaces while minimizing user/kernel interactions. Scheduler activations [Anderson 1992] aim to limit preemption by informing the user level about the scheduling decisions made by the kernel. However, activations use kernel upcalls to communicate the kernel's scheduling decision to the application, while our model uses application-level polling to synchronize with the kernel's scheduling decisions. Additionally, our applications also inform the kernel about their timing needs or priorities.

The reflective scheduler by Ruocco [Ruocco 2006] uses an event system that allows application-specific adaptation for real-time applications. However, this scheduler requires all events to have deadlines associated with them, gives higher priority to real-time applications, and unlike our scheduler is non-work conserving. Similarly, the TMO model [Jenks 2007] uses time-triggered events that have higher priority than message triggered events, but TMO threads can starve best-effort threads.

While cooperative polling is used by applications, soft timers [Aron 2000] use kernel-level polling at key trigger points, such as kernel entry to efficiently schedule soft-timer events (e.g., packet transmission) internal to the kernel. Both approaches aim to avoid unnecessary preemption or interrupts.

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 network and CPU availability [Krasic 2007]. This technique was inspired by other works on quality-adaptive streaming [Rejaie 1999].

7. Conclusions

Our QStream video streaming application supports mobile devices, high-quality media streaming, and multi-party conferencing, all of which can saturate resources and thus require adaptation. We found that the combination of tight timing and synchronization constraints, heavy resource demands and adaptation is challenging for current operating systems. These limitations led to the design and implementation of the cooperative polling model that exposes timing constraints so that time-sensitive applications adapt more effectively while still preserving timeliness during overload. Our results show that our approach can provide worst-case timing of roughly 1ms to user-level applications even under heavy load. We believe this is a boundary between what general purpose and specialized real-time systems can support.

At the same time, a standing issue in developing schedulers for time-sensitive applications is the need to ensure fairness with best-effort applications. This work shows that cooperative polling can be integrated with fair share scheduling, providing the benefits of both.

We see several avenues of future work. We are currently evaluating cooperative workloads on multi-core processors, and integration of cooperative polling in the Linux CFS scheduler. We are interested in exploring the use of cooperative polling for user-level device drivers [Williams 2008] that need tight timing for correctness. Another direction would be incorporating cooperative polling as a hypercall to improve scheduling in virtual machine monitors. Finally, we note that our QStream application and the user- and kernel-level cooperative polling framework is open source software and is available at `http://qstream.org`.

References

[Adya 2002] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *Proc. of the USENIX Technical Conference*, June 2002.

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

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

[Berry 1992] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. ISSN 0167-6423.

[Corbato 1962] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley. An experimental time-sharing system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 335–344, 1962.

[Duda 1999] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. of the SOSP*, pages 261–276, 1999.

[Engelschall 2006] Ralf S. Engelschall. The GNU Portable Threads. http://www.gnu.org/software/pth/, 2006.

[Glauert 1991] Tim Glauert, Dave Carver, Jim Gettys, and David P. Wiggins. *X Synchronization Extension Library Version 3.0*. X Consortium Standard, 1991. `http://www.xfree86.org/current/synclib.pdf`.

[Goel 2002] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on a commodity OS. In *Proc. of the OSDI*, pages 165–180, December 2002.

[Govindan 1992] Ramesh Govindan and David P. Anderson. Scheduling and IPC mechanisms for continuous media. In *Proc. of the SOSP*, pages 68–80, October 1992.

[Jain 1984] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301,

DEC Research, September 1984. URL http://www.cse.wustl.edu/~jain/papers/fairness.htm.

[Jenks 2007] Stephen F. Jenks, Kane Kim, and et al. A middleware model supporting time-triggered message-triggered objects for standard linux systems. *Real-Time Systems*, 36(1-2):75–99, 2007.

[Jones 1997] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proc. of the SOSP*, pages 198–211, October 1997.

[Krasic 2008] Charles Krasic and Jéan Sebastian Légaré. Interactivity and scalability enhancements for quality-adaptive streaming. In *Proc. of the ACM Multimedia*, Vancouver, 2008.

[Krasic 2007] Charles Krasic, Anirban Sinha, and Lowell Kirsh. Priority-progress CPU adaptation for elastic real-time applications. In *Proc. of the Multimedia Computing and Networking Conference (MMCN)*, January 2007.

[Krohn 2007] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proc. of the USENIX Technical Conference*, pages 87–100, 2007.

[Leslie 1996] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[Lu 2000] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proc. of the RTSS*, December 2000.

[McVoy 1996] Larry W. McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996. URL citeseer.ist.psu.edu/mcvoy96lmbench.html.

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

[Nieh 1997] Jason Nieh and Monica Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proc. of the SOSP*, pages 184–197, October 1997.

[Packard 2000] Keith Packard. Efficiently scheduling x clients. In *In Proceedings of the Freenix Track of the USENIX Annual Technical Conference*, pages 44–44, Berkeley, CA, USA, 2000. USENIX Association.

[Rejaie 1999] Reza Rejaie, Mark Handley, and Deborah Estrin. Quality adaptation for congestion controlled video playback over the Internet. In *Proc. of the ACM SIGCOMM*, pages 189–200, October 1999.

[Ruocco 2006] Sergio Ruocco. User-level fine-grained adaptive real-time scheduling via temporal reflection. In *Proc. of the RTSS*, pages 246–256, 2006.

[Schwarz 2007] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h.264/avc standard. *IEEE Trans. Circuits Syst. Video Techn.*, 17(9):1103–1120, 2007.

[Steere 1999] David Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proc. of the OSDI*, pages 145–158, February 1999.

[Valin 2007] Jean-Marc Valin. On adjusting the learning rate in frequency domain echo cancellation with double-talk. *IEEE Transactions on Audio,Speech, and Language Processing*, 15(3):1030–1034, March 2007.

[Williams 2008] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gun Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of the OSDI*, 2008.

[Yang 2008] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proc. of the OSDI*, 2008.

[Yang 2007] Zhenyu Yang, Wanmin Wu, Klara Nahrstedt, Gregorij Kurillo, and Ruzena Bajcsy. Viewcast: view dissemination and management for multi-party 3d tele-immersive environments. In *Proc. of the MULTIMEDIA*, pages 882–891, 2007.

[Zeldovich 2003] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert T. Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proc. of the USENIX Technical Conference*, pages 239–252, June 2003.