Support for Time-Sensitive Applications via Cooperative Polling

by

Mayukh Saubhasik 2008

Bachelor of Technology, Birla Institute of Technology and Science Pilani, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

September 2008
© Mayukh Saubhasik 2008
Abstract

Time sensitive applications like media players/editors and games are increasingly being deployed on a variety of computing platforms with varying processing power, screen size, etc. Due to varying availability of resources the application has to adapt itself in order to meet its timing constraints. For example a video player might drop frames or resize them depending on the available Central Processing Unit (CPU) and screen size. Therefore these applications are both CPU intensive and time sensitive. Existing systems are incapable of dealing with applications with both these requirements. Most solutions either require an estimation of CPU usage (not possible for adaptive applications) or they suffer from starvation problems. We present a system which consists of an event driven way of structuring time sensitive applications and a kernel scheduler which helps the applications meet their timing constraints. Our approach, called 'cooperative polling', enables the applications to share timing information with each other and the kernel in order to meet their timing requirements, while still maintaining long term fairness. Our system is also capable of dealing with timing requirements which arise indirectly (not specified by the application) via Input Output (I/O), etc. As part of our evaluation we modified an adaptive video player application and the display subsystem for Linux to use our cooperative polling approach. We also extended the display server to implement a mechanism by which clients can convey their timing requirements to the server. Our evaluations show that this approach achieves event dispatch latency two orders of magnitude lower than existing schedulers, while still maintaining overall fairness and low overhead. We also show that the programming effort needed to convert an existing event based application to use our approach is quite trivial.
Table of Contents

Abstract .......................................................... ii
Table of Contents ................................................. iii
List of Tables ..................................................... vi
List of Figures ..................................................... vii
Glossary ............................................................. ix
Acknowledgments ................................................... x

1 Introduction ....................................................... 1
  1.1 Motivation .................................................. 1
  1.2 Related Works ............................................. 2
  1.3 Contributions .............................................. 4
  1.4 Thesis Structure .......................................... 4

2 System Design and Algorithms ................................. 6
  2.1 Requirements .............................................. 6
  2.2 Programming Model ..................................... 7
    2.2.1 Application Scheduler API ......................... 8
    2.2.2 Kernel Scheduler API ............................... 9
  2.3 Cooperative Polling ..................................... 10
    2.3.1 Application Scheduler ............................. 10
    2.3.2 Kernel Scheduler .................................. 13
  2.4 Combining coop.poll with select ...................... 19
5.6.3 Overhead for the Modified Xserver .................................. 57
5.6.4 Code Change Overhead ................................................. 58
5.7 Chapter Summary ......................................................... 58

6 Future Work and Conclusion .............................................. 60
6.1 Future Work ............................................................. 60
  6.1.1 Incorporating coop_poll onto the Hypervisor ................. 60
  6.1.2 Using STM for Implementing Multi-core QStream .......... 61
  6.1.3 Support for Scheduler Groups/Cgroups ....................... 62
  6.1.4 Passive Load Balancing ........................................... 62
  6.1.5 Combining coop_poll with epoll ............................... 62
6.2 Conclusion ..................................................................... 62

Bibliography ...................................................................... 64

A Pseudocode ..................................................................... 67

B Linux Kernel Programming .............................................. 72
  B.1 Debugging Techniques .................................................. 72
  B.2 System Call Constraints .............................................. 74
  B.3 The Need for Precise Task Accounting ......................... 74

C Notes on Publication ....................................................... 76
List of Tables

5.1 Nanosleep accuracy .............................................. 52
5.2 Xserver modification: CPU overhead .......................... 57
5.3 Xserver modification: Memory overhead ..................... 57
5.4 LOC count for Xserver modifications(Including comments and log messages) ........................................ 58
List of Figures

2.1 Application scheduler API .......................... 8
2.2 Event type definition ............................... 9
2.3 Application-level event scheduler ................. 11
2.4 Coop POLL Interface .............................. 13
2.5 Using the coop.poll system call for inter-application cooperation .......................... 14
2.6 State diagram for the task states ................. 21
2.7 coop realtime task timeline ......................... 23

3.1 Timespec structure .................................. 27

5.1 Single Adaptive thread .......................... 44
5.2 Multiple Adaptive thread ......................... 45
5.3 CPU fairness with multiple adaptive threads ........ 46
5.4 Video quality in frames per second ................ 47
5.5 X11 Tardiness ........................................ 49
5.6 Timeliness limit for a purely fairshare scheduler .... 50
5.7 Misbehaving Thread ............................... 51
5.8 Application level throughput ....................... 54
5.9 Context switch costs (Uni-Processor) .............. 55
5.10 Context switch costs (SMP) ....................... 56
List of Algorithms

1. Xserver core event loop ........................................ 39
2. Timeslice calculation ........................................... 67
3. Pseudocode for the scheduler choice function .......... 68
4. Pseudocode for updating the virtual time for each task .. 68
5. Pseudocode for enqueuing a task onto the runqueue ... 69
6. Pseudocode for Dequeuing tasks from the runqueue ... 70
7. Pseudocode for calculating the fairshare timeslice .... 70
8. Pseudocode for preemption function ......................... 71
# Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFS</td>
<td>Completely Fair Scheduler</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>I/O</td>
<td>Input Output</td>
</tr>
<tr>
<td>UBC</td>
<td>University of British Columbia</td>
</tr>
<tr>
<td>POSIX</td>
<td>Portable Operating System Interface</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multiprocessing</td>
</tr>
<tr>
<td>STM</td>
<td>Software Transactional Memory</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
</tbody>
</table>
Acknowledgments

I would first like to thank my supervisor, Dr. Charles 'Buck' Krasic for his constant guidance and advice throughout the thesis. I am also grateful to all my fellow lab mates for making it a fun place to work in. I would also like to thank Dr. Norman Hutchinson for being my second reader, and for providing me with invaluable comments and suggestions to improve my thesis. I am immensely grateful to my family for being a constant source of love, affection and support.
Chapter 1

Introduction

In this chapter we provide the motivation for this work and detail the major contributions made by it. This work is part of an ongoing project at University of British Columbia (UBC), which aims to address the shortcomings in current software systems when dealing with time sensitive applications.

1.1 Motivation

Most of the major software subsystems in a modern computing platform - memory management, persistent storage management, processor scheduling and the networking subsystem, have been designed for throughput rather than timeliness. Increasingly, the distinction between general-purpose and real-time computing has become blurred as rich media applications such as video, audio and animated computer graphics have become an integral part of everyday computing. Two emerging trends are driving the development and deployment of these applications. First, these applications are increasingly being used on mobile devices such as smartphones and Internet tablets. The devices are notable in that they have modest processors to conserve power usage and maximize battery life. Second, at the other end of the spectrum, improvements in I/O devices such as cameras and displays are enabling higher data rates (e.g., high-definition H.264 video), for a high quality experience, but the compression and decompression requirements to support these rates can surpass the limits of the fastest processors currently available.
An important challenge in the design of these applications is the need to support these diverse environments, from very high-definition video for tele-immersion to low-resolution video for mobile platforms. An appealing idea is encode once, run everywhere, wherein data is encoded and stored once at high resolution, and then the applications adapt the data on the fly, at the client or the server or both, 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 content to the same client for applications such as surveillance cameras, film direction and editing, news feed selection, multi-party video conferencing, etc. More generally, it has applications in areas such as real-time visualization, virtual reality and interactive graphics, where large data sets need to be visualized in real time.

These adaptive applications are unique because they are both time-sensitive (i.e., they have timing constraints that must be satisfied for correct operation) and they can saturate resources. For example, digital multi-track audio processing software, used by musicians, has very stringent latency constraints, and also substantial CPU requirements due to sophisticated graphical visualizations. Similarly, the echo cancellation components of video conferencing systems are extremely latency sensitive, and the quality of video sources (e.g., 1080p HD) continues to improve and push on the computational performance limits of available processors. These requirements are challenging for commodity operating systems designed for throughput oriented and interactive applications. For example, Unix based feedback schedulers use a simple heuristic based on low CPU usage to give priority to interactive applications [6]. However, adaptive applications have high resource requirements and thus may not be run at the desired times. Running them with high priority, a basis for classic real-time scheduling schemes, does not work either because these applications would simply starve all other best-effort applications. We discuss related approaches in the next section.

1.2 Related Works
The scheduling problem has been studied extensively by the real-time community, starting from the seminal work by Liu and Layland[20]. Much of this work uses
release-times and priorities for scheduling real-time tasks, but to provide scheduling guarantees, these tasks use higher priority than best-effort tasks and can starve them.

Reservations [16, 19, 22, 23, 26] have been proposed as a method for avoiding starvation. With reservations, each task is given an allocation, consisting of a certain proportion of the CPU over a certain period of time to ensure that the release-times of real-time tasks can be satisfied. Best-effort tasks can be given a certain minimum allocation to avoid starvation, ensuring co-existence of both types of tasks. The RBED reservation scheduler [12] advances this work by allowing dynamic adjustment of both the proportion and the period parameters of tasks while still providing reservation guarantees. However, the main challenge with reservation-based systems is that they require specification of task resource requirements, and in general-purpose environments, such a specification may not be statically available.

As a result, several research projects have explored using online methods to estimate task resource requirements. Stankovic [21] uses a feedback controlled earliest-release-time first (FC-EDF) algorithm to adjust allocations of tasks to reduce the number of their missed deadlines. Abeni [10] uses feedback in a reservation-based scheduler to remove the need for specifying worst-case execution time (WCET) in the task parameters. The real-rate scheduler [15, 28] uses application-specific progress metrics to estimate resource requirements. These estimation techniques introduce two problems. First, the estimation needs to be accurate since under-estimation can lead to missed deadlines and over-estimation leads to poor resource utilization. However, accurate estimation is challenging when resource consumption varies unpredictably, as our results show for video applications. Second, feedback-based scheduling can lead to instability for adaptive applications since the scheduler’s resource estimation and allocation mechanism can interact poorly with the “back-off” behavior of the adaptive application (i.e., it is difficult to compose two or more feedback controllers).

Given these issues with reservations, this work revisits the use of priority-based scheduling for adaptive, time-sensitive applications. We use two methods to avoid starvation. First, we use fair share scheduling across both adaptive and best-effort applications. fair share scheduling not only avoids starvation, but also does not re-
quire specification or estimation of resource requirements, avoiding the problems described above. Second, we prioritize adaptive applications when they need to run their timer events, ensuring their timeliness. However, the priority of the application is elevated for a short period of time only, with resources being shared fairly over longer periods.

1.3 Contributions

This work resulted in a working system which comprises of a kernel scheduler and set of specially modified applications. The user level application specifies its timing requirements via a system call, and subsequently the scheduler will execute the application at their specified time, within an order of a millisecond. The system can also handle un-anticipated timing requirements associated with events like the arrival of some I/O. We modified a streaming video player application and the graphical display server to quantitatively evaluate the benefits of our approach.

As an added feature, we revamped the scheduling logic for the graphical display server to a fairshare model, which makes it more robust against an overly aggressive client. We also extended our precise timing mechanism to the nanosleep system call, thus allowing normal un-modified applications to have access to a super accurate sleep routine.

Special emphasis has been given to develop a stable, efficient, and up-to-date version of the scheduler for the Linux operating system. The scheduler has the added benefit of being completely tickless (i.e., it does not rely on a periodic timer tick). This feature would be especially useful for power constrained devices which need to limit their interrupt flow rate.

1.4 Thesis Structure

The thesis is structured into four major chapters, Chapter 2 explains the overall system design and the algorithms involved, next we explain the finer points about the kernel implementation in Chapter 3. Our modifications for the Xserver are detailed in Chapter 4. Chapter 5 contains an evaluation of various aspects of our system including timeliness improvements, performance, overhead and human effort. Finally Chapter 6 details the possible directions in which this work can be
extended.

This thesis also contains an appendix that contains pseudocodes for all the major functions for our kernel implementation and some programming tips and debug techniques which are quite handy while doing kernel programming.
Chapter 2

System Design and Algorithms

In this chapter we will explain the key conceptual and algorithmic principles behind our system. We will first formalize the design requirements for our system, and then explain the programming model for using our system. In order to use our system, the application must be structured as an event driven model, with short running events. The events are atomic units of computation, and thus the minimum latency of the application is bound by duration of the longest running event. Next, we introduce our cooperative polling infrastructure and then provide details about the two major components of this system - the application level event scheduler and the kernel scheduler. After that we explain how we combined our cooperative polling infrastructure with the I/O polling infrastructure in the kernel, to provide sub-millisecond response to I/O. Lastly we describe how we extended this idea by converting the nanosleep system call to implicitly use this feature. This chapter only provides the conceptual ideas, Chapter 3 explains the details for our actual implementation.

2.1 Requirements

Based on above shortcomings for commodity operating system schedulers and realtime schedulers, the set of requirements for our scheduler is as follows:

1. The user application must be able to communicate its timing requirements to the kernel scheduler.
2. The kernel scheduler must support both best effort tasks and time sensitive tasks.

3. The scheduler should support user level scheduling for a group of tasks. This requirement evolved into the concept of domains as explained in Section 2.3.2.

4. The scheduler must be multiprocessor-aware.

5. The scheduler should be work-conserving and starvation-free.

6. The scheduler must minimize the scheduling latency for tasks with timing requirements.

7. The application throughput should not suffer, despite the improved timing behavior.

8. The timing sensitive tasks must be able to respond to I/O as quickly as possible, with near zero scheduling latency.

9. Task priorities are the primary mechanism by which user level applications affect the kernel scheduler. Therefore to maintain compatibility with existing applications our scheduler must support some form of task priorities.

We will explain our solution to each of the above requirements in the section(s) below.

2.2 Programming Model

Our system uses an event-driven programming model, and two symbiotic schedulers (at the application and the kernel level) to support the timing requirements of adaptive applications. We describe the programming model and the scheduler APIs below.

Our event-driven programming model is inspired by the principles of reactive programming [11]. 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 frees the programmer from the
need to use locking and synchronization primitives required in multi-threaded pro-
grams.

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. Avoiding blocking is generally challenging to satisfy in practice. However, we have im-
plemented an asynchronous I/O subsystem that eases programming significantly. Having only short-running events seems counter-intuitive, since long computations seem inherent to adaptive applications (e.g., video decompression). However, most long computations use loops, and each iteration can be divided into a separate event. This focus on short, non-blocking events promotes an environment that allows software to quickly respond to external events, such as arrival of data from the network, hence the name reactive programming.

The non-preemptive and short-lived computation requirements match well with event-based applications, but it should be possible to use non-preemptive threads libraries such as Pth [13] for implementing adaptive applications. Another alternative is to use the TAME event system [18] that offers the programmability advantages of threads.

2.2.1 Application Scheduler API

Our programming model uses a per-thread application scheduler that operates independently of application schedulers in other threads. Program execution is a sequence of events (function invocations) that are run non-preemptively.

Figure 2.1 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 one of its events to end the dispatching of events. The

```c
submit(EventLoop *l, Event *e);
cancel(EventLoop *l, Event *e);
run(EventLoop *l);
stop(EventLoop *l);
```
struct Event {
    enum { TIMER, BEST EFFORT } type;
    Callback callback;
    TimeVal release-time;
    TimeVal app.time;
};

Figure 2.2: Event type definition.

application can also call cancel to revoke an event it had previously submitted.

Figure 2.2 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-time field specifies an absolute time value. Timer events are not eligi-
ble for execution until the release-time time has passed. Once eligible, timer
events take priority over all best-effort events. Note also that the application sched-
uler never skips or drops any timer events, including delayed events, and it is the
responsibility of the application to adapt to delayed events. The app.time field
is used by best-effort events. Its value is used to control execution order among
threads in a thread group as explained in the next section.

The scheduler does not require any specification of resource requirements
because we assume that the application can adapt its best-effort events during
overload. We also assume that timer events do not saturate the processor. This
assumption is reasonable because most computation within adaptive applications
is not time sensitive. For example, CPU-intensive video decoding does not have
explicit timing associated with it, while video display has timing constraints but
requires limited processing. Other adaptive applications such as real-time visual-
ization share similar characteristics. If schedulability guarantees are required for
the timer events, then existing real-time approaches described earlier can be used
to schedule these events.

2.2.2 Kernel Scheduler API

Our kernel scheduler uses a hierarchical scheduling discipline in which each thread
belongs to a group. Threads within a group are allocated resources according
to application-specific policy as expressed by the \texttt{app.time}, or application time value of the best-effort events. Similar to virtual-time based schedulers, the thread with the least application time is run within the thread group, allowing application-specific fairness. For example, a video application can set this value to the number of frames processed. Then two video threads running in the same thread group (e.g., multi-party video conferencing) would have the same frame rate or video quality even though the CPU is allocated differently to the threads. We call the thread with the least application time the most important thread in the group. Thread groups can be used to schedule the threads of a single application, cooperating adaptive applications, or all the applications of a user. Different thread groups are allocated resources using a weighted fair share approach.

### 2.3 Cooperative Polling

Cooperative polling supports inter-application cooperation by sharing an application's timing and progress information with the kernel and with other applications. Our overall scheduling approach combines application-level event-driven scheduling and kernel-level fair share scheduling with a new \texttt{coop.poll} system call that serves as a conduit between the two schedulers. In this section, we describe an event-based application scheduler, and its straightforward extension to support inter-application cooperation through the use of \texttt{coop.poll}. Next, we describe our fair share based kernel scheduler, and how the information provided by \texttt{coop.poll} is used by the kernel scheduler to provide enhanced service to adaptive applications without compromising fairness.

#### 2.3.1 Application Scheduler

Figure 2.3 shows the application-level event scheduling algorithm. The events are stored in the \texttt{timer.events} and the \texttt{best.effort.events} priority queues, sorted by release-time and application time respectively. The \texttt{submit} and \texttt{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 scheduler services all events provided by the application, even when events
run(EventLoop 1) {
  do {
    if head_expired(l.timer_events) {
      e = q.head(l.timer_events);
      cancel(l, e);
      callback_dispatch(l, e);
    } else if q_not_empty(l.best_effort_events) {
      e = q.head(l.best_effort_events);
      cancel(l, e);
      callback_dispatch(l, e);
    } else {
      yield(l);
    }
  } while (l.stop != True);
}

yield(EventLoop 1) {
  if q_not_empty(l.timer_events) {
    sleep until next release-time;
  } else {
    l.stop = True;
  }
}

Figure 2.3: Application-level event scheduler.

arrive faster than they are dispatched. This approach can allow the priority queues 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 scheduler 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 rate at which certain events are generated and by invoking cancel for some existing events to skip less important steps (e.g., parts of video decoding) as necessary to maintain timeliness [17].

The coop.poll System Call

We improve kernel scheduling performance and enable cooperation between time-sensitive applications with a new coop.poll system call that voluntarily yields the processor. An example of its usage is shown in Figure 2.5. The system call takes a thread group (recall from Section 2.2.2 that the kernel uses a group-based
hierarchical scheduler), and two IN-OUT event parameters (IN parameters pass a value, OUT parameters return a value). The IN values of the event parameters specify the earliest timer and the most important best-effort event in the current thread to the kernel scheduler. These 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 \texttt{coop.poll} returns, the OUT values are set to the earliest release-time among timer events across all threads, and the least application time among the best-effort events of all threads within the thread group. Our kernel expects that threads will yield voluntarily (i.e., call \texttt{coop.poll}) according to the OUT values. In exchange for this cooperation from the thread, the kernel scheduler will give it preferential treatment, as described later in Section 2.3.2. Thus these event parameters represent a quid-pro-quo quality of service agreement. Figure 2.4 illustrates how the user level applications are connected to the kernel scheduler via the coop.poll interface.

Figure 2.5 shows that the \texttt{coop.poll} system call can be incorporated in the application scheduler shown in Figure 2.3 by simply modifying the \texttt{yield} function (the \texttt{run} function is unchanged). The \texttt{yield} function in Figure 2.5 is designed so that events are executed across threads in the same order as events in the single-threaded scheduler shown in Figure 2.3. To enable sharing, we add two proxy events to the event loop state, coop.timer.event and coop.best.effort.event, that act on behalf of other applications. The release-time and application time of these proxy events are set by \texttt{coop.poll} to reflect the earliest timer across all other threads and the most important best-effort event among all other threads within the thread group. After the \texttt{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 \texttt{yield} so that the current thread yields voluntarily to other applications in the \texttt{callback.dispatch} routine shown in Figure 2.3. The \texttt{cancel} calls at the beginning ensure that the event queues contain only events internal to the current task, preventing \texttt{yield} from spinning (i.e., live-locking) wherein a thread transitively yields to itself.

In summary, the cooperative polling model combines reactive programming with a new \texttt{coop.poll} primitive to allow cooperation between multiple reactive applications. In the next section, we describe the kernel support we have designed
and developed that allows our model to support a mixed environment consisting of adaptive and traditional best-effort applications.

2.3.2 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-queueing (WFQ) to accomplish fair sharing. Below, we provide an overview of this algorithm before describing our cooperative fair share scheduler.
yield(EventLoop 1) {
    cancel(1, l.coop.timer.event);
    cancel(1, l.coop.best.effort.event);
    l.coop.timer.event =
        q.head(l.timer.events);
    l.coop.best.effort.event =
        q.head(l.best.effort.events);

    // coop.poll sleeps until
    // next timer-event release-time or when
    // best-effort event has least app.time
    coop.poll(l.thread.group,
        /* IN-OUT */
        &l.coop.timer.event.release-time,
        /* IN-OUT */
        &l.coop.best.effort.event.app.time);

    // events call yield when dispatched
    submit(l, l.coop.timer.event);
    submit(l, l.coop.best.effort.event);
}

Figure 2.5: Using the coop.poll system call for inter-application cooperation.

**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. The scheduler uses a run queue sorted by earliest virtual time to select best-effort threads (the next section describes support for threads that use the coop.poll system call). The run queue keeps track of minimum and maximum virtual times as well. 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 which sleep cannot accumulate CPU allocation that would subsequently allow them to starve other threads. This "use it or lose it" approach is an elegant method of accommodating the sporadic requirements of I/O-bound threads. The use of maximum virtual time is less frequent, but important. If the processor has been idle (i.e., the run-queue is empty), then the virtual time of the next thread is set
to be the maximum virtual time of the queue, ensuring that thread virtual times are non-decreasing, and thus threads cannot gain unfair CPU time. During thread migration the migrating thread’s virtual time is reset to the maximum virtual time on the target CPU, recalibrating the thread’s virtual time as explained in Section 2.3.2.

Aside from which thread to run next, the scheduler also computes how long the thread should run, i.e., the timeslice of the thread. It computes the timeslice as \( ts = \frac{\text{period}}{N} \), where \( \text{period} \) is a global scheduling parameter that expresses the minimum responsiveness target of the kernel, and \( N \) is the number of runnable threads. A typical value of the period parameter is 20ms. The idea is that every runnable thread should get a chance to run once per period. Smaller period values improve short-term fairness, but to prevent live-lock, and limit context-switch overhead, the scheduler enforces a minimum timeslice (e.g., 100 microseconds). Note that our approach approximates a quasi-periodic scheduler since the value of \( N \) will vary dynamically.

**Cooperative Fair Share Scheduling**

The cooperative fair share scheduler provides enhanced timing support by combining earliest-release-time first scheduling with fair sharing.

We define threads as being *cooprealtime* or *cooperative* when they issue the `coop.poll` system call and adhere to the cooperative behavior described below. This system call inserts a thread issuing the call into a timer queue, sorted by earliest release-time. When there are cooprealtime threads in the timer queue (either runnable or blocked), the scheduler uses the queue to compute the earliest release-time among all the cooprealtime threads, and uses this release-time to adjust the timeslice of the thread that is chosen to run (shown below).

When a cooprealtime thread is run, the OUT value of the timer release-time parameter shown in Figure 2.5 is set to the timeslice value, so that the thread can yield at the end of its timeslice. The cooperative scheduler considers two cases depending on whether the earliest timer release-time is in the future or in the past:

1. When the release-time is in the future, the scheduler selects the thread with the smallest virtual time and sets its timeslice \( ts = \min(release-time - now, \frac{\text{period}}{N}) \).

2. Otherwise, the scheduler selects the thread with the earliest release-time and
set its timeslice $ts = 0$.

In the first case, when $\text{release-time} - \text{now} > \frac{\text{period}}{N}$, the scheduler uses the fair share scheduling algorithm described earlier. Otherwise, it uses earliest-release-time first scheduling because the next thread is only allowed to execute until the next release-time. In the second case, the earliest coop real-time thread is selected to run, but its exposed timeslice is set to zero, allowing the thread to run but only for a minimal amount of time. As a result, the application scheduler of a coop real-time thread will execute timer events with a 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 behavior expected of coop real-time applications, and we say such an application is cooperative.

### Policing Misbehaving Threads

Applications using `coop.poll` receive preferential treatment because they are scheduled immediately when their release-times are due (the second case described above), but our scheduler includes a simple policing mechanism to ensure that they do not gain long-term advantage by misusing `coop.poll`. Our policing mechanism demotes a coop real-time thread to best-effort status so that the thread is subject to fair sharing exactly as other best-effort threads. In particular, the kernel does not maintain release-time information for best-effort threads the way it does for coop real-time threads, so they are unable to run at specific release-time times. Policing is temporary and threads regain coop real-time status each time they call `coop.poll`. The scheduler performs policing for three reasons: 1) running beyond timeslice, 2) non-cooperative sleep, and 3) exceeding a virtual time threshold. We describe these reasons below.

The scheduler enforces a thread's timeslice with a kernel timer. However, when a coop real-time thread is the selected thread (i.e., it had called `coop.poll` when it blocked), the kernel timer is scheduled a short period after the timeslice. This period is a scheduling parameter called `coop.slack` (1ms by default). Cooperative threads are expected to yield soon after their timeslice release-time, but if they fail to do so within the period, they are demoted. Second, applications using the reactive model (see Section 2.2) are normally expected to sleep by calling
coop.poll. If the application sleeps in the kernel for any other reason (i.e., the thread blocks outside coop.poll), then it will not have a release-time and must be demoted.

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 timer release-time into the timer queue. However, 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 that issues many release-times in a short period can increase its share of CPU in the short term, but this will cause its virtual time to advance faster than other threads. If its virtual time exceeds the others by more than the threshold, it is demoted. Although a misbehaving thread may periodically become cooperative, the threshold will ensure that the cumulative effect is bounded. Since the threshold is on the order of milliseconds, the long term advantage a single thread can gain is small. A malicious application can employ a large numbers of threads, either sequentially or concurrently, but higher-level resource containment mechanisms are more appropriate for defending against such attacks.

**Thread Groups/Domains**

Until now, we have described the use of application-level timer events to combine release-time-based scheduling with fair sharing, where fairness is defined in terms of actual execution time. The scheduler can also use the application time of best-effort events to implement application-specific fairness policies within a thread group. Adaptive applications are able to adapt the quality of their results according to the share of processor they are allocated. Furthermore, the relationship between quality (e.g., frame rate) and processor time can be highly variable over time (see Figure 5.3, 5.4 and Section 5.1.2), so that a fair CPU allocation can result in an extremely unfair quality allocation. The use of thread groups allows a set of threads to pool their overall CPU allocation, and then to subdivide the allocation within the group according to application policy. For example, the group abstraction can be used to implement equal quality for all threads within the group.

The scheduler requires the following modifications to support thread groups. It maintains a best-effort queue for each thread group, sorted by least application
time. The queue contains all the best-effort events supplied to `coop.poll`. Furthermore, all threads in the group share a single common weight and virtual time value, and the common weight value is the sum of the weights of the individual threads within the group. Recall from Section 2.3.2, when the next coop real-time release-time is in the future, the scheduler selects the next thread to run based on fair sharing. With thread groups, the scheduler uses the group's best-effort queue, replacing the fair share selection with the most important thread in the group. In addition, the policing mechanism described in the previous section removes a thread from a thread group during demotion. Our evaluation in Chapter ?? demonstrates the use of thread groups to provide fairness based on equal video quality.

**Task Priority**

Most operating system schedulers support some form of task priorities, but the actual effect that a difference in priorities has is scheduler dependent. Some schedulers choose to translate a higher priority to mean higher preference, whereas others translate it to higher CPU allocation and some onto both. For our virtual time based scheduler, higher task priority maps onto higher CPU allocation. We translate the priority onto a weight parameter where the virtual time for a processes with higher weight advances at a slower rate. Thus in the long run, the task will have a CPU allocation which is directly proportional to its priority.

**Multiprocessor Support**

We follow Linux's model of a per CPU runqueue, with tasks being migrated from one CPU to another to achieve load balancing. Each CPU has its own notion of virtual time, hence migrating tasks need to recalibrate their virtual times. We set the migrating task's virtual time equal to the maximum virtual time on the destination CPU. This prevents the migrating task from interfering with the execution order of the existing tasks on that CPU. We have not implemented any specific load balancing policy and are currently re-using the default policy on Linux.

For ease of implementation and simplicity, task migration is only supported for best-effort tasks, coop real-time tasks are pinned onto their current CPU. We intend to handle load balancing for coop real-time tasks at the user level, wherein
the application initializes an event loop on each CPU, and then dispatches events to each of them, according to its load balancing policy.

2.4 Combining **coop.poll** with **select**

The coop.poll facility as described above, enables a task to respond to a known release-time with minimum latency. This mechanism works as long as the application informs the kernel of its release-time in advance, but, sometimes the application has sporadic requirements associated with I/O. For example, the application might receive the release-time information via a network socket or by reading an on-disk file. In such cases, the application needs to be run as soon as it receives some input. This will ensure that it has a chance to read in the release-time information, before the release-time expires.

The primary mechanism by which applications synchronously poll for activity on a set of I/O file descriptors is via the select/poll system call interface. All Portable Operating System Interface (POSIX) compliant operating systems contain an implementation for the select/poll interface. Combining coop.poll with the select interface would allow a task to yield the CPU in a controlled manner and have the kernel monitor a set of file descriptors for activity on the task’s behalf. As part of this design we modify the coop.poll system call to incorporate an extra IN argument which contains information about the set of file descriptors which are to be monitored. To help implement this feature we introduce two new modified task states to the kernel scheduler:

- **PollSleep**: This is an enhanced version of the sleep state, wherein the task is required to be run with near zero scheduling latency. The latency between the task waking up, and being finally run, should be minimized.

- **PollRunnable**: A task in this state is both runnable and asynchronously monitoring the file descriptors for activity. It would be run either when the scheduler decides that its the most fitting task to run next, else when the kernel detects I/O on the given file descriptors.

The state diagram given in Figure 2.6 illustrates the transition between all the task states. Tasks can yield the CPU due to their timeslice expiring or if they ex-
plicity invoke the *sched_yield* or the *coop_poll* system call. Well behaving cooprealtime tasks are expected to use the *coop_poll* call to yield the CPU before the expiration of their timeslice. Subsequently depending on the *coop_poll* input parameters the cooprealtime task will transition onto a PollSleep, PollRunnable or Sleep state. If the task is interested in monitoring for i/o and has active pending events then it transitions onto the PollRunnable state, else if it has no pending events it goes onto the PollSleep state. We implement the PollRunnable state by retaining the task in the scheduler’s runqueue, and simultaneously registering interest in the given file descriptors. We treat the exact time at which the kernel detects any activity on the file descriptors as an implicit release-time for the task. We replace the existing release-time associated with this task if this new release-time is earlier than it. The PollSleep state is implemented by registering interest in the given file descriptors and then putting the task to sleep till its given release-time arrives. This is an enhanced form of sleep, where the task wakeup is treated as an implied release-time. Once it yields the CPU the only way a task can transition onto the running state, is if the scheduler choice function decides to run it next. The scheduler only considers tasks which are either in a Runnable or PollRunnable state while making this choice. Both the newly introduced states inject a new release-time into the system while another task is executing, next we will explain how our scheduler takes these release-times into account.

If the newly inserted release-time is earlier that the earliest known release-time in the system (system here refers to each individual CPU, since each CPU is running its own independent scheduler) at that point, then the timeslice for the currently executing task will overrun this newly inserted release-time. This is because, the scheduler was un-aware of this newly inserted release-time while calculating the current timeslice, and the atomic nature of our timeslices ¹ prevents us from preempting the currently executing task. Therefore in order to minimize the latency between detection of i/o and the subsequent execution of the task, we need to compromise on the atomic nature of our timeslices. The preemption logic for best-effort tasks and cooprealtime tasks are different. We will explain them both in detail in the next two subsections.

¹In our system a task cannot be involuntarily preempted before the expiration of its timeslice
2.4.1 Best-effort Task Preemption

Best effort tasks can be preempted by directly invoking the scheduler at an appropriate time. (When the code is not holding any locks, etc). This simple preemption logic could potentially lead to unbounded context switches, therefore we limit them by making the best effort task non-preemptable within the first X microseconds of its execution, where X is equal to the minimum timeslice allowable on the system. Preempting a cooprealtime task is a bit more involved than this as explained below.

2.4.2 Avoiding Involuntary Preemption for Cooprealtime Tasks

When the kernel preempts a cooprealtime task, it has no way of knowing whether the application has serviced its last reported release-time or not. Therefore the kernel has no valid release-time information for this task, and is forced to demote it to best effort status. The crux of the problem lies in the disconnect between
the user level application and the kernel scheduler. An ideal solution would be for the user level application to inform the kernel of the next upcoming release-time every time it services its own earliest release-time. This would keep the kernel scheduler in sync with the state of the user level application and avoid the forcible demotion. This approach has but one serious drawback, it can lead to a very high number of kernel-user level boundary crossings.

A practical way to implement the same thing would be to setup frequent voluntary preemption points along the task’s execution timeline. Let’s refer to these points as rendezvous points, the user application and the kernel would exchange release-time and best-effort event information at these points, and these points would also act as voluntary yield points. The kernel will reschedule the task at these points, if there is a task with an earlier release-time waiting to be run. The rendezvous points are implemented via a special system call known as a rendezvous call (as our implementation chapter will explain, we decided to combine the coop_poll call and the rendezvous call into one unified system call).

The preemption logic for coop realtime tasks gives the task a chance to voluntarily yield the CPU before forcibly preempting it. It grants the task io_latency + coop_slack time units to voluntarily yield itself before forcibly preempting it and demoting it to best effort status. Let us refer to the periodic time interval between the rendezvous points by the term io_latency. As Figure 2.7 illustrates visually, when a new release-time is inserted into the system while a task is executing, the sum of io_latency and coop_slack determines the maximum amount of time the system is willing to wait before forcibly preempting the current executing task. Thus this value governs the maximum latency which is added to the system, when reacting to a new hitherto unknown release-time. The task can easily avoid this forcible preemption by strictly honouring the rendezvous points. Its up to the user level task to honour its rendezvous points, the kernel only informs the application about the next rendezvous point each time the latter makes a rendezvous call.

To summarize, the changes described in this section have modified the system such that it can handle the insertion of release-times via a mechanism separate from the coop_poll system call. This allows the system to asynchronously insert new release-times, which may be associated with activities like: task wakeup or I/O activity. If the currently executing task is a cooprealtime task, then it will be
end of timeslice task A has to yield for task A before IO was detected
else gets demoted

Rendezvous Point

Task B: Coop realtime task which detected IO activity

Task A: Coop realtime task being preempted

IO Latency

IO Latency + coop_slack

Timeline for Task A

Figure 2.7: coop realtime task timeline

informed of this new release-time at its next rendezvous point. The task then has coop_slack + io_latency time units to react to this new release-time, before being forcibly preempted.

2.5 Converting nanosleep to a Partial coop_poll

The nanosleep system call is used by applications which need to sleep for a very accurate amount of time, but, even this call is subject to the inaccuracies which are introduced due to scheduling latency. To alleviate this problem we converted the nanosleep call into a partial coop_poll with a release-time in the future and no pending best-effort events. The sleep timeout period is converted into a release-time value by adding the current time to the timeout period. This is the exact time at which the task expects to run. Informing the kernel of this timing requirement in advance enables it to reduce the scheduling latency for this task. The amount of time the task has to wait to be scheduled onto the CPU after waking up is minimized. This is a partial coop_poll call since the bottom half of the coop_poll call is unneeded in this case. We are only interested in informing the kernel of our timing requirement without needing to know the timing requirements of the other tasks on
Section 5.5 in chapter 5 shows that this change reduced the inaccuracy in the nanosleep to within a millisecond.

2.6 Chapter Summary

We have introduced a new programming interface for structuring time sensitive applications, and have provided kernel support for this in the form of a new scheduler and a system call interface by which the application informs the kernel of its timing requirements. Our system can deal with known and unknown timing requirements. i.e., it can also take into account timing requirements which get introduced into the system by the kernel on behalf of some user level task. This enables our system to react to I/O and to explicit timing requirements, with minimum latency, and without compromising on long term fairness. We have combined coop.poll and select into one integrated system call which is capable of both exchanging timing information with the kernel and asynchronously poll for I/O. The application using this combined system call is ensured of being scheduled as soon as there is any activity in the interested file descriptors.
Chapter 3

Implementation

In this chapter, we provide overview and details involved in implementing the design concepts of Chapter 2 onto a working prototype. We first give a high level overview of the implementation, next we formalize the invariants maintained. Invariants help to debug and reason about a system. We then go on to describe the porting effort involved in keeping our codebase up-to-date with recent changes in the kernel. Finally we provide implementation details about all the major functions in our scheduler.

3.1 Implementation Overview

Our scheduler is implemented as a new scheduling class for the Linux kernel. All the data structures for our scheduler are per CPU, and they are protected via the runqueue lock. This design greatly simplifies the locking requirements for our code, since the core scheduler framework takes care of locking and unlocking the runqueue lock at the appropriate places. We have three per-cpu priority queues as given below:

- VT queue: This queue contains all runnable tasks, ordered by their virtual time.
- Sleep queue: This contains all cooprealtime tasks which are sleeping, ordered by their future release-times.
• Release-Time queue: This contains all runnable cooprealtime tasks, ordered by their release-times

All these queues are implemented via a heap. The heap data structure gives us O(1) access to the head of the queue, and O(log(n)) deletion and insertion.

Each CPU has a set of coop domain/thread groups, each of which contains a release-time queue and an best-effort queue. Similar to the per-cpu queues, the release-time queue contains all the cooprealtime tasks within a given domain ordered by their release-times and the best-effort queue contains all the tasks within a domain ordered by their user defined priority field. The per-cpu queues contain data which is already present in one of the domain queues, this redundancy was added to avoid iterating over all the domains while choosing the next task to run. This is a classic case of space vs time tradeoff. There is also a special temporary coop domain per CPU, which is used to track tasks which need to transition into a cooprealtime state only temporarily. Tasks within this domain do not share their virtual times.

We enforce our timeslice with the help of a one-shot high resolution timer. Our scheduler does not require the periodic timer tick, thus making it quite suitable for devices with low power requirements. The periodic timer tick introduces unnecessary interrupts into the system, preventing the CPU from transitioning into a deeper sleep state.

The set of states for a task has been expanded to include the following extra states:

1. PollSleep: This is a modified sleep state wherein the task needs to be run as soon as it's woken up. The task transitions onto this state via a coop.poll call.

2. PollRunnable: The is a modified runnable state, wherein the task is in a runnable state and is also interested in monitoring a set of file descriptors for any activity. The Task should be run either when its turn arrives or there is any activity on the monitored file descriptors.

These extra states are implemented via a set of flags embedded in the task descriptor structure. The task descriptor has also been augmented with extra fields to store
The invariants in our system are listed below:

- Virtual time shall only flow forward. At no point in time will the virtual time of a task go backward, apart from when it is being migrated between processors. This property ensures that a task can never gain an unfair advantage by decreasing its virtual time. We use a timespec structure as defined in Figure 3.1 to store this value.

Assuming that the time_t opaque type is implemented as signed 32 bit integer, the virtual time can overflow in $\frac{2^{31} - 1}{365 \times 24 \times 60 \times 60} \approx 68$ years. Although this poses no immediate problems, one could fix this by detecting when an overflow is about to happen, and subsequently resetting the virtual time for all the tasks on that processor to zero. We can also use wrap-around arithmetic to solve this problem. Wrap around arithmetic is a clever way of doing comparisons for unsigned integers, such that overflow does not affect the results. The Linux kernel contains a macro (time_after) which already implements this logic. ¹

- A coorealtime task has to have an entry in either the per-cpu release-time queue or the per-cpu sleep queue.

- A sleeping task cannot have an entry in any of the scheduler queues, apart from the per-cpu/per-domain sleep queues.

³LXR url for the time_after macro: http://lxr.linux.no/linux+v2.6.26.3/include/linux/jiffies.h#L106
• A task executing in user mode does not have any entries in the kernel release-
time/sleep queues.

• All tasks in a coop domain share their virtual time, except for the ones in the
temp domain.

3.3 Porting the Implementation to the 2.6.25 Kernel

In order to keep up with the latest changes being introduced into the Linux kernel,
we decided to port our implementation onto the official Linux kernel repository.

The scheduler code has undergone a lot of churn during the period of this thesis
work. Linux 2.6.25 introduced a whole new scheduler subsystem along with a new
fairshare scheduler known as Completely Fair Scheduler (CFS). The 2.6.25 kernel
also features some advanced resource containment mechanisms like group aware
scheduling and cgroups. Group aware scheduling allows you to subdivide the cpu
allocation according to user defined groups. These scheduling groups are persistent
across CPUs. The cgroups feature is a more generic framework for implementing
resource containers, by which a group of tasks can be associated with a set of
parameters for one or more subsystems. The user interfaces with the framework
via the proc filesystem, cgroups are modelled as directories onto which one can
add processes. The core scheduler code has been modularized to allow for easy
and clean implementation of specific scheduling regimes. We have implemented
our scheduler as yet another scheduling class, which can co-exist with CFS.

3.3.1 Linux Modular Scheduler

All the major scheduling functions listed in Table 3.3.1 have been converted into
function pointers, wherein the scheduling class is responsible for providing the
specific function. The function pointers are bundled together using a read-only
structure, these function pointers can therefore be resolved during compile time,
avoiding the usual runtime overhead associated with using function pointers. Most
of our scheduler code is contained within this new scheduling class, apart from

\[^2\]URL for official Linux repository:git://git.kernel.org/pub/scm/Linux/kernel/git/torvalds/Linux-2.6.git
some minor hooks into the core scheduling code. These hooks are responsible for the cleanup and initialization of data structures during task destruction/creation.

- Choosing the next task to run
- Dequeuing a task from the run queue
- Enqueing a task onto the run queue
- Context switching out of a task
- Migrating a task to a remote run queue
- Load balancing function
- Checking whether a recently woken up task should preempt the currently running one.

3.3.2 CFS

The core logic in CFS is surprisingly similar to our scheduler, apart from one key difference - CFS relies on the scheduler tick to dynamically adjust its timeslices, instead of statically assigning one.

CFS replaces the traditional run queue by an ordered red black tree. Runnable tasks are ordered by their virtual runtimes, which are calculated by dividing the execution time by the number of runnable tasks on the CPU. At each scheduling decision, the task with the lowest virtual time is chosen to run next. This algorithm is almost exactly the same as the one used in our scheduler.

CFS does not have static timeslices, instead the scheduler calculates the ideal time for which the task should run depending on a global period parameter and the task’s weight parameter. This calculation is exactly the same one our scheduler uses, as given in Algorithm 7. CFS re-calculates this ideal timeslice value on each scheduler tick, and reschedules the currently executing task in case it has run beyond this time period. This allows CFS to dynamically tune its timeslice while the task is running, taking into account any new tasks which have become runnable. This added tunability comes at the cost of having to maintain periodic scheduler ticks.
3.4 Implementation Details

The major functions for a scheduler are listed in Section 3.3.1. Since the Chapter 2 already provides the conceptual basis for all these functions, we will only provide the major implementation details in the following paragraphs. The pseudocode for most of the functions is provided in Appendix A.

Implementing Domains We implement the sharing of virtual times for all the tasks within a domain by having only one representative from each domain in the VT queue. If chosen to run, this representative delegates the choice of whom to run to another function, as given in Algorithm 3. We only implement a single level of depth while grouping tasks, i.e., we do not support nested groups.

Rendevouz Calls The rendevouz call has been combined with the coop.poll call to one unified system call. The rendevouz call functions in exactly the same manner as the coop.poll call, except that the task does not necessarily yield the CPU. The output parameter of this combined call takes into account the earliest per-cpu release-time and the time of the next rendevouz point. The earlier of the two is passed onto the user level application.

Along with some help from the user level application the kernel is able to intelligently determine whether the call should yield or not. The kernel does not yield, if the call is made before the expiration of the task’s current timeslice and the task indicates that it still has work to do. A task is deemed to have work when it has pending best-effort events or expired timeout events. We must also handle the special case where the task wants to yield the CPU although it has pending work, this case occurs when a task yields the CPU due to another higher priority best-effort event being present in its domain. We handle this special case by having the application set a special flag in the input parameter for the call to indicate this scenario.

Combining coop.poll with select We implemented the combined coop.poll and select call by re-using the core I/O polling mechanism already present within the Linux kernel. This polling mechanism in the Linux kernel uses the con-
cept of waitqueues to implement waiting on I/O. All tasks interested in waiting for a particular event, insert themselves onto a waitqueue associated with that event, and then go to sleep. Once the task gets activated and is chosen to be run by the scheduler, it wakes up and removes itself from the waitqueue. Therefore the very act of running a task de-registers it from the waitqueue. This fact, greatly simplifies our implementation, since this allows us to cancel an ongoing coop_poll+select call by simply running the task.

The combined call takes in all the parameters for the select call (input, output and exception file descriptors) apart from the timeout parameter. The timeout parameter is implicitly calculated by taking into account the given deadline. The kernel uses the presence of these extra parameters to infer whether the task is interested in doing a combined call. If none of them are present, the call degenerates into a simple coop_poll call.

The combined call will transition the task onto either the PollRunnable state or PollSleep state depending upon the given input parameters. If the task indicates that it has no pending events currently (release-times in the future, no best-effort events), then it will transition onto the PollSleep state, else it will go onto the PollRunnable state. Since both these states require the task to monitor a set of file descriptors, we call out into the core polling mechanism to register ourselves onto the correct waitqueues. The time till the next given release-time is set as the timeout parameter. We also modified the mechanism by which the kernel implements this timeout to use high resolution timers instead of the old timer wheel mechanism. The timer wheel's accuracy is constrained by the timer frequency, which in most kernels is set at 250 HZ, thus giving us an accuracy of 4 ms. The high resolution timers enable sub millisecond accuracy.

Next we will explain how we go about implementing these two new special states.

PollRunnable State The PollRunnable state is implemented by retaining the task in the per-cpu runqueue and simultaneously registering interest in the given file descriptors. From this state the task can either transition into the runnable state or directly go to running. The task transitions onto a runnable state if the given release-time expires or there is any I/O detected on the provided file descriptors.
The task can also directly start running if the scheduler decides that it is the most eligible task to run next. As explained in the beginning of this paragraph, we do not have to explicitly de-register ourselves from the waitqueues, the task does this by itself once its executed. In case the task transitions into the runnable state, the logic described in Section 2.4 ensures that the amount of time that the task has to wait in the runqueue before being scheduled onto the CPU is minimized. As part of that we may shorten the timeslice allocated to the currently executing task and reprogram the timeslice timer to fire earlier. We also update the per-process timeslice parameter maintained by the kernel to reflect this shortened timeslice. The kernel needs this timeslice parameter to distinguish between a rendezvous call and a coop_poll call (see Section 3.4).

**PollSleep State** The PollSleep implementation is very similar to the PollRunnable state, the only difference being that the task is removed from the per-cpu runqueue. Hence the task can only transition onto the runnable state, it cannot directly go to running from this state.

**Load Balancing** We re-use most of the load balancing logic already present within the kernel. The core scheduling code expects each scheduling class to provide an iterator to go over the runnable tasks within that scheduling class. We implemented this iterator by maintaining a list of runnable tasks. This extra list was needed because there was no central data structure containing all runnable tasks. Recall that there is only one entry representing all the tasks for each domain in the per-cpu VT queue.

There are two forms of load balancing within the Linux kernel, active and passive. Active load balancing refers to the migration of already runnable tasks from one CPU to another, to reduce imbalance. The kernel can also attempt to load balance the CPUs whenever a new task is created, this is referred to as Passive load balancing. Due to time constraints, we have only implemented support for active load balancing.

**Task Preemption** The preemption logic as given in Algorithm 8 gives an unconditional preference to realtime tasks, therefore if a realtime task wakes up, the
currently executing task is preempted in favor of it. The logic also ensures that best-effort tasks under our scheduling regime are guaranteed to run for the minimum timeslice before being preempted. For coop realtime task's the logic shifts the timeslice timer to \( \text{io\_latency + coop\_slack} \) time units from now. (This is only done if the existing timeslice endpoint is further than \( \text{now + io\_latency + coop\_slack} \))

**Task Priority** We translate the task's static priority into a weight parameter via a statically encoded table. We also maintain the total sum of weights for a given CPU and the sum of weights for each domain. These values are used while updating the virtual time for a task and while calculating the fairshare timeslice, as given in Algorithms 4 and 2 respectively.

**Tuning Parameters** The scheduler exposes some of its internal data structures and certain statistical information via the proc filesystem. The location /proc/bvtstat contains information about the number of tasks which got policed, the pid for the task which got policed last, the number of tasks which got run in lieu of having the earliest release-time, etc. /proc/coopstat contains statistics pertaining to the coop poll call usage. The locations /proc/sys/kernel/bvt_sched_period_us and /proc/sys/kernel/bvt_sched_unfairness_us can be used to tune the values for the global time period and the unfairness threshold for coop realtime tasks.

### 3.5 Chapter Summary

We have converted the design and algorithms given in the previous chapter onto a highly efficient and stable implementation. The implementation is based on the latest Linux kernel (2.6.25) and makes full use of all the recent advancements in Linux including high resolution timers, modular scheduler and a fully preemptible kernel. Our scheduler implementation has all the features expected from a modern scheduler including support for task priorities, multi-processor load balancing and task wakeup preemption.
Chapter 4

Xserver Modifications

Our cooperative polling approach is targeted primarily at applications which follow an event driven structure. To evaluate our approach we consider an event driven application that we have developed ourselves from scratch (QStream). In order to demonstrate that this cooperative polling approach is generally applicable we chose a significant existing event driven application, the Unix X11 display server, to be converted to use our cooperative polling approach.

The display subsystem in all Unix based operating systems has a display server which is responsible for rendering graphics onto the display device. The most popular implementation for this display server is the open source one from the X.Org project [6]. This server is an ideal candidate for further demonstrating the applicability of our approach, since its already event based and improving the timeliness of this server will improve the end-to-end timeliness for a variety of applications which rely on it. Applications like our video player rely on the display server to service its frame display requests with minimum tardiness, thus improving the timeliness of the display server will help our video player’s overall tardiness.

This chapter describes the changes that were made to the display server to improve its timeliness by using our cooperative polling approach. We first provide the motivation behind our changes and the design requirements in Section 4.1. Next, Section 4.2 describes our modifications to the core scheduling logic to use a weighted fairshare model. This change is an extra feature and is not a requirement to use our cooperative polling approach. Finally Section 4.3 explains the set of
changes needed to implement low latency event dispatch for the server.

4.1 Requirements

The X11 server (X server) forms the core graphical interface for Unix based operating systems, and is thus crucial for time-sensitive Unix applications (multimedia applications, games) requiring real-time visualization and interaction with the user. The X11 architecture uses a socket to communicate requests between applications and the X server. Communication through the socket is subject to scheduling delays, therefore timeliness of display depends on responsive scheduling. High CPU utilization and the resulting scheduling unpredictability typically occur when the device is constrained, either because the computational power of the device is modest (e.g., multimedia-enabled cell phones) or the demands of the application are high (e.g., HD video). Consequently, applications requiring low latency or precise timing (e.g., video conferencing, multi-track audio processing) may be subject to unpredictable scheduling delays.

In video applications, lack of control over timing can result in video tearing effects that occur when the image on the screen is actually comprised of parts of two separate frames. These artifacts tend to be especially noticeable in scenes with high movement. Applications can reduce or eliminate these artifacts if they have more predictable control over timing.

We have based our changes on the Xorg code base for the Ubuntu [8] Gutsy release (2.1.3.0.0.dfsg-12ubuntu8.4). The original Xserver scheduler selects a client on each iteration of the dispatch loop and then services the selected client until it has run out of requests. This approach is flawed since it allows an aggressive client to dominate over all the other clients, starving them of access to the Xserver. Keith Packard improved upon this by implementing a new Xserver scheduler based on dynamic priorities with fixed timeslices. The new scheduler approximated fairness among the clients by boosting the priority for an idle client and decrementing the priority for a client which overran the timeslice [24]. Although this new scheduler was a significant improvement over the original one, we still felt the need for an accurate fairshare scheduler which accounted for the amount of time each client actually ran for instead of using heuristics. Section 4.2 explains our approach in
more detail.

The server supports the communication of timing requirements via the X Synchronization Extension [14]. The main primitive provided by this extension is called XSyncAwait, which specifies a waiting condition and provides barrier-like functionality. Any X11 requests made by an application subsequent to calling XSyncAwait are deferred until the condition specified becomes true. Although this extension is present in X11 implementations, it is incomplete because the conditions do not include high resolution timers. Support for high resolution timers is necessary, but not sufficient to guarantee timing. The Xserver clients use sockets to communicate their release-time information to the server, therefore the server needs to wake up and read this information from the sockets as soon as the data is written onto the sockets, else it might come to know of a release-time much too late. Section 4.3 explains how we satisfied these requirements.

4.2 Virtual Time based Client Scheduling

As with our changes to the Linux scheduler we want the X11 scheduler to have good timeliness and fair sharing. To change the scheduling logic to an exact fair-share model, we measure the amount of time each client runs for and accumulate the total time in a virtual time parameter. On each iteration of the dispatch loop, we select the client with the lowest virtual time, this ensures that in the long run each of the clients shall receive equal share of the server. To enable some amount of lower level batching of requests, we service each client for a minimum time period, instead of switching after each request. This timeslice parameter is compile-time configurable.

4.3 Low Latency Event Dispatch

This section explains the set of changes needed to incorporate support for low latency dispatch of requests with associated timing requirements. Section 4.3.1 explains the changes needed in the core scheduling loop. Section 4.3.2 explains the mechanism for clients to inform the Xserver of their timing requirements and Section 4.3.3 details the changes needed to incorporate coop_poll into the Xserver.
4.3.1 Conversion to a Reactive Programming Model

The original Xserver implementation is based on an event loop model, wherein there is a central dispatch loop, which selects the next client to be served based on some heuristics. We modified this dispatch/scheduling loop to our reactive event model (see Section 2.2), where we first drain the sockets of all the requests and store them in a per client queue. This de-couples the reading of requests from dispatching of requests, and potentially allows the server to dispatch high priority requests out of order. This ahead of time draining of sockets allows the server to account for the timing requirements of all the clients before making its scheduling decision. Section 4.3.2 and Section 4.3.3 explain the changes needed to complete this feature.

4.3.2 Release-Time Extension

We modified the X Synchronization extension to support high resolution timers, and added a new high resolution timer which reflects the current system time. The Xserver clients can now use the XSyncAwait call to notify the server of any timing requirements with reference to this new timer. We associate the request immediately succeeding the XSyncAwait request with the given timing requirement. For example the Xserver client can specify a specific time at which it wants an image to be displayed, it does this by first making a XsyncAwait call specifying the time of display, and next making the actual PutImage call to display the image.

The modified dispatch loop for the Xserver deals with two kinds of requests - timeout requests and best effort requests. Any request which has an associated timing requirement is classified as a timeout request, all other requests are classified as best effort requests. On each iteration of the event loop, we check to see if there are any expired timeout requests which need to be served, else we pick the client with the lowest virtual time and start serving its best effort requests. All the timeout requests are stored in a priority queue ordered by time and the best effort events are stored in the order in which they are received by the server.

In order for the Xserver to dispatch the timeout requests with minimum latency, it needs to first read the request off its network sockets. Therefore its needs to periodically drain its sockets by reading all the pending requests. We model this
draining event as a repeating timeout request with a fixed periodicity. A shorter
period is better for low tardiness but can be wasteful if I/O is not actually occurring.
The periodicity is compile-time configurable, and is currently set at 7.5 milliseconds. This periodic draining of sockets is only required when the Xserver is not
using coop.poll. The next section explains how the select+coop.poll interface removes the need for this periodic drain event.

4.3.3 Incorporating coop.poll onto the Xserver

As mentioned before the above changes aren't enough to guarantee good timing,
the Xserver like all other time sensitive applications needs the support from the kernel scheduler to meet its timing requirements. Therefore, we modified the Xserver
to use our coop.poll system call to notify the kernel of its timing requirements.
Additionally the Xserver also needs to respond to I/O as soon as possible, and it
achieves this by using the select+coop.poll feature. As explained in Section 2.4,
this feature enables the application to be run as soon as there is any activity on its
interested file descriptors. Therefore we do not need the periodic drain event anymore. The previous statement is only partially true, since the Xserver still needs
to periodically drain the sockets while its running, the kernel only monitors the
file descriptors on its behalf once it yields the CPU. So, the periodic drain event is
cancelled once the Xserver yields and is restarted when the Xserver starts running
again.

The modified Xserver event loop incorporating all the above features is given
in Algorithm 1. For measurement purposes, the event loop measures the tardiness
for each timeout request it dispatches and logs this value.

4.4 Chapter Summary

We made three primary modifications to the Xserver:

- The core scheduling logic was modified to use weighted fairsharing.
- The Xsync extension was extended to support high resolution timers.
- The Xserver uses the coop.poll system call to exchange timing information
  with the kernel.
Algorithm 1 Xserver core event loop

\[\text{CoopPoll}() \{\text{Bootstrap by doing a CoopPoll}\}\]

LoopStart :

while !\text{DispatchException} do

\begin{itemize}
  \item if is\text{EmptyCoopClients()} AND \text{clientRelease-TimeExpired()} then
    \begin{itemize}
      \item \text{client} $\leftarrow$ get\text{EarliestTimeoutClient()}
    \end{itemize}
  \item else
    \begin{itemize}
      \item if active\text{Clients} $\neq$ 0 then
        \begin{itemize}
          \item \text{client} $\leftarrow$ getNext\text{AsapClient()}
        \end{itemize}
      \item else \{\text{Yield via CoopPoll}\}
        \begin{itemize}
          \item cancel\text{ProxyClients()}
          \item CoopPoll()
          \item Drain\text{Sockets()}
          \item GOTO LoopStart
        \end{itemize}
    \end{itemize}
\end{itemize}

startTime $\leftarrow$ Get\text{Time()}

currentTime $\leftarrow$ Get\text{Time()}

while client\text{HasRequests(client)} AND (currentTime - startTime) < client\text{-Timeslice} do

\begin{itemize}
  \item Dispatch\text{Client(client)}
  \item Update\text{ClientVirtualTime(client)}
  \item Flush\text{Output()}
  \item currentTime $\leftarrow$ Get\text{Time()}
\end{itemize}

end while

end while

\end{itemize}

The weighted fairshare scheduling only serves to prevent an over aggresive client from dominating the server and to enable the server to be fair to all its clients. The support for high resolution timers enables a Xserver client to associate a timing requirement with any arbitrary request. As explained above this alone is not enough to ensure that these timing requirements will be met with minimal tardiness, for that we need to use the kernel support in the form of \texttt{coop.poll}. Since the Xserver receives it timing requirements via network sockets, we need to use the combined \texttt{select+coop.poll} system call to ensure we read in these timing requirements as soon as they are sent.
Chapter 5

Performance Evaluation

Our system contains three types of applications - best effort, non-adaptive timing sensitive and adaptive applications. Best-effort applications have no associated timing requirements and have modest to high CPU requirements. Non-adaptive timing sensitive applications have modest CPU requirements along with associated timing requirements. Adaptive applications are both CPU intensive and have associated timing requirements. In this chapter, we will evaluate the timeliness, fairness and the overhead of our approach with a workload which consists of varying mix of all three types of applications. Each workload is run for 5 minutes with some additional time for startup and teardown.

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 task, using the make \(-j\) parameter. The modified Xserver represents a non-adaptive timing sensitive application with associated timing requirements (the clients inform the server of their requirements as explained in Section 4.3.2) and modest CPU requirements. We use our QStream video streaming player to represent an adaptive application. We control the number of adaptive threads by streaming different videos concurrently. Each video has a highly variable bit rate and variable processing requirements. We use this adaptive application because it represents an extremely challenging target for the scheduler. 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 ex-
experiments that show the benefits of incorporating cooperative polling into the X11 server.

We compare our kernel scheduler against two schedulers, the recently introduced, default Linux scheduler called the CFS, and the Linux real-time priority scheduler. We use CFS in its "low-latency desktop mode" configuration. We do not compare against a reservation scheduler because it would either under-utilize the system or require estimation of application requirements, which is problematic for adaptive applications. Also, there are no available implementations of a reservation scheduler for a recent Linux kernel, making meaningful comparisons difficult, since several scheduling-related enhancements have recently been added into Linux (see Section 3.3).

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

**Hardware Setup**  The setup for all the experiments consists of three PCs connected by a LAN, two as video servers, and the other as the 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. Hyperthreading is disabled for all of our experiments, apart from where it is explicitly stated.

**Software Setup**  The kernel is compiled with preemption, high resolution timers and Symmetric Multiprocessing (SMP) enabled. Group scheduling and kernel debugging are disabled. The default parameters for our scheduler are set as 20 milliseconds for the global period, 20 milliseconds for the unfairness threshold (the virtual time differential upto which cooprealtime tasks can be temporarily unfair), coop_slack is set at 2 milliseconds, and io_latency is set to 100 microseconds. The CFS scheduler which ships with Linux 2.6.25.1 is used while comparing our scheduler against CFS.
In Section 5.1, we evaluate our kernel scheduler with a simplified scenario consisting of only adaptive and best-effort applications. Next we add the Xserver also into this mix of applications in Section 5.2. This section evaluates the performance of the X11 server when it uses cooperative polling. The experiments in Section 5.3 establish the timeliness behavior which can be expected from a purely fairshare scheduler. Next, we evaluate the policing mechanism present in our kernel in Section 5.4, and show that only the mis-behaving task is effected and all other tasks are shielded from it. Section 5.5 evaluates the latency improvements for our enhanced nanosleep system call, and finally Section 5.6 measures the application level throughput, CPU, memory, and programming effort overhead for our approach.

5.1 Adaptive and Best-effort Applications

This section evaluates the performance of our system in a simplified scenario wherein the workload only consists of best-effort and adaptive applications. Our workload consists of a varying number of QStream video players and background gcc build tasks. To prevent the Xserver execution from interfering with our results, we disable frame output for our video players. Section 5.1.1 establishes the baseline performance for an adaptive application in the presence of background load, next we extend the experiment to include multiple adaptive applications in Section 5.1.2. We evaluate the scenario for a varying number of players ranging from 2 to 12 players, but due space limitation we are only presenting our results for 8 players in this section. The results for all the other cases are quite similar to the 8 player one.

5.1.1 Baseline Performance

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 performance that can be expected with multiple adaptive applications. We use four concurrent invocations of gcc for the kernel build workload. Figure 5.1 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. In this experiment, the Linux real-time scheduler and our cooperative scheduler have similar tardiness, dispatching all timer events within 1.5ms of their deadline. The cooperative scheduler does marginally better, by dispatching them 0.5ms earlier than the Linux real-time scheduler. The realtime scheduler preempts the currently running task when a realtime task wakes up. The cooperative scheduler avoids this extra preemption cost by incorporating the release-time information into the timeslice calculation.

This number represents close to optimal tardiness given the granularity of events within our QStream implementation. In particular, we have verified this number by measuring event execution times in QStream, and found that the worst-case times for the best-effort events are in the 0.5 ms to 1 ms range.

The Linux CFS scheduler fares poorly in terms of tardiness. Many events are dispatched over 20 ms after their deadline. This tardiness is unacceptable for various audio applications and video conferencing, specially echo cancellation [25, 29]. 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. While real-time priorities provide direct information to the scheduler, we will see that they have other shortcomings that prevent their use in general purpose environments.

5.1.2 Multiple Adaptive Threads

In this experiment, we employ multiple adaptive threads in our workload. We use a mix of 8 adaptive threads and 4 best-effort threads. We run the cooperative scheduler in two scenarios, one in which the threads run in the same thread group, and the other in which they run in separate thread groups (see Section 2.3.2). In the single-group experiment, the adaptive threads aggregate their CPU allocations and use the best-effort event app.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 5.1: Single Adaptive thread

Figure 5.2 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.

We evaluated fairness in this experiment by measuring the CPU usage of the threads. Figure 5.3 shows the cumulative CPU usage of the adaptive threads with the three schedulers. The kernel build jobs run several short compilations, and we do not plot their CPU usage directly. However, the overall usage of these best-
Figure 5.2: Multiple Adaptive thread

effort threads is easy to infer from the time remaining in the graphs, since the total CPU total usage is 100% in all these experiments. Figures 5.3(a) and 5.3(c) show that the Linux CFS scheduler and our scheduler (the multiple groups case) provide comparable fairness to adaptive threads. However, 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 part 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. Figure 5.3(b) shows a significant shortcoming of real-time scheduling in that it starves the best-effort threads.

Figures 5.3(c) and 5.3(d) show the allocation of the adaptive threads using multiple and single thread groups. The aggregate allocation is similar, but the CPU distribution among threads in the single group case is much less uniform. However, the single group case allows threads in the group to cooperatively allocate CPU
to achieve a common goal, such as application-specific fairness. Figures 5.4(a) and 5.4(b) show the video frame rate for the two cooperative polling cases. The single thread group achieves perfectly fair quality (frame rates), while fair CPU allocation with multiple groups delivers highly unfair video quality.

5.2 Xserver Performance

Our previous experiments measured the tardiness of the QStream player. In those experiments, we had disabled frame display to the X11 server. In this section, we run experiments with frame display enabled and measure the performance of the X11 server, modified to incorporate cooperative scheduling. The performance measurements for the QStream players remain un-changed, hence they are not pre-
Figure 5.4: Video quality in frames per second.

Figure 5.4: Video quality in frames per second.

Presented in this section. The workload in these experiments is the same as in the previous section (8 adaptive threads and `make -j4 kernel build`), and we use our cooperative scheduler to run the experiment. As described in Section 4.3.2, the QStream player communicates video frame deadlines to the X11 server using the synchronization extension. We run two experiments that compare the performance of the X11 server with and without cooperative polling. In both cases, we use the synchronization extension to measure the tardiness of X11 frame display events. In the first experiment, we use the synchronization extension to measure tardiness, which is not possible with an unmodified server because it does not know about display deadlines. In the second experiment, the synchronization information is used by the X11 server to supply timer event deadlines to the kernel via `coop_poll`.

In this experiment the Xserver is being run at a higher priority since the CPU
needs for the Xserver exceed the fairshare. The Xserver has very little CPU usage of its own, most of it is due to serving client requests. Therefore if the system has a high number of clients this can cause the Xserver to exceed its fairshare. Like most systems we deal with this problem by assigning the Xserver a higher priority than its client applications.

Figure 5.5 shows the cumulative distribution of tardiness for the frame display events in X11. The results show that without cooperative polling, the tight timing achieved by the QStream adaptive threads (see Figures 5.1 and 5.2) 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. However, 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 also helps X11 even though it does not use cooperative polling directly. The X11 tardiness would be close to the Linux CFS CDF shown in Figure 5.2 if the adaptive threads were running under CFS. When using cooperative polling, the X11 server has tardiness in the 1 ms range, even under the heavy load imposed by this experiment.

5.3 Limits of a Fairshare Scheduler

This section evaluates the best timeliness behavior that can be expected from a purely fairshare scheduler by varying the timeslice parameter (shorter timeslices will result in better timeliness). It attempts to prove the importance of switching to a task in an informed manner taking into account its explicit release-time information, rather than just switching tasks at a very high frequency.

For this experiment we used our faircoop kernel along with 8 independent QStream players with their X11 outputs disabled and 4 background loads. The QStream players were not using the coop_poll system call. The global period for our scheduler was set to 1 ms, the lowest possible. The cumulative tardiness plot is given in Figure 5.6. As shown in the figure, around 37 percent of the readings register zero tardiness, but the worst case tardiness is around 5 ms and the majority of the measurements are below 4 ms. This low tardiness is achieved at the cost of
Figure 5.5: X11 Tardiness

a very high context switch rate of 9348/sec. In comparison the context switch rate for the equivalent (with all the players running in different domains) case using coop.poll was 2211/sec. Thus even at this high context switch rate, the pure fairshare scheduler cannot match the timeliness characteristics of our coop.poll infrastructure.

5.4 Misbehaving Cooperative Applications

The policing mechanism present in our system is designed to shield the cooprealtime tasks from the ill-effects of another mis-behaving cooprealtime task. A cooprealtime task is expected to yield the CPU at the deadline given to it by the scheduler, a mis-behaving task may delay or completely neglect this voluntary yielding. This section aims to evaluate this scenario by using a modified version of the QStream application which deliberately delays its yielding. The actual delay added was randomly chosen from the range of 0..10 milliseconds. The application
adds this delay on a probabilistic basis, where we can tune to the probability of a
delay being added from 0 to 1.

In this experiment we run 8 QStream video players with their X11 output en-
abled and 4 background loads. Each player was executing in separate thread group.
One of the players was mis-behaving with a probability of 0.01 (i.e., it would de-
lay the voluntary yielding 1 percent of the time). Figure 5.7 shows that only the
tardiness for the mis-behaving thread is affected. To prevent the Xserver tardiness
readings from being incorrectly skewed, we filter out the Xserver tardiness mea-
surements which were due to the mis-behaving thread. The mis-behaving thread
gets policed repeatedly causing it to have a high tardiness of around 40 ms, the rest
of the players and the Xserver remain un-affected. Although only 1 percent of the
voluntary yields are delayed, the subsequent policing causes a cascading effect due
to which more than one timeout event gets delayed. Thus a 1 percent probability
of delay causes around 40 percent of the timeout events to get delayed.
5.5 Nanosleep Improvements

As explained in Section 2.5 we have modified the nanosleep system call to use the coop_poll call to improve its accuracy. This section measures the improvement via a simple synthetic benchmark that we wrote. The synthetic benchmark measures the accuracy of the nanosleep call by measuring the latency between the actual sleep duration and the expected duration. We repeat this measurement 10,000 times to measure the average and maximum latency observed.

We ran our synthetic benchmark along with 4 background loads (The background loads are kernel build threads invoked via make -j4). Table 5.1 provides the measurements for our scheduler, CFS and Linux realtime. The average and worst case latency for the enhanced nanosleep is comparable to the realtime case, whereas CFS has approximately 25 times higher latency. This low latency version of nanosleep would be invaluable to applications like echo cancellation and realtime audio effects, which require an accurate sleep mechanism.
Table 5.1: Nanosleep accuracy

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Average latency (usecs)</th>
<th>Max Latency (usecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFS</td>
<td>362</td>
<td>83,423</td>
</tr>
<tr>
<td>Realtime</td>
<td>10</td>
<td>2302</td>
</tr>
<tr>
<td>FairCoop</td>
<td>14</td>
<td>4157</td>
</tr>
</tbody>
</table>

5.6 Overhead

We have established the timeliness improvements gained from our approach in the previous sections, and proven that we need to switch tasks in an informed manner rather than just context switch at a very high frequency to achieve this. In this section, we evaluate the overhead for our approach. First we measure the affect of our modification on the application throughput (Section 5.6.1), and then further measure the CPU (Section 5.6.2) and memory overhead for each of its components - the kernel and the Xserver. We also measure the programming effort needed to convert an existing system to use our approach. The programming effort is quantified by the number of line(s) of code which needed to be changed.

5.6.1 Application Throughput

In this subsection we measure the application level overhead for our approach. One of the easiest ways to measure the throughput for our video player application is by measuring the net frame rate, but this is only true if processing each frame takes the same amount of CPU and memory resources. Therefore we use a specially designed video which has identical frames, each of which takes the same amount of processing.

We compare the overhead with the X11 output enabled and with it disabled. So we run 8 QStream video players and 4 background loads for these cases. When running under the coop.poll kernel all the players are placed under different groups. Figure 5.8 plots the net throughput while running on the coop kernel along with the modified Xserver and on the CFS kernel with an unmodified Xserver. The modified application runs around 10% slower for the X11 output disabled case, and 16% slower when X11 output is enabled. This frame rate slowdown in actual terms is from 10-16 fps respectively. The higher slow down in the X11 enabled
case suggests that our Xserver modifications are a significant contributor to the overall overhead. The rest of the overhead can be attributed to the higher number of context switches incurred in the coop.poll case. The coop.poll version has approximately 1300 (X11 disabled) to 2000 (X11 enabled) more context switches than the comparable CFS version. These extra context switches are essential to the working of our scheduler, and help it meet the timing requirements for the applications. Next we will measure the overhead for each of the individual components of our system, starting with the kernel scheduler.

5.6.2 Scheduler Overhead

We used the open source benchmarking suite LMbench[1] to measure the context switch costs for our scheduler. We compare it to the CFS scheduler which ships with Linux 2.6.25. We compared both the single processor and the SMP version of the kernels. The SMP version was tested with hyperthreading enabled, so that the hardware would emulate two processors.

The benchmark allows you to specify the number of processes, and the data set for each of them. The code size is fixed at 2.7 thousand instructions. We ran the benchmark with data sets of size 0, 32kb and 128kb, while varying the number of processes from 2 to 20. Each run consisted of 50 iterations along with 10 warm up iterations1. The data is presented in the form of graphs in Figures 5.9 and 5.10.

The context switch cost for our scheduler is equal to the CFS scheduler in the uniprocessor case. In the multi-processor case we have an additional overhead of 1 microsecond. This additional overhead might be an artifact of the way the LMbench measures context switch costs. The test instantiates a ring of processes and passes a token around, each task wakes up on receiving the token, and subsequently goes to sleep after passing it along to the next task. The amount of time taken for this hand-off is directly proportional to the context switch cost. This time can be influenced by the distribution of processes among the processors; if the sending and receiving task’s are on different CPUs, one could potentially overlap some of the computation in time, thus reducing the measured costs. The CFS scheduler achieves this kind of a favorable distribution due to its passive load balancing sup-

1For more details refer to the man page for the tool at http://www.bitmovier.com/lmbench/lctx.8.html
Figure 5.8: Application level throughput
Figure 5.9: Context switch costs (Uni-Processor)
Figure 5.10: Context switch costs (SMP)
Table 5.2: Xserver modification: CPU overhead

<table>
<thead>
<tr>
<th>Xserver version</th>
<th>Time Taken (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Xserver</td>
<td>11.69</td>
</tr>
<tr>
<td>Modified Xserver with Coop_poll enabled</td>
<td>13.37</td>
</tr>
<tr>
<td>Modified Xserver without Coop_poll</td>
<td>12.98</td>
</tr>
</tbody>
</table>

Table 5.3: Xserver modification: Memory overhead

<table>
<thead>
<tr>
<th>Xserver version</th>
<th>Peak Memory usage (kB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Xserver</td>
<td>3000</td>
</tr>
<tr>
<td>Modified Xserver with Coop_poll enabled</td>
<td>3000</td>
</tr>
</tbody>
</table>

port, whereas our scheduler would tend to clump them together, with one half of the processes on one CPU and the rest on the other.

5.6.3 Overhead for the Modified Xserver

We measure the throughput overhead for our modified Xserver by running gtkperf [7]. Gtperf is an application designed to test GTK+ performance. We ran each test 100 times and measured the total time taken for all the tests. The tests were run on a uniprocessor machine with our modified faircoop kernel. The measured times are given in table 5.2. The data in the table suggests that the coop_poll enabled version has an overhead of 14.37% and the coop_poll disabled version has an overhead of 11.03%. On further investigation the to the higher CPU usage for the modified Xserver is due to the frequent draining of sockets in our code, the suboptimal use of the glib data structures and the frequent gettimeofday calls. The coop_poll version has an even higher overhead since it drains the sockets at an higher frequency that the coop_poll disabled version, and due to the additional cost associated with copying the file descriptor sets onto the IN parameter for the select+coop_poll call. These copying costs associated with select and select+coop_poll can be alleviated by using the epoll call instead of select(We would also need to combine epoll with coop_poll). Some of the cost can also be removed by using the glib data structures more efficiently.

We measured the memory overhead for our modifications by using a heap pro- filer called massif, which is a part of the valgrind [9] memory debugging toolkit.
Table 5.4: LOC count for Xserver modifications (Including comments and log messages)

<table>
<thead>
<tr>
<th>Change</th>
<th>Line(s) inserted</th>
<th>Line(s) deleted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Converting to an event based model</td>
<td>389</td>
<td>174</td>
</tr>
<tr>
<td>Virtual Time based scheduling</td>
<td>219</td>
<td>62</td>
</tr>
<tr>
<td>Enhanced timers for the Xsync Extension</td>
<td>265</td>
<td>0</td>
</tr>
<tr>
<td>Incorporating coop_poll to the Xserver core event loop</td>
<td>135</td>
<td>0</td>
</tr>
</tbody>
</table>

We ran a set of tests from the x11perf tool on a freshly started Xserver, and measured its peak memory usage. As given in table 5.3 there is no long term memory overhead for our modifications. Our implementation only temporarily allocates extra memory, the extra memory needed per request is de-allocated as soon as the server dispatches the request.

5.6.4 Code Change Overhead

Table 5.4 lists the lines of code needed to modify the Xserver. Most of the change was needed to convert the Xserver to a reactive programming based model and for converting the core scheduling logic to a virtual time based one. The virtual time based scheduling is an extra change, its not a requirement to convert an application to use our coop_poll infrastructure.

Given the approximate line count for the entire Xserver codebase (Xorg 2:1.3.0.0.dfsg-12ubuntu8.4) excluding extensions is around 236,221, the lines of code needed to incorporate coop_poll into the application is around 0.3 percent of the total. This would suggest that its quite trivial to convert an existing event based application to use our coop_poll infrastructure.

5.7 Chapter Summary

The evaluation proves that this approach is capable of meeting the task’s timing requirements to within a sub millisecond range as long as the application is structured in a strict event based manner with short running atomic events. Although this might seem like a very demanding requirement, we feel that this event based model of programming is more suited to a time sensitive application. There are
quite a few existing realtime applications which use this approach.

The approach incurs an application level overhead of 16 percent, most of which is due to a higher number of context switches and an inefficient implementation of the Xserver modifications. The higher number of context switches are un-avoidable since they are needed to ensure fairness and timeliness can both co-exist. Our scheduler does these extra context switches only when required, it dynamically adjusts the context switch rate depending on the system load and timing requirements. Apart from the inefficient Xserver modifications the rest of our system does not incur any extra overhead. Even the programming overhead in terms of the lines of code needed to convert an existing event based application to use our coop_poll infrastructure is quite trivial (around 0.3 percent).
Chapter 6

Future Work and Conclusion

This chapter describes some of the potential directions in which this work can be extended and finally concludes the thesis.

6.1 Future Work

This section details out the possible future extensions for this work. This work has potential application in the area of virtualization and multi-core processing, both of which are recent upcoming trends in the field of computer science. We also describe the minor changes needed to address the shortcomings in our current kernel implementation.

6.1.1 Incorporating coop_poll onto the Hypervisor

One of the key problems while using virtualization for real-time systems is in the loss of timeliness due to the hypervisor’s overly simplistic scheduler. The hypervisor’s scheduler does not take into account the timing requirements for the individual tasks on the guest Operating System (OS).

The most popular approach to this problem has been to run a full fledged Real-Time Operating System (RTOS) alongside the generic OS and always give higher preference to the RTOS when scheduling guests. This approach has been adopted by all major companies like VirtualLogix[5], TenAsys[4] and Real Time Systems[3] in their real-time hypervisors. The drawbacks of this solution are in the additional
need to run a full fledged RTOS, the strict separation of time-sensitive applications and best-effort applications (time-sensitive applications need to be run in a different guest OS), and the in-ability to handle mixed applications (applications which behave as both best-effort and time-sensitive). Moreover a real-time guest OS can quite easily starve the other guest operating systems due to its strictly higher preference.

Incorporating the coop.poll approach onto the hypervisor’s scheduler would help alleviate the problems with current real-time hypervisors. For the approach to work, the guest operating systems will need to honour timing requirements for the applications, the hypervisor’s scheduler is only responsible for scheduling the guest operating systems and not the individual tasks on each of them. The applications would directly inform the hypervisor of their timing requirements via a hypercall, very much similar to the current coop.poll system call and the hypervisor scheduler would take these into account while scheduling the guest OS.

6.1.2 Using STM for Implementing Multi-core QStream

Software Transactional Memory (STM) is a new form of concurrency control mechanism which is quite similar to the database transactional model. It allows you to control access to shared memory from concurrent threads of execution. A transaction is defined as an atomic block of code which reads or writes shared memory, the intermediate states for this atomic block should not be visible to other transactions.

The events in a purely event based system map directly onto transactions, i.e., each event can be treated as a transaction. Doing so greatly simplifies the implementation for a parallelized multi-core version of the same event based application, since events can be evenly distributed across all available processors without bothering about complicated locking issues. The design would also include one central entity responsible for controlling the distribution of these events, and for coordinating work between the various processors. We could leverage this approach to design and build a highly scalable version of QStream video player application for multi-core machines.
6.1.3 Support for Scheduler Groups/Cgroups

The Linux kernel recently introduced some advanced resource containment mechanism like scheduler groups and cgroups scheduler support. Our scheduler currently lacks support for both of them, adding these would help our kernel become a more viable option as compared to standard Linux schedulers such as CFS, which already have support for these features. Resource containment allows you to subdivide the CPU among different user groups / task groups, preventing a range of denial of service attacks.

6.1.4 Passive Load Balancing

Our kernel scheduler currently lacks support for passive load balancing as explained in Section 3.4. Adding this feature would make the load balancing mechanism more effective.

6.1.5 Combining coop.poll with epoll

We have only combined coop.poll with the select interface, it would be useful to also combine it with the two other asynchronous I/O interfaces present inside the Linux kernel - poll and epoll. This would help to incorporate coop.poll into a wider variety of programs.

6.2 Conclusion

In this thesis, we have introduced a novel approach to scheduling time sensitive applications while still maintaining long term fairness among all the applications on the system. Our approach is capable of scheduling both best-effort applications and time-sensitive applications under one unified framework. The time-sensitive applications may be adaptive (adapts it performance based on available resources) or non-adaptive in nature. The primary pre-condition for using our framework is that the time-sensitive application should be structured in an event based model. Our approach aims to reduce unpredictable timing by minimizing involuntary pre-emption, and sharing of timing information with the kernel and other time-sensitive applications.
Some applications have timing needs which are associated with I/O activity, hence we extended our framework to enable such applications to react to I/O with minimum latency. We have implemented a combined interface which allows the application to share its timing information and poll for I/O. The scheduler ensures that the application gets to run either when its given release-time is expired or once it detects some I/O activity.

We have implemented our scheduler in the latest Linux kernel in the form of a separate scheduling class. The Linux scheduler has been recently re-designed into a modular framework with different schedulers co-existing in the form of scheduling classes. To prove the universal applicability of our approach we modified two example applications; a custom developed video streaming solution (QStream) and the X11 display server. They each represent two different classes of applications, QStream is a time-sensitive adaptive application and the display server is time-sensitive but is not capable of adapting. We evaluate our scheduler using a varying mix of all three types of applications - best-effort, time-sensitive adaptive, time-sensitive non-adaptive. Our evaluations show that our approach is capable of meeting sub millisecond timing requirements. This is orders of magnitude better than the CFS scheduler used in mainstream Linux. Our evaluations also prove that its not possible to attain this level of timing by just context switching at a high rate, it is important to switch tasks in an informed manner.

The kernel implementation and the X11 server modification are both open source and are available in public repositories. Git repository for the kernel: ssh://dsg.cs.ubc.ca/home/mayukh/gitrepos/coopkernel.git, Svn repository for the X11 modifications: svn+ssh://dsg.cs.ubc.ca/home/mayukh/svnroot/xorg_qsf_1.3. The custom built video streaming solution is available at: http://QStream.org.
Bibliography


http://www.gnu.org/software/pth/. → pages 8

→ pages 36

the IEEE Real Time Technology and Applications Symposium (RTAS), pages  
434–441, May 2004. → pages 3

[16] 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  

elastic real-time applications. In Proceedings of the Multimedia Computing  
and Networking Conference (MMCN), Jan. 2007. → pages 11

[18] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In  
pages 8

[19] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers,  
R. Fairba irns, and E. 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. → pages 3

pages 2

Performance specifications and metrics for adaptive real-time systems. In  
→ pages 3

[22] 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. → pages 3

65


Appendix A

Pseudocode

This chapter contains pseudocode listings for most of the common functions within our kernel.  

Algorithm 2 Timeslice calculation

**INPUT**: Task, RunQueue

**OUTPUT**: TimeSlice

FairTimeSlice $\leftarrow$ CalculateFairShareTimeSlice(GlobalPeriod, Task)

TimeTillNextReleaseTime $\leftarrow$ GetTimeTillNextReleaseTime(RunQueue)

if FairTimeSlice > TimeTillNextReleaseTime then

TimeSlice $\leftarrow$ TimeTillNextReleaseTime

else

TimeSlice $\leftarrow$ FairTimeSlice

end if

if TimeSlice > MinTimeSlice then

TimeSlice $\leftarrow$ MinTimeSlice \{Limit context switches\}

end if

return TimeSlice

---

1 The actual code for the modified kernel can be found in the following git repository: ssh://dsg.cs.ubc.ca/home/mayukh/gitrepos/coopkernel.git
Algorithm 3 Pseudocode for the scheduler choice function

```
INPUT: RunQueue
OUTPUT: Next task to run
if isEmpty(RunQueue) then
    return NULL
endif
Task ← TaskWithEarliestReleaseTime(RunQueue)
if Task. ReleaseTime > CurrentTime then
    if Task. VirtualTime - MinVirtTime(RunQueue) < RunQueue.UnfairnessThreshold then {Check task’s VT is within fairness threshold}
        GOTO ChoiceDone
    else
        PoliceTask(Task)
    endif
else
    Task ← TaskWithMinVirtTime(RunQueue)
end if
if isCoopRealTime(Task) then
    Task ← ChooseTaskWithinDomain(Task. Domain)
endif
ChoiceDone:
if TaskInPollRunnable(Task) then
    WakeupTask(Task) Task.BreakOut ← TRUE {This flag is used to break out of the select/epoll/poll call}
endif
return Task
```

Algorithm 4 Pseudocode for updating the virtual time for each task

```
INPUT: Task
INPUT: TimeConsumed
if isCoopRealTime(Task) then
    Domain ← TaskDomain(Task)
else
    Task. VirtualTime ← Task. VirtualTime + TimeConsumed ÷ Task. Weight
endif
```
Algorithm 5 Pseudocode for enqueing a task onto the runqueue

INPUT: RunQueue
INPUT: Task
AddToList(RunQueue. List, Task)
if isCoopRealTime(Task) AND TaskInPollRunnable(Task) then
    RemoveTaskFromReleaseTimeQueue(Task, RunQueue.ReleaseTimeQ)
    SetTaskReleaseTime(GetCurrentTime())
    InsertTaskIntoReleaseTimeQueue(Task, RunQueue.ReleaseTimeQ)
else if !isCoopRealTime(Task) AND TaskInPollRunnable(Task) then
    SetTaskAsTempCoop(Task)
    SetTaskReleaseTime(GetCurrentTime())
    InsertTaskIntoReleaseTimeQueue(Task, RunQueue.ReleaseTimeQ)
else if isCoopRealTime(Task) AND !TaskInPollSleep(Task) then
    InsertTaskIntoReleaseTimeQueue(Task, RunQueue.ReleaseTimeQ)
    RemoveTaskFromSleepQueue(Task, RunQueue.SleepQ)
end if
if !isTempDomain(Task.Domain) AND Task.Domain.NumTasks > 1 then {Do not enqueue task, if domain is active already}
    EXIT
else if !isCoopRealTime(Task) AND TaskInPollSleep(Task) then
    SetTaskAsTempCoop(Task)
    SetTaskReleaseTime(GetCurrentTime())
    InsertTaskIntoReleaseTimeQueue(Task, RunQueue.ReleaseTimeQ)
else if isCoopRealTime(Task) AND TaskInPollSleep(Task) then
    RemoveTaskFromReleaseTimeQueue(Task, RunQueue.ReleaseTimeQ)
    SetTaskReleaseTime(GetCurrentTime())
    InsertTaskIntoReleaseTimeQueue(Task, RunQueue.ReleaseTimeQ)
    if !isTempDomain(Task.Domain) AND Task.Domain.NumTasks > 1 then
        EXIT
    end if
end if
if RunQueueEmpty(RunQueue) then
    Task.VirtualTime ← MaxVirtualTime(RunQueue)
else if Task.VirtualTime < MinVirtualTime(RunQueue) then
    Task.VirtualTime ← MinVirtualTime(RunQueue) {Fast forward virtual time}
end if
InsertTaskIntoRunQueue(Task, RunQueue)
Algorithm 6 Pseudocode for Dequeing tasks from the runqueue

```
INPUT: RunQueue, Task
RemoveFromList(RunQueue.List, Task)
if TaskInPollRunnable(Task) then
    EXIT
else if isCoopRealRealTime(Task) then
    RemoveTaskFromReleaseTimeQueue(Task, RunQueue.ReleaseTimeQ)
    RemoveTaskFromAsapQueue(Task, RunQueue.AsapQ)
    if isTempDomain(Task.Domain) OR Task.Domain.NumTasks <= 0 then
        RemoveTaskFromRunQueue(Task, RunQueue)
    end if
if NonCoopSleep(Task) then {Police task, if its not sleeping within coop.poll}
    PoliceTask(Task)
end if
else
    RemoveTaskFromRunQueue(Task, RunQueue)
end if
```

Algorithm 7 Pseudocode for calculating the fairshare timeslice

```
INPUT: GlobalPeriod, Task
CPU ← TaskCpu(Task)
if IsCoopRealTime(Task) then
    Domain ← TaskDomain(Task)
    FairSharePeriod ← GlobalPeriod * Domain.SumOfWeights ÷ CPU.SumOfWeights {Divide the GlobalPeriod in ratio of the weights}
else
    FairSharePeriod ← GlobalPeriod * Task.Weight ÷ CPU.SumOfWeights
end if
```
Algorithm 8 Pseudocode for preemption function

INPUT: RunQueue, Task

\[ \text{CurrentTime } \leftarrow \text{GetCurrentTime()} \]

if Task = CurrentTask then \{ Do not preempt one's own self \}

\[ \text{EXIT} \]

endif

if isRealTime(Task) then \{ Allow a real time task to unconditionally preempt \}

\[ \text{RescheduleTask(CurrentTask)} \]

endif

if isCoopRealTime(Task) AND ( TaskInPollRunnable(Task) OR TaskInPoll-Sleep(Task) ) then

if isCoopRealTime(Task) then

\[ \text{Delta } \leftarrow \text{CurrentTime } - \text{Task.TimeSliceStart} \]

if Delta \( \geq \) RunQueue.MinimumTimeSlice then

\[ \text{RescheduleTask(CurrentTask)} \]

else

\[ \text{TimeLeft } \leftarrow \text{RunQueue.MinimumTimeSlice } - \text{Delta} \]

\[ \text{CancelTimer(RunQueue.Timer)} \]

\[ \text{NewReleaseTime } \leftarrow \text{CurrentTime } + \text{TimeLeft} \]

\[ \text{ArmTimer(NewReleaseTime)} \]

endif

else

\[ \text{TimeLeft } \leftarrow \text{Task.TimeSliceEnd } - \text{CurrentTime} \]

if TimeLeft \( > \) RunQueue.CoopSlack + RunQueue.IOLatency then

\[ \text{CancelTimer(RunQueue.Timer)} \]

\[ \text{NewReleaseTime } \leftarrow \text{CurrentTime } + \text{RunQueue.IOLatency } + \text{Run-Queue.CoopSlack} \]

\[ \text{ArmTimer(NewReleaseTime)} \]

endif

endif

endif
Appendix B

Linux Kernel Programming

Linux kernel programming is tough. In this appendix we will try to list some of the common pitfalls and debugging techniques which we employed in the course of our development. The scheduler's unique importance makes the extremely challenging and interesting at the same time. The scheduler is responsible for choosing which task to run, and for how long; any error in this code causes the entire system to lockup. Section B.1 lists the debugging techniques used, Section B.2 explains the constraints involved in implementing system calls in Linux and Section B.3 explains the need for a precise process accounting feature in Linux.

B.1 Debugging Techniques

This section explains few of the debugging techniques and their associated constraints.

*Using printk*  Printk is the kernel's equivalent of the userland printf function, this function has been designed to be highly resilient, and can be used from almost any context - task, interrupt, software interrupts etc. One of the few places where its un-safe to use a printk is while holding the runqueue lock. The printk function will occasionally try to wakeup klogd, which is the kernel's logging daemon. The wakeup function will try to acquire the runqueue lock before adding the task to the runqueue. This leads to a deadlock, since spinlocks in Linux are not recursive.
**Debugging the Kernel** Whenever the kernel encounters an error, it dumps out debug messages onto the console before halting the Central Processing Unit (CPU). Most of the time, these messages will not make it to the kernel log, since the entire system hangs up before the kernel daemon has a chance to write them out. The standard method to debug such a problem would be use a serial cable to capture the console output, but sometimes it's not possible to access the developmental machine physically or the machine lacks a usable serial port. The solution in such cases is to use very helpful kernel feature called netconsole, which allows you to capture the console output via ssh. This feature captures most of the console output, apart from messages very early in the boot sequence when the network interface has not been initialized yet.

**Debugging Lockups** Kernel lockups refer to a system state where the system becomes unresponsive to any input. It is quite common to encounter a few of these while doing kernel development. Your first line of defense would be the magic system request key, this unique key combination allows you to do some basic troubleshooting like - dumping the register state, rebooting the machine safely, etc.

The next step would be to cross check your locking code, since most kernel lockups are due to some form of incorrect locking. The best way to catch such errors is to enable the extensive lock debugging infrastructure built into the Linux kernel. The lock dependency engine which is better known as LockDep can potentially catch all theoretical locking errors. The engine does not wait for the locking error to occur, but forewarns you if an error is theoretically possible.

The LockDep engine requires extra annotation information for each function which releases or acquires locks. Sometimes such information maybe missing, or the lockup might not be due to a locking issue. In such cases, the Linux kernel contains a Soft lockup detection feature, which detects if the CPU is locked up, and subsequently prints out useful debug information (including a full stack trace). Soft lockups are defined as lockups with interrupts enabled. The way this works is that on each timer tick, the kernel checks if the system is progressing forward or not. Since any healthy system has a steady flow of interrupts, the kernel uses this count as a measure of forward progress. If the kernel detects that no progress has been made for some pre-defined time period then it prints out a warning along with
the necessary debug information.

Hard lockups are defined as lockups with interrupts disabled, and in such cases the soft lockup detector is rendered useless. The Linux kernel also includes a Non-maskable interrupt (NMI) watchdog, which should be functional on most modern processors. The NMI is a special interrupt which cannot be disabled via the usual interrupt disabling instruction used within the kernel. If any CPU in the system does not execute the periodic local timer for more than 5 seconds, then the NMI handler print out a warning along with necessary debug information.

B.2 System Call Constraints

System calls in Linux pass their arguments via registers, thus the number of arguments is constrained by the number of free general purpose registers. For the x86 architecture this number is fixed at 6. While designing our modified system call combining coop.poll with select, we decided to increase the number of arguments for our system call to 7. This led to highly unreliable behavior, since the 7th argument got assigned random values. This bug was quite hard to trace since the kernel or the compiler is unable to warn you of this behavior. Hence we should take this constraint into account while designing/porting a system call onto different architectures.

B.3 The Need for Precise Task Accounting

While implementing our scheduler we were relying on the kernel task accounting code to monitor the cpu usage for each task. The task accounting code in Linux is statistical in nature, such that on every timer tick the kernel wakes up and charges whichever task was running at that time. If the system was running kernel code at that time then the system's usage counters are incremented. This method greatly simplifies the task accounting code, but is fraught with some serious drawbacks. Our scheduler can cause multiple context switches within a single timer tick interval. In this case the periodic sampling would only charge the processes which were running on the tick interval boundary, thus reporting incorrect values for cpu usage, etc. We fixed this problem by doing task accounting on each context switch. We calculate how long each task ran and then charge it for this time. Currently this
method does not distinguish between system and user time, since task accounting is only done on context switch boundaries neglecting user level to kernel crossings. A more complete solution would be to use the Precise Process Accounting (NMI) project from Motorola which aims to remove these shortcoming from the Linux kernel.[2]
Appendix C

Notes on Publication

Parts of chapter 2 and chapter 5, section 1.1 and 1.2 of chapter 1 and section 4.1 of chapter 4 are taken from our OSDI 2008 submission by Charles Krasic, Ashvin Goel, Anirban Sinha and me. All the writing for the paper was done by Krasic and Goel.

Some of the co-authored thesis sections have been slightly updated to reflect the current state of the system. All authors have given permission to include material from the co-authored paper in this thesis.