Achieving Predictable Timing and Fairness Through Cooperative Polling

by

Anirban Sinha

B.Tech., Institute of Engineering and Management, Kolkata, India, 2002

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

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

November 2007

Abstract

Time-sensitive applications that are also CPU intensive like video games, video playback, eye-candy desktops etc. are increasingly common. These applications run on commodity operating systems that are targeted at diverse hardware, and hence they cannot assume that sufficient CPU is always available. Increasingly, these applications are designed to be adaptive. When executing multiple such applications, the operating system must not only provide good timeliness but also (optionally) allow co-ordinating their adaptations so that applications can deliver uniform fidelity.

In this work, we present a starvation-free, fair, process scheduling algorithm that provides predictable and low latency execution without the use of reservations and assists adaptive time sensitive tasks with achieving consistent quality through cooperation.We combine an event-driven application model called *cooperative polling* with a fair-share scheduler. Cooperative polling allows sharing of timing or priority information across applications via the kernel thus providing good timeliness, and the fair-share scheduler provides fairness and full utilization.

Our experiments show that cooperative polling leverages the inherent efficiency advantages of voluntary context switching versus involuntary preemption. In CPU saturated conditions, we show that the scheduling responsiveness of cooperative polling is five times better than a well-tuned fair-share scheduler, and orders of magnitude better than the best-effort scheduler used in the mainstream Linux kernel.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would first like to thank my supervisor, Dr. Charles 'Buck' Krasic for his tireless commitment, guidance and advice as and when I needed it. I feel fortunate to have worked under him. I am also extremely grateful to Dr. Ashvin Goel from the University of Toronto for unofficially being my co-supervisor and providing me with excellent suggestions and help from time to time. I must also thank Dr. Norman Hutchinson for being my second reader and providing me with some very useful comments and recommendations. I would also like to take this opportunity to thank all the anonymous reviewers of our papers and persons who gave us valuable feedback at the SOSP poster session which helped to shape up this work. I thank the University and in particular, the Computer Science Department for giving me the opportunity to work and learn during the past two years. A special thanks goes to Robert Love for clarifying my doubts during my kernel development work.

Many thanks to Geoffrey Lefebvre and Andy Warfield for their suggestions and help during the course of my work. I am grateful to Brendan Cully for his help during my benchmarking. I express my heartfelt gratitude to my two wonderful friends, Meghan and Abhishek for being there for me anytime, always. I also thank Kan for his friendship, support and help all the time. I would miss his funny comments and jovial nature that kept the lab alive. Thanks also to my other lab mates: Andrei, Gang, Mike Wood, Mike Tsai, Gitika, Mayukh, Cuong, Camilo and Brad for making it a wonderful place to work. Thanks to all the CSGSA folks with whom I worked as a part of the graduate student society body. I also thank my friend Satyajit Chakrabarti for encouraging me to apply to UBC.

Finally I thank my family and specially my grand-pa and grand-mom for their endless love and for being there for me, always, with everything, no matter what.

Anirban Sinha

The University of British Columbia
November 2007

To the loving memory of my beloved grand-father, *Late. Mr. Bimal Kumar Mitra* who is not among us to see this day.

To my family who are always there for me, no matter what.

Statement of Co-Authorship

Section 3.1 of Chapter 3 is taken from our co-authored MMCN paper [23]. Krasic was the lead author in the paper and all the writing for the paper was done by him alone.

Parts of Chapter 1, Chapter 4 and Section 2.1 of Chapter 2 are taken from our co-authored Eurosys 2008 paper by Charles Krasic, Ashvin Goel and me. Krasic was the lead author in the paper and all the writing was done jointly by Krasic and Goel.

Some parts of the thesis abstract and Chapter 1 are largely based on our co-authored SOSP 2007 poster abstract [39]. I was the lead author for the co-authored poster; however, certain portions of the abstract were written and edited by Krasic and Goel.

The co-authored thesis sections include only slight modifications from the co-authored poster/paper. All authors have given permission to include material from the co-authored paper in this thesis.

Chapter 1

Introduction

1.1 Motivation

The difference between traditional real time and commodity operating systems has become increasingly blurred over time. On one hand, the demand for using more and different kinds of multimedia capable *time sensitive*[1] applications on a commodity operating system has gone up. On the other hand, the need for running non-real time, best effort applications on an otherwise dedicated real time embedded core has increased. The important challenges are therefore to provide predictable timeliness for the time sensitive applications in a system running other best effort applications. At the same time, it is important not to compromise the overall throughput of the system significantly. Often, these time sensitive applications are also CPU intensive. Since they are targetted at running under diverse hardware platforms, they are also adaptive in nature. When multiple adaptive time sensitive applications run on a single system, it is important to provide mechanism that can facilitate coordinated adaptation, so that the fidelity is stable across all applications.

The classic approach to best effort scheduling is the well known multilevel feedback queue scheduling algorithm [5, 34]. In this work, we focus on the Linux kernel that also uses the above scheduling algorithm for scheduling tasks[7]. The salient features of the algorithm is as follows: The algorithm uses priorities for each task. Within each priority level, each task is scheduled according to the first-in, first-out (FIFO) principle. It then tries to classify applications either as CPU intensive or IO intensive based on how

[1] In this thesis, we use the words *time sensitive* and *realtime* interchangeably to mean one and the same thing - soft realtime, primarily multimedia applications.

long and how often the task spends time sleeping. It then prioritizes the ones it thinks are IO intensive over the CPU intensive ones. This classification is workload dependent and dynamically varies with time. This best effort strategy does not ensure predictable timeliness as far as the realtime applications goes. Further, the heuristics break when workloads are both IO and CPU intensive in nature. Examples of such workloads include multimedia streaming applications, databases, most web servers, etc. Further, preemptive kernel scheduling policies introduce unpredictable timing behavior, especially in the presence of issues such as priority inversion, lock preemption, livelock, deadlock, etc. As a result, applications that are time-sensitive may not run at the desired time, get the desired allocation, or may miss important deadlines, and these problems are hard to understand or debug. Preemption in an uninformed fashion also interacts badly with the adaptation logic in adaptive multimedia applications, resulting in their poor adaptation.

In general, scheduling for general purpose systems is an issue of tradeoff between responsiveness and throughput. Traditionally kernel designers have favored throughput over responsiveness in their algorithm design. Our approach tends to find a better balance between the two. Also, it is our belief that unpredictable timing is more aggravated by the fact that time-sensitive applications are not aware of the timing requirements of other time-sensitive applications. If such applications were aware of the timing requirements of other applications, they may be able to accommodate the others, and vice versa. Furthermore, by sharing timing information, these applications may be able to more effectively adapt their behavior while still preserving timeliness during overload. For realtime systems, previous approaches have relied on CPU time reservations for real time processes to ensure proper timeliness. However, this approach has several disadvantages for a general purpose operating system. First, reservations result in potential underutilization of the system. To ensure proper timeliness, users are often tempted to over allocate CPU for real time tasks. Over allocation can adversely affect the throughput of the system. Secondly, in a commodity environment, with a mix of real time and best effort applications, the system is overloaded most

of the time. Hence, reservation is not a feasible option.

This thesis describes a novel scheduling algorithm based on a combination of cooperative polling and a fair share scheduler that enables time-sensitive and best-effort tasks to co-exist in a tightly unified framework. In doing so, we remain careful so as not to be unfair to best effort applications in terms of their throughput. We also avoid using reservations for real time tasks. In this present work, we have focus more on the real time tasks and not so much on the best effort tasks. Evaluation of our kernel implementation for various kinds of best effort workloads is left as a future work for this thesis.

1.2 Our Approach

Cooperative polling uses a new system call called `coop_poll` that time-sensitive applications use to share event information such as deadlines and priorities with the kernel. Across applications, intra-application event dispatchers use this shared information to determine appropriate times to yield, and optionally to achieve co-ordinated quality adaptations. By yielding in an informed fashion, applications minimize involuntary preemption thus achieving more predictable timing. Our kernel scheduler uses the information available from `coop_poll` to provide better responsiveness to applications that use `coop_poll`, hence providing an incentive for such cooperation. Besides rewarding cooperation among time-sensitive tasks, our model which combines fair-share scheduling, enables two other significant contributions: a) we use preemptive scheduling to prevent the possibility of `coop_poll` being abused (either intentionally or otherwise) to gain unfair advantage, and b) unlike existing approaches that have attempted to integrate conventional real-time scheduling algorithms into general-purpose operating systems with limited success, our approach allows time-sensitive and best-effort tasks to co-exist in a tightly unified framework.

1.3 Research Objectives

In this work, we present a novel starvation free scheduling algorithm in the kernel that meets three important requirements for supporting adaptive time-sensitive applications in a general-purpose OS: a) good timeliness: tasks must receive predictable and low latency execution even under CPU saturation conditions, b) fairness: long term throughput of all tasks (or fidelity of the time-sensitive tasks) should be assured, avoiding starvation, and c) full utilization: unnecessary idle periods should be avoided (work conservation). We meet these requirements by combining an event-driven application model called *cooperative polling* with a fair-share scheduler. Cooperative polling allows sharing of priority information across applications via the kernel. It thus provides good timeliness and the fair-share scheduler provides fairness and full utilization.

1.4 Contributions

This thesis makes the following contributions:

- Design of a novel cooperative polling scheduling algorithm that allows adaptive time sensitive applications to coexist along with best effort tasks in a commodity operating system environment satisfying the objectives above.
- Implementation of the algorithm in the Linux 2.6.20 kernel.
- Modification of the existing adaptive media streaming application Qstream [21, 24] at the user level to benefit from the new `coop_poll` system call infrastructure.
- Performance evaluation of the algorithm showing that our algorithm performs many times better under overloaded conditions than the stock 2.6.20 Linux kernel scheduler.

1.5 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 describes our scheduling algorithm and the cooperative polling mechanism. It goes on to describe some of the details of our kernel implementation. It further provides a brief description of the user level prototype of our scheduling algorithm from which the kernel design followed. Chapter 3 focuses on the performance evaluation of our implementation in the kernel against a multimedia workload. It also shows how our algorithm performs many times better than the stock Linux kernel heuristics. We review some of the related work in this space in Chapter 4. Chapter 5 discusses some of the most important future works in this area and then summarizes the work and finally concludes.

Chapter 2

Algorithm Design and Implementation

In this chapter, we discuss the algorithms and design principles behind our scheduler implementation. We also give some details about our actual kernel implementation as we describe these algorithms. In the beginning, we describe our event driven programming model at the user level and how we modify this model to use our new `coop_poll` primitive. This is only described in the algorithmic level since its actual implementation was already available to us prior to starting this work. Later on, we describe our `coop_poll` primitive and our kernel scheduling algorithm in detail. We describe how `coop_poll` has been integrated into the whole scheduling algorithm. We also give important details of our kernel implementation of `coop_poll` and our overall scheduling algorithm.

2.1 Cooperative Polling

The cooperative polling model aims to provide support for applications that require timeliness and that adapt during overload. This model is event-based and makes certain assumptions which we collectively refer to as reactive programming. The distinctive aspect of our model is that it enables inter-application cooperation by sharing event information with the kernel and across applications.

We discuss these aspects of the model below.

2.1.1 Reactive Programming

The cooperative scheduling model is founded on the principles of reactive programming described below, which are assumed by the design of cooperative scheduling outlined in the subsequent section.

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

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

3. Events should avoid long running computations.

Although not universal to all reactive models, the implementation of the reactive model in the Qstream application executes events atomically. This lack of preemption frees the programmer from the need to use locking and synchronization primitives required in multi-threaded programs.

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

Figure 2.1 lists the key primitives in our scheduling model. The application calls `submit` to submit an event for execution. To initiate dispatching of events, the application calls `run`, which normally runs for the lifetime of the application. The application must submit at least one event before calling run, and it calls `stop` from within one of its events to end the dispatching of events. The application can also call `cancel` to revoke an event it had previously submitted.

```
submit(EventLoop l, Event e);

cancel(EventLoop l, Event e);

run(EventLoop l);

stop(EventLoop l);
```

Figure 2.1: Basic event API

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

Figure 2.2: Event type definition

2.1.2 Events, Event Loop and Scheduling Policy

The scheduling policy component of our model aims to provide predictable timing by reducing scheduling latency.

Figure 2.2 shows the type definition of an event. An application specifies each event as either a best-effort (also sometimes referred to as an *asap* event) or a deadline event. The `callback` field specifies the function that will handle the event and any data arguments to be passed. The `deadline` field specifies an absolute time value. Deadline-based events are not eligible for execution until the `deadline` time has passed. Throughout this thesis, we use the above notion of deadline events consistently. This notion of *deadline* in our work is opposite to the common understanding of deadlines where it is used to mean the time at which a work should finish. However, we prefer to stick to our former definition. Once eligible, deadline events take priority over all best-effort events.

The `priority` field is used by best-effort events. It is up to the appli-

cation to use priorities to control execution order. For example, in a video application it is important to keep sound uninterrupted because users are sensitive to audio glitches. Hence, the application would assign a high priority to events related to servicing the audio output device. When best-effort events have the same priority, the deadline field is overloaded and used as a secondary sort key for ordering best-effort events.

Figure 2.3 shows the the event dispatch algorithm (or simply the event loop) used by Qstream. The deadline and best-effort events are stored in the `deadline_events` and the `best_effort_events` priority queues, and the `submit` and `cancel` operations are realized by insertion and removal from these queues. These operations are idempotent and have no effect if the event is already submitted or canceled, or is a null event.

The dispatcher simply services *all* events as provided by the application even when events arrive faster than they are dispatched. This approach can cause the queue fill levels to increase, perhaps unboundedly, if overload is persistent (e.g., the CPU is just too slow for the given application). However, Qstream chooses this approach because it makes the scheduling policy and the dispatcher simple and predictable. Also, Krasic, while designing Qstream believed that effective overload response requires application-specific adaptation. The QStream video client implements such adaptation by reducing the generation of certain events and invoking `cancel` for some existing events to skip the less important steps (e.g. of video decoding) as necessary to maintain timeliness [23].

2.1.3 Inter-application Cooperation

We improve kernel scheduling performance and enable cooperation between time-sensitive applications with one new primitive, `coop_poll`. This function voluntarily yields the processor and facilitates sharing of event information between the kernel and time-sensitive tasks. The detail algorithmic description of the user level prototype of `coop_poll` is discussed in Section 2.2.1 and that of the kernel implementation in Section 2.3.2. In this section, we describe `coop_poll` in the context of the event dispatcher. The

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

yield(EventLoop l) {
    if(q_not_empty(l.deadline_events)) {
           sleep_until_next_deadline;
    } else{
           l.stop = True;
    }
}
```

Figure 2.3: Event dispatcher algorithm (or the event loop)

primitive `coop_poll` takes one IN and one OUT parameter. The IN parameter specifies the most important deadline and best-effort events in the current thread. These values are used to wake up the thread at its next deadline, or when its best-effort event has the highest priority among all threads within a cooperation group. The concept of cooperation group is further explained in Section 2.3.1 when we discuss scheduling domains.

When the `coop_poll` call returns, the OUT parameter is set to the most important deadline across all other threads and the most important best-effort events within the current task's cooperation group. This information is used by the current thread to yield the processor as well as by the kernel to decide its overall scheduling policy.

Figure 2.4 shows the use of the `coop_poll` call in a modified `yield` function that enables inter-process cooperative scheduling. The `run` routine remains unchanged from Figure 2.3. This `yield` function is designed so that events are executed across threads in the same order as events in the single-threaded dispatcher function shown in Figure 2.3.

The first two arguments in the call to `coop_poll` export the thread's own most important deadline and best effort events. To enable sharing, we add two proxy events to the event loop state, `coop_deadline_event` and `coop_best_effort_event`, that act on behalf of other applications. The deadline and priority values of these proxy events are set by `coop_poll` to reflect the most important deadline and best effort event of all the other applications. After the `coop_poll` call, the proxy events are submitted to their respective event queues in the current thread. The callback function for these events is set to `yield` so that the current thread yields voluntarily to other applications in the `callback_dispatch` routine shown in Figure 2.3. The `cancel` calls at the beginning ensure that the event queues contain only events internal to the current process. This in turn prevents yield from spinning where a thread transitively yields to itself.

```
yield(EventLoop l) {
    cancel(l, l.coop_deadline_event);
    cancel(l, l.coop_best_effort_event);
    if (q_non_empty(l.deadline_events) ||
        q_non_empty(l.best_effort_events)) {
struct inParam in  =
{
.dead_ev = q_head(l.deadline_events),
.best_ev = q_head(l.best_effort_events),
};
struct outParam out =
{
.dead_ev = NULL,
.best_ev = NULL,
};
        // coop_poll sleeps until next deadline
        coop_poll(in, &out);

        l.coop_deadline_event.callback =
        l.coop_best_effort_event.callback = yield;

        submit(l, l.coop_deadline_event);
        submit(l, l.coop_best_effort_event);

    } else {
        l.stop = True;
    }
}
```

Figure 2.4: Dispatcher support for inter-application cooperation

```
struct SharedMemoryArea {
       struct mutex_type m;
       struct CoopSlot Slots[MAX_SLOTS];
       //...

 };
```

Figure 2.5: Definition of the shared memory area

2.2 The User Level Prototype

In this section, we describe briefly the design of the prototype of our scheduling algorithm at the user level. The prototype was available to us prior to implementing our algorithm in the kernel. Its main purpose was to give us an idea about how well (and if at all) the algorithm would work when we implement it in the kernel. We describe the user level prototype here because, (a) our kernel implementation initially was largely inspired and followed from the user level design. (b) in Chapter 3, we describe the results we got from benchmarking our user level prototype. We then later compare our kernel implementation results with the prototype results and show that the kernel outperforms the later in terms of efficiency. We refer to our user level algorithm to analyze some of these results.

In order to model the cooperation between the Qstream applications (without the support of the kernel), the prototype uses a user level shared memory file, shared between all the Qstream applications in a cooperation group for exchanging event information. Each individual Qstream application within a cooperation group owns a specific slot in this shared memory area where it writes its own deadline and best-effort event information. Figure 2.5 shows the structure of the shared memory area. The mutex p is used to protect the area from concurrent accesses. However, note that since in our cooperative regime, any one application runs at one time, the mutex contention will be light, if there is any.

Figure 2.6 describes the data structure representing a single slot in the shared memory area reserved for a specific task in that cooperation domain.

```
struct CoopSlot {
     pid_t id;
     TimeVal timeout_deadline;
     boolean have_asap;
     int     asap_priority;
     TimeVal asap_deadline;
     pthead_cond_t yield_cond;
     boolean sleeping;
     // other statistics gathering parameters ...
};
```

Figure 2.6: Definition of a single slot shared memory area

The process ID of that task is pid. The important attributes of this data structures are the information pertaining to the deadline event (the deadline itself), the best effort event (priority and deadline) and a flag value that shows whether that application has outstanding besteffort events (the `have_asap` field). One another important attribute is the `yield_cond` attribute. This is a condition variable that is used to implement the actual yielding logic. In the next section, we describe the `qsf_co_op_yield`() function where we explain how this is done. Lastly, the boolean `sleeping` variable is used to denote whether the application owning that slot is sleeping (due to voluntary yielding).

In the next section, we describe the yielding logic as implemented in this prototype.

2.2.1 The User Level `qsf_co_op_yield()` Routine

Figure 2.4 shows the general cooperative yielding mechanism implemented in an event loop that uses the kernel `coop_poll` primitive. In the absence of the kernel primitive, the `qsf_co_op_yield()` function roughly does the same operations that a kernel `coop_poll` implementation would do. The algorithm used by the function `qsf_co_op_yield()` is shown in Figure 2.7.

The `qsf_co_op_yield()` routine takes two IN arguments. `my_slot` is actually a pointer to the current task's own slot in the shared memory

```
qsf_co_op_yield (CoopSlot my_slot, CoopSlot ext_slot) {
      lock_mutex(SharedMemoryArea.m);
      export_deadline_event_info(my_slot);
      export_best_effort_event_info(my_slot);
      my_slot.sleeping = true;
      wake_up(ext_slot.yield_cond);
      cond_wait(my_slot.yield_cond, SharedMemoryArea.m);
      my_slot.sleeping = false;
      import_deadline_event_info();
      import_best_effort_event_info();
 }
```

Figure 2.7: User-level coop yield

area. `ext_slot` points to the slot belonging to the external task to whom the current task wants to yield. This decision is made based upon the information imported after the current task wakes up. We discuss this later when we discuss the import routines.

The mutex is locked before the shared memory area is modified. Before cooperative yielding, the function writes its most important deadline and best effort event information into its own slot. This is done so that other external tasks within the same cooperation group may use it to decide the most appropriate task to yield to. In the routine, this is done by calling the `export_deadline_event_info` and the `export_best_effort_event_info` functions. These functions are also responsible for calling `cancel()` to cancel external events as described in Section 2.1.3. Note that, if there are no outstanding best effort events, the current slot's `have_asap` attribute remains `false`. Similarly, if there are no deadline events, the current slot's `timeout_deadline` attribute is set to all zeros.

Next, the routine sets the current task's status to sleeping and wakes up the external task by signaling through its condition variable (`ext_slot.yield_cond`). Subsequently, the routine puts the current task to sleep by invoking `cond_wait` on its own condition variable. Thus, at the end of this statement, the new external task is now active and running and the

old task is inactive and sleeping.

When the current task wakes up from its cooperative yield, it sets its sleeping attribute to `false` and imports the external deadline and best effort event information from the shared memory area. This is done by the `import_deadline_event_info`() and `import_best_effort_event_info`() functions. Each of these two functions iterate through all the slots in the shared memory area, except its own, and finds the most important (the earliest) deadline event and the most important best effort event (determined by the {priority,deadline} dual key). It then schedules the external proxy events as has been described in Section 2.1.3.

The routine `qsf_co_op_yield()` models a pure (and ideal) cooperative scheduling algorithm to reasonable accuracy. In Section 3.8.1, we analyze the performance of this algorithm.

2.2.2 The User level Fairshare Algorithm

The earlier section described how the cooperative scheduling algorithm was modelled in the user level. In this section, we describe how we model the fairshare algorithm in the user level before implementing it in the kernel.

In order to model the algorithm, the import functions described in Section 2.2.1 needed to be changed, keeping the rest of the `qsf_co_op_yield`() routine the same as before. Two separate import routines were actually replaced by single one, `qsf_co_op_import`() which we shall describe a little later. The model also introduces a new attribute called `virtual_time` into the `CoopSlot` structure shown in Figure 2.6. This new attribute keeps track of the virtual times of each of the cooperative tasks. Hence, the `export_best_effort_event_info(my_slot)` now writes the `virtual_time` information for the current task into the shared memory area. The concept of virtual time is discussed in the next section.

The Concept of Virtual Time

The use of virtual time in designing multimedia-real time schedulers is an old idea [6]. Virtual time is a technique used by many of these schedulers to

evenly share the CPU among all the runnable processes. In our work, we use one particular interpretation of virtual time. Briefly speaking, virtual time measures the CPU time consumed by a process. It increases monotonically as the task executes on a CPU. The increase is in exact correlation with the wall clock time. When a task is context switched out, its virtual time no longer increases. Thus, the amount of virtual time charged to a process is an indication of how long the process ran. The only exception to this rule is when the task sleeps and is no longer in the runqueue. When the task wakes up again, we assign its virtual time equal to the minimum virtual time of all the runnable tasks. In this work, the term *borrowing* is used to refer to this assignment[2]. In this specific case, borrowing is only performed from among the tasks in our scheduling domain (i.e., the tasks that are cooperating with each other through the shared memory area). The borrowing mechanism ensures that tasks that sleep can not accumulate CPU entitlements that would subsequently allow them to starve other tasks. This *use it or loose it* approach is an elegant method of accommodating the sporadic requirements of IO bound tasks. We use this *same* concept of virtual time both in the user level prototype implementation as well as our main kernel scheduler implementation.

The `qsf_co_op_import()` Routine

Figure 2.8 shows the basic import algorithm used by the user level fairshare model. It takes one single parameter, the reference to the `EventLoop` structure. The purpose of this routine is to first iterate across all the slots in the shared memory area and find the slot with the smallest virtual time *and* having outstanding best effort events. For those slots with no outstanding best efforts, the target is to find the slot with earliest deadline.

This is an $O(n)$ algorithm where n is the number of slots. This is acceptable for prototype implementation as n is very small. In our case, we do not use more than 12 players. Hence, we use no more than 12 slots.

[2]The readers should not get confused with other different meanings of the term *borrowing* used in the related literature (e.g., the borrowed virtual time algorithm [10]).

```
qsf_co_op_import(EventLoop l)
{
   i=0;
   smallest_vt_slot = 0;
   earliest_dead_slot = 0;
   N = find_num_active_slots();
   Arr = l.SharedMemoryArea;
   while (i < MAX_SLOTS) {
        curr_slot = Arr.Slots[i];
        if (curr_slot == l.my_slot) goto skip;
        if (isEmpty(curr_slot)) goto skip;
        if(curr_slot.have_asap)
                find_slot_with_smallest_vtime(curr_slot,
                                              smallest_vt_slot);
        else
                find_slot_with_earliest_dead(curr_slot,
                                             earliest_dead_slot);
        skip:
                i = i + 1;
   }

   fair_share_period       =  max(l.global_period / (2 x N),
                                  l.min_timeslice);
   fairshare_deadline      =  now + fair_share_period;
   external_event_deadline =  earliest_dead_slot.timeout_deadline;

   if (external_event_deadline < fairshare_deadline)
        l.coop_deadline_event.deadline = external_event_deadline;
   else
        l.coop_deadline_event.deadline = fairshare_deadline;

   submit(l.coop_deadline_event);
   submit(l.coop_best_effort_event);

   if (l.my_slot.have_asap != TRUE)
        my_slot.virtual_time = smallest_vt_slot.virtual_time;
}
```

Figure 2.8: The import routine for the user level fairshare model.

Next, the algorithm calculates the fairshare period according to the Equation 2.1.

$$timeslice = max(sched\_granularity/(2 \times N), min\_timeslice) \qquad (2.1)$$

where N is the number of players and the scheduling granularity is either a constant or is tunable by the user. `min_timeslice` is a constant in our algorithm that determines the smallest timeslice a process can get. This is necessary so as to limit the number of context switches and prevent live locking. Typically, this value is few hundred microseconds. Division by two for calculating the fair share period was just a tuning parameter. We decided not to change it even after we completed our kernel scheduler implementation.

Fairshare deadline is calculated by adding the fairshare period with the current time given by the variable `now`. Depending upon which one of the two deadlines, the earliest external deadline or the fairshare deadline is earlier, a proxy deadline event is scheduled for yielding at the appropriate time as has already been discussed in Section 2.1.3. A proxy best effort event is also scheduled so that the task may yield to the external process in case it catches up with all its best effort events (and there are no expired deadline events).

Finally, as we have discussed in Section 2.2.2, after waking up[3], if the task finds that it had no best effort events before it went to sleep (recall that the `export`() routine would keep the `have_asap` attribute cleared if the task did not have best effort events before yielding), it would borrow by setting its own virtual time as the minimum of all the ones in the slot. This is achieved by the last two lines in the above routine.

In the preceding section, we have described our user level prototype in detail. Understanding the details of the user level prototype is useful in understanding our kernel algorithm. In the next section, we describe our kernel algorithm in detail.

[3]Note that the task calls `qsf_co_op_import`() after it wakes up from a cooperative yield (sleeping on condition variable).

2.3 The Kernel Algorithm

In this section, we describe our main contribution - the kernel scheduling algorithm. As has already been discussed in Chapter 1, our goal is to design a starvation free, fair scheduler that can provide predictable timeliness for real time multimedia workloads without considerably sacrificing throughput. Further, we wanted a reservation free, work conservating mechanism to achieve our target. In addition, since uninformed context switches bring unpredictable behavior and result in poor adaptation for adaptive multimedia tasks, we wanted to eliminate pre-emptive scheduling for real time tasks through a cooperating scheduling scheme. Based on the objectives mentioned above, we made the following design decisions for our scheduler:

1. A fair share scheduler should be used to allocate the CPU resource uniformly to all tasks. This will ensure fairness in the system and avoid starvation.

2. There should be an interface through which realtime applications can communicate their timing requirements to the kernel so that the kernel can use this information while making scheduling decisions.

3. The kernel in turn should inform realtime tasks of the requirements of other realtime tasks in the system so that they may cooperate with each other and (optionally) coordinate their adaptation. Cooperation also helps to eliminate involuntary context switches.

4. Real time tasks should receive a preferential treatment over best effort tasks during scheduling.

5. Even though real time tasks get preferential treatment, they should still be bounded by the fair share timeslice as decided by the fairshare scheduler.

6. Any realtime task that violates its timeslice quota or fails to cooperate properly should be *policed*, i.e., demoted to a best-effort task until

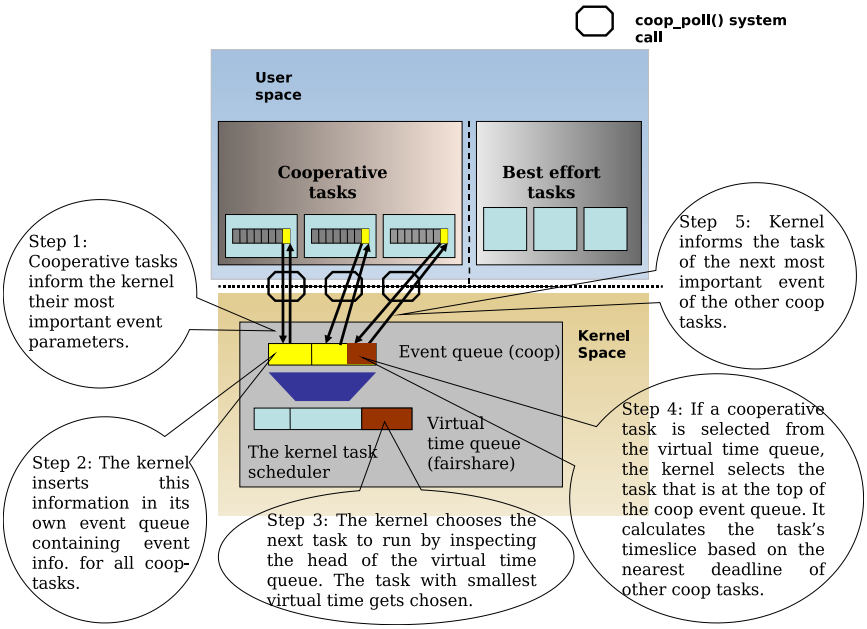


Figure 2.9: The overall scheduling approach

it again re-earns its cooperative realtime status by yielding at the appropriate time.

7. At the implementation level, our scheduler should not depend on coarse notions of time (e.g., jiffies). The scheduling mechanism should be compatible with the idea of the tickless kernel. Further, very few scheduler tuning parameters should be exposed through the `/proc` filesystem.

Figure 2.9 shows our overall scheduling mechanism that aims to satisfy all the objectives mentioned above. Each of the user level cooperative applications uses a best effort event queue and a deadline event queue to sort their events according to priority order. They inform the kernel of their most important event parameters through the IN parameter of a new primitive, `coop_poll`. The kernel then inserts this information into its own per CPU event queues. This is discussed in detail in later sections. These queues

contain the most important event information for all the cooperative tasks. In our algorithm, we use a virtual time based fairshare scheduler to choose the next task to execute. We use a general policy of choosing the task with smallest virtual time for the next execution. Our runqueue consists of a priority queue with runnable tasks sorted by their virtual time. We treat all cooperative realtime tasks as belonging to a single domain with a common virtual time. The task or domain with the smallest virtual time is at the head of the queue. Therefore, the kernel simply chooses to execute the task at the head of this queue. If this task happens to be a cooperation domain[4], the kernel then inspects the per-CPU event queues to select the most important realtime task based upon the event information of all realtime tasks. In any case, a task's timeslice is calculated on the basis of overall fairshare period and the nearest deadline of all the realtime tasks. If a realtime task is selected for execution, the deadline information of other realtime tasks is reported as an OUT parameter of `coop_poll`() so that the task can yield cooperatively to this other realtime task. Best effort tasks do not interact with the kernel through the `coop_poll` interface. They do not receive any preferential treatment from the scheduler either.

In this thesis, we do not consider SMP load balancing algorithms and task migration issues. It is left as a future work.

In the subsequent sections, we explain our scheduling mechanism in greater detail. We also discuss some of the important issues associated with the implementation of our algorithm in the Linux kernel. Most of our kernel implementation is contained within the two files `kernel/sched_fairshare.c` and `kernel/coop_poll.c`. They together consists of about 2500 lines of commented kernel code. This also includes a small portion of code for tracing and debugging. Including the header files and the heap implementation, our entire codebase is roughly 3500 lines of commented kernel code.

[4]There can be more than one cooperation domain. See Section 2.3.1, the next section, for details.

2.3.1 Scheduling Domains and Data Structures

In order to facilitate co-existence of best effort and real time applications, we introduce scheduling domains (or groups) in our algorithm. A set of tasks treated by the scheduler under the same set of policies constitutes a scheduling domain. In our present algorithm, we have one separate domain for the best effort tasks (the best-effort domain) and another domain for the cooperative real time tasks (the cooperative domain). In Section 2.5, we discuss the introduction of multiple cooperative domains in our algorithm in order to accommodate non-adaptive applications. All the tasks belonging to a cooperative domain form a cooperation group. The tasks in this group cooperate with each other and use their CPU allocation in a way so that they can maintain uniform fidelity at the application level (discussed in detail later). Each domain has its own scheduling parameters, including the `virtual_time` parameter. Our fairshare scheduler treats the cooperative domain like a single task having a common virtual time, with timeslice proportional to the number of tasks in that domain (more details on this later).

The structure of a scheduling domain along with the new attributes introduced into a task's `task_struct` structure is shown in Figure 2.10. In the structure, `dom_ptr` is a reference to the domain structure of which the task is a member. `timeslice_start` and `timeslice_end` are the starting and ending times of a process' timeslice. The `sched_deadline` value is set by our scheduler to determine when the timeslice for the current task ends. This is discussed in detail in Section 2.3.3 when we talk about our scheduling algorithm. As a memory optimization step, these last three parameters could also be made members of the per-CPU runqueue as there is always ever only a single task running in a given CPU. `is_well_behaved` is a boolean flag variable that denotes whether a cooperative realtime task is well behaved (yielding at required times and respecting fairshare allocations). Later, we will show that we use this flag for *policing* purposes.

In our algorithm, we use efficient priority-queues as the basic building block. We use them for two main purposes:

1. To sort all the processes (or cooperative domains as a whole) in the system by their virtual time. The task having minimum virtual time is the task that is next selected for execution. This *minimum virtual time first* policy ensures fair CPU allocation.

2. To sort best effort and deadline events from all tasks in a cooperation group so that the kernel can select the most important cooperative process for execution.

Choosing the next process for execution takes $O(1)$ time. All insertions and removal of tasks from these queues can be done in $O(lg(n))$ time.

In our kernel implementation, we use binary heaps to represent these priority queues. This heap implementation was already available at the user level as a part of the Qstream codebase. We ported this heap codebase to the kernel with slight modifications (for performance reasons). The resulting codebase is available in `/lib/heap.c` file. There are two per-CPU heaps, one for the best effort events and one for the deadline events for each of the cooperative domains. There is one global per-CPU virtual time heap for the fairshare scheduler. This serves as the runqueue for our scheduler. The kernel itself has its own per-CPU runqueue data structure and in our implementation, we make our heaps a part of that runqueue. Thus, we do not need to implement any locking mechanism of our own to protect our data structures.

In the algorithms written in the subsequent sections, we represent the global virtual time priority queue by `Wfq` and the per coop-domain ones by `deadline_events` and `best_effort_events`.

In the next section, we describe in detail an important part of our scheduling algorithm - the `coop_poll`() system call interface. Thereafter, we go on to describe our main kernel algorithm.

2.3.2 The `coop_poll` System Call Interface

In this section, we describe the algorithm used by our new system call interface `coop_poll`() (representing *cooperative polling*) in detail.

```
struct  task_struct {
     // ...
     sched_dom_ptr dom_ptr;
     TimeVal       timeslice_start;
     TimeVal       timeslice_end;
     boolean       is_well_behaved;
     TimeVal       sched_deadline;
     //  ...
}

struct sched_dom  {
      enum    {COOP_DOM, BEST_EFFORT_DOM} sched_dom;
      TimeVal virtual_time;
      int            num_tasks;
      // other domain specific parameters ...
}
```

Figure 2.10: Modified kernel task_struct structure

The `coop_poll` system call takes one IN parameter and one OUT parameter as has already been discussed in Section 2.1.3. The structure of the IN and OUT parameters are shown in Figure 2.11. Both of these structures are identical. Each of them encapsulates the information for both the deadline and the best effort events. This includes,

1. The deadline time of a deadline event (the `deadline_timeout` parameter).
2. The best effort event priority (the `best_effort_priority` parameter).
3. The best effort event deadline (the `best_effort_timeout` parameter).
4. A flag value that shows whether there are any best effort events at all (the `have_asap` parameter).

The figure also shows the details of the structure of the cooperative domain. As discussed previously, it has two priority queues, one for the

```
struct inParam {
     TimeVal deadline_timeout;
     TimeVal best_effort_timeout;
     TimeVal best_effort_priority;
     boolean        have_asap;
}

struct outParam {
     TimeVal deadline_timeout;
     TimeVal best_effort_timeout;
     TimeVal best_effort_priority;
     boolean        have_asap;
}

struct sched_dom CoopDomain {
     sched_dom = COOP_DOM;
     struct dom_param param;
     struct PQueue deadline_events;alg:sched
     struct PQueue best_effort_events;
}
```

Figure 2.11: The data structures used by coop_poll .

deadline events (sorted by their deadlines) and one for the best effort events (sorted by their dual keys, priority and deadline).

We now discuss the `coop_poll`() system call algorithm shown in Figure 2.12. In the algorithm, the `currentTask` variable refers to the task that is in execution (the one that called `coop_poll`). The variable `CoopDomainPtr` is a reference to the structure of the cooperative domain shown in Figure 2.11. `EventPtr` is a reference to the event structure shown in Figure 2.2.

Our initial design of kernel `coop_poll`() was largely derived from the `qsf_co_op_yield`() routine described in Section 2.2.1. For the actual yielding, we used a per-task completion variable and the corresponding wait queue therein. A cooperative task would wake up another task by calling `complete`() on the target task's private completion variable. Since our user level prototype gave us the kind of performance we looked for, we thought, a direct translation of the algorithm in the kernel would be the best thing to do. However, when we started integrating our fairshare scheduler and policing mechanism with `coop_poll`(), we realized that this approach was grossly incorrect. This is because we used a sleeping semantics in `coop_poll`. However, `coop_poll` is a mechanism to yield to another cooperative task for a small duration of time (specially when the yielding task had other best effort work to do). Thus, `coop_poll` behaved more like the `sched_yield`() system call. Removing the task from the runqueue during yielding would have been an incorrect approach. When we realized it, we had to scrap our old implementation and reimplement the system call. This time we used `schedule()` from `coop_poll` to choose the next task to execute keeping the yielding task still in the runqueue. In the subsequent sections, we only describe this final and modified algorithm in greater detail.

The `coop_poll` system call can be split into a top half and a bottom half. The top half is executed before the process actually yields to the kernel. All the code from the beginning of `coop_poll`() until before the call to `schedule`()(the core kernel scheduler) consists of the top half of `coop_poll`(). When the process is scheduled again for execution (after its cooperative yield), it starts right after the `schedule`() call. All the code from this point until the end of `coop_poll`() consists of the bottom-half.

We will describe the two halves in detail in the subsequent sections.

The Top Half of `coop_poll()`

At the very beginning, the routine removes nodes of the current task from the cooperative domain heaps if they are there. This is important because we design our algorithm in a way such that the event node of the currently executing cooperative task is left at the top of the heap. This design policy follows from the observation that when a task is executing and has not exhausted its timeslice, there is no need to change the state of the scheduler data structures. When the task cooperatively yields by calling `coop_poll`() and requests a new task to be scheduled, it is therefore important to update the nodes in the heap and remove stale nodes so that a new task can be selected based on the updated information.

`coop_poll`() not only signifies voluntary yielding, it is also a request to seek membership with the cooperation domain. The routine then checks to see if the current task is a member of the cooperative domain. If not, it marks the current task as belonging to this domain and sets its `well_behaved` flag to TRUE. Recall from our discussion in Section 2.1.1 on reactive programming that any cooperative process that obeys the reactive model can not block, preempt or sleep anywhere other than the routine it uses to yield voluntarily. Thus, as long as the process is executing within `coop_poll`(), it is considered to be *well behaved* (because it is the only place where it can be preempted). If the process yields or blocks or is preempted because it has exceeded its scheduler determined allocation at any place other than this routine, the process is considered to be misbehaving. It is then subjected to *policing* which will be described in Section 2.3.3. Therefore, when the task enters `coop_poll`, its `is_well_behaved` flag is set to TRUE. When it leaves the routine, this flag is set to FALSE.

The routine then checks the input arguments to see if the task has outstanding best effort events or deadline events or both. If the task does not have outstanding best effort events and the deadline time of its earliest deadline event is in the future, it calls `putTaskToCoopSleep`. This routine

is responsible for putting a task into *cooperative sleep*. This sleep is different from involuntary sleeps in that the scheduler is aware of the task's needs and therefore can make attempts to schedule the process as close as possible to its future deadline. Note that during the time the task is cooperatively sleeping, it remains removed the runqueue. This is a reasonable action since the task has nothing else to do until the time corresponding to its next deadline event. In order to achieve finer grained context switching even when the system was not overloaded, `putTaskToCoopSleep` first inserts the deadline event parameters of this task in a separate per-CPU priority queue named `coop_sleep_queue`. The significance of this action will be further explained in a later section where we describe our algorithm to calculate the task timeslice. It then puts the task to cooperative sleep until its deadline time. In our kernel implementation, the actions of `putTaskToCoopSleep` is performed by the function `cooperative_sleeper`(). The code for the later is shown in Appendix B.2.

If the task has both best effort and deadline events, the routine inserts these events in the queues and calls `schedule`() to select the next task for execution based upon the updated information in the heaps.

The Bottom Half of `coop_poll`()

After the task resumes its execution, the routine extracts the next most important best effort event in the system (if any), other than current task's own event (the current task's own event nodes are at the top of the queue). It then sets the best effort attributes of the *out* parameter accordingly. The deadline attribute of the *out* parameter is set from the current task's `sched_deadline` parameter. This parameter is set by our scheduling algorithm when it selects a new cooperative real time task for execution. We discuss this in the next section. Finally, the task's `is_well_behaved` flag is set to FALSE (as has been discussed earlier) so that when the task sleeps uncooperatively or gets preempted, policing can be imposed.

```
coop_poll(struct inParam in, struct outParam out) {
     if(taskInCoopQueues(currentTask)) {
           q_remove(CoopDomainPtr.best_effort_events,
                    currentTask);
           q_remove(CoopDomainPtr.deadline_events,
                    currentTask);
     }
     if(isTaskCoopRealtime(currentTask) != TRUE) {
           setTaskDomain(currentTask,COOP_DOM);
     }
     currentTask.is_well_behaved = TRUE;
     if(in.have_asap == TRUE) {
           q_insert(CoopDomainPtr.best_effort_events,
                    in.best_effort_timeout,
                    in.best_effort_priority,
                    currentTask);
           q_insert(CoopDomainPtr.deadline_events,
                    in.deadline_timeout,
                    currentTask);
      } else if(in.deadline_timeout > now) {
           q_insert(CoopDomainPtr.deadline_events,
                    in.deadline_timeout,
                    currentTask);
           putTaskToCoopSleep(currentTask,
                              in.deadline_timeout - now);
           goto wakeup;
     } else {
            in.deadline_timeout = now;
            q_insert(CoopDomainPtr.deadline_events,
                    in.deadline_timeout,
                    currentTask);
     }
     schedule();
wakeup:
    flag = FALSE;
    if(taskInBestEffortQueue(currentTask) == TRUE) {
           q_remove(CoopDomainPtr.best_effort_events,
```

```
                    currentTask);
            flag = TRUE;
    }
    EventPtr topBE = NULL;
    if(q_not_empty(CoopDomainPtr.best_effort_events)) {
            topBE =
            q_head(CoopDomainPtr.best_effort_events);
    }
    if (flag == TRUE) {
            q_insert(CoopDomainPtr.best_effort_events,
                    currentTask);
    }
    if (topBE != NULL) {
          out.best_effort_timeout  = topBE.deadline;
          out.best_effort_priority = topBE.priority;
    }
    out.deadline_timeout = currentTask.sched_deadline;
    currentTask.is_well_behaved = FALSE;
    return;
}
```

Figure 2.12: The coop_poll algorithm.

Our kernel implementation of `coop_poll` follows directly from the algorithm described above. The complete C source code for the actual implementation is given in Appendix B.1.

We have described in detail our `coop_poll` system call interface in this section. In the next section, we describe our main kernel scheduling algorithm. We show how the `coop_poll` system call becomes an integral part of our scheduling algorithm.

2.3.3 The Kernel Scheduling Algorithm

In this section, we discuss our main kernel scheduling algorithm. As we have discussed previously, our scheduling algorithm employs a combination of fairshare scheduling and cooperative polling. Fairshare scheduling ensures

overall fairness and a starvation free system. However, it alone can not ensure uniform coordinated adaptation for multiple adaptive time sensitive tasks. This is because each these tasks has variable CPU requirements over a period of time and across all the runnable tasks at a particular time. Hence, a simple fairshare model can not match their requirements. Further, as we will show in Chapter 3, even though we can increase the overall timeliness of time sensitive applications to a certain degree by increasing the fairshare scheduling granularity, we soon hit a bottomline threshold. The timeliness can not be improved beyond this point.

Therefore, our scheduling algorithm must satisfy two conflicting requirements. First, in order to ensure predictable timeliness, we can't throw away fairness and time isolation between tasks. Secondly, since fairshare scheduling alone can not match the requirements for adaptive time sensitive tasks, we need to allow these tasks to consume CPU resources according to their requirements so that they have uniform fidelity at the user level.

In order to meet these two conflicting requirements, we decided to take the following policies:

1. The timeslice for all tasks in a system with no time sensitive tasks is calculated on the basis of Equation 2.2, where N is the number of runnable tasks (including the number of realtime tasks) and `sched_min_timeslice` has the same meaning and value as `min_timeslice` in Equation 2.1. `sched_granularity` is either a scheduling constant or can be adjusted by the user. The timeslice with mixed best effort and time sensitive tasks is based on a combination of fairshare timeslice as well as the earliest next deadline of all the time sensitive tasks. This combined approach should ensure preferential treatment of the time sensitive tasks, based on their deadlines over best effort ones.

$$timeslice_{fairshare} = max(sched\_granularity/N, sched\_min\_timeslice) \tag{2.2}$$

2. All the real time tasks taken together get a combined CPU allocation

as a group (the cooperation group/domain). This allocation is directly proportional to the number of real time tasks and the fair share allocation per task determined by Equation 2.2. However, there is no enforcement of fairsharing between tasks within the same cooperation domain. The real time tasks are allowed to use their allocation based on their requirements. This leeway gives them an opportunity to coordinate their adaptation, within the limits imposed by the fairshare scheduler.

3. We provide strict time isolation in that a task can't be pre-empted before the timeslice expires, unless it sleeps.

Based on the policies, we describe our main scheduling algorithm next.

Main Scheduler Routine

Figure 2.13 shows our main scheduling routine `schedule()`. It takes two external global parameters. They are `sched_granularity` and `sched_min_timeslice`. The meaning of these two has already been explained. The very first part of the routine enforces our third policy as discussed above - we do not context switch unless the timeslice for the currently running task has expired (or it went to sleep). In our algorithm, timeslice boundaries are enforced by one shot timers. Hence, if this timer is still active for the current task, we allow the task to keep running.

If the timeslice did expire (or the previous task went to sleep and its timer was cancelled), we charge the current task by an amount of time proportional to its running time. This is done by the `safely_charge_running_times()` routine discussed in the next section.

`choose_next_task`() is the core of our scheduler. It chooses the next task to execute based on certain criteria. We discuss this routine in detail separately in a later section. Once a new task is selected, its timeslice start time is assigned to be the current time. In our kernel code, we use the native kernel function `ktime_get_ts`() in order to get the kernel monotonic time value in `ktime` within a `timespec` structure. This provides the best available timing resolution (64 bit). Thereafter, its timeslice is calculated and a timer

is scheduled so as to enforce this timeslice. The later is performed by the `schedule_timer`() routine. In our kernel implementation, we use one shot high resolution timers for all our allocation enforcement. None of our code paths depend on the jiffy level timing granularity. However, we do let users choose between high resolution and jiffies through a kernel configuration option. This is provided so as to let users test our code even when high resolution timers are not available for older kernels. Our use of one shot timers and independence from jiffies make our code compatible with tickless kernels.

The new task's `sched_deadline` attribute is set based on the timeslice of this task. As discussed in Section 2.3.2, the cooperative tasks use this information to inform the user level of the time by which it must again yield to the kernel. This is the basic essence of cooperation. Finally, the actual context switching is done by call to routine `context_switch`() which is architecture specific.

```
Global TimeVal sched_granularity;
Global TimeVal sched_min_timeslice;
schedule() {
     prevTask = currentTask;
     if (fsTimerActive == FALSE) {
           safely_charge_running_times(prevTask);
           nextTask = choose_next_task();
           nextTask.timeslice_start = now;
           TimeVal timeslice = calculate_timeslice();
           schedule_timer(timeslice);
           nextTask.sched_deadline = now + timeslice;
     } else {
           nextTask = prevTask;
     }
     if (nextTask != prevTask)
           context_switch(prevTask,nextTask);
}
```

Figure 2.13: The main kernel scheduler routine

Charging Running Times

Figure 2.14 shows the routines that charge the running time to the current task. Notice that for best effort tasks, we charge them their exact running time. However, for realtime tasks, we scale their running times by the number of realtime tasks in their domain. This is because realtime tasks receive an allocation as a group in totality and are treated as a single task by the fairshare scheduler. The scaling ensures that the group as a whole gets the fair amount of CPU allocation depending upon the number of tasks in that group. From the point of view of the fairshare scheduler, one may also consider the coop group as a single task with a weight equal to the number of coop-realtime tasks the group has.

```
update_running_times(prevTask) {
     prevTask.timeslice_end = now;
     TimeVal running_time = prevTask.timeslice_end -
                            prevTask.timeslice_start;

     if (prevTask.dom_ptr.sched_dom == COOP_DOM) {
         running_time = running_time/CoopDomain.numtasks;
     }
     prevTask.dom_ptr.virtual_time =
         prevTask.dom_ptr.virtual_time + running_time;
}
safely_charge_running_times(prevTask) {
     remove_node_from_fs_q(prevTask);
     update_running_times(prevTask);
     reinsert_node_to_fs_q(prevTask);
}
```

Figure 2.14: Routines that charge running times to the current task

The Scheduling Logic

Figure 2.15 shows the main scheduling logic used by our combined fairshare-cooperative scheduler. We employ a two-level hierarchical approach in our scheduler. At the top level, the fairshare scheduler selects the task with the minimum virtual time. If this task happens to be a real time task, our next level of cooperative logic selects the most appropriate cooperative realtime task. The routine that does this selection is `choose_next_coop_task`().

`choose_next_coop_task`() first checks to see if there are any expired deadline events in the deadline event heap of the coop-domain. If there is, it then selects the corresponding task. If not, then it inspects the best-effort queue and selects the task corresponding to the event at the head of this queue.

The next section describes how we calculate the timeslice value corresponding to this selected task.

```
choose_next_coop_task() {
     if (head_expired(CoopDomain.deadline_events)){

        nextDeadEv = q_head(CoopDomain.deadline_events);
        return task(nextDeadEv);

     }else if(q_not_empty(CoopDomain.best_effort_events)){

        nextBeEvent =
               q_head(CoopDomain.best_effort_events);
        return task(nextBeEvent);

     } else {
        return ERR;
     }
}

choose_next_task() {
     nextTask = q_head(Wfq);
     if (nextTask.dom_ptr.sched_dom == COOP_DOM) {
```

```
            nextTask = choose_next_coop_task();
        }
        return nextTask;
}
```

Figure 2.15: The main scheduling logic

```
anirbans@ani:/proc$ cat bvtstat
timestamp 229211679
global bvt period = 20000000 nsec
current bvt period (cpu 0) = 100000 nsec
minimum bvt period = 100 usec

anirbans@ani:/proc$  cat coopstat
timestamp 229404690
cpu#    coop_poll#      yields#
cpu0:   2435753         2435753
```

Figure 2.16: The scheduling statistics exported through `/proc` filesystem.

Calculating Timeslice

Figure 2.17 shows the main routine that calculates the timeslice for the next chosen task. We start by calculating the fairshare period using Equation 2.2. This calculation is done by the routine `find_fair_share_period`(). In our kernel implementation, we allow the users to tune the global scheduling period in Equation 2.2 by exporting a scheduling parameter in the `/proc` filesystem. By writing an appropriate value of the period (in micro seconds) into `/proc/sys/kernel/bvt_sched_period_us`, the global period can be adjusted. This is the only adjustable scheduling parameter exposed through `/proc` even though we do export some other scheduling statistics as well (see Figure 2.16).

```
calculate_timeslice(nextTask) {

      TimeVal fsPeriod = find_fairshare_period();

      if (nextTask.dom_ptr.sched_dom == COOP_DOM) {
            fsPeriod =  CoopDomain.numtasks * fsPeriod;
      }

      TimeVal timeslice        = fsPeriod;
      TimeVal earliestCoopDead = find_earliest_deadline();
      TimeVal coopPeriod       = earliestCoopDead - now;
      if (coopPeriod < 0) coopPeriod = 0;
      nextDeadTask = find_earliest_deadline_task();
      if (nextTask.dom_ptr.virtual_time + coopPeriod <
          nextDeadTask.dom_ptr.virtual_time) {
             timeDelta = nextDeadTask.dom_ptr.virtual_time
                         - (nextTask.dom_ptr.virtual_time
                          + coopPeriod);
             coopPeriod = coopPeriod + timeDelta;
      }
      if (timeslice > coopPeriod) {
             timeslice = coopPeriod;
      }
      if (timeslice < sched_min_timeslice) {
             timeslice = sched_min_timeslice;
      }
      return timeslice;
}
```

Figure 2.17: The routine that calculates a task's timeslice

Since all cooperative realtime tasks receive allocation as a group, the period we calculate previously is then multiplied by the number of tasks in the cooperative realtime domain if the next chosen task is one of them. This value is the overall fairshare period (`fsPeriod`) as determined by the

fairshare scheduler.

We have discussed previously that one of our design decisions was to give preferential treatment to realtime tasks by taking into consideration their deadline information. The next few steps in the routine try to achieve this objective. We first find the time period between now and the next global deadline which we name `coopPeriod` in the routine. The next global deadline is found by the routine `find_earliest_deadline()`. This routine finds the global earliest deadline of all realtime tasks, including those whose event information is in the `coop_sleep_queue` (see Section 2.3.2). Next, it finds the pointer to the non-sleeping realtime task having the earliest deadline. Then it checks to see whether by running the next task by `coopPeriod` results in the task having a virtual time greater than the previously found realtime task. If not, it adjusts this period accordingly. The period thus calculated becomes the period of the next running task provided it is not greater than the overall fairshare period calculated earlier or less than the minimum timeslice.

This approach ensures that when the deadline of the next realtime task expires, the current task will be preempted unless the virtual time of the currently running task is still smaller than that of the realtime task. In that case, the current task keeps running until its virtual time becomes greater. Further, there might be other best effort tasks that have smaller virtual time than the realtime task whose deadline just expired. After preempting the current task, the scheduler will then choose to run one of those best effort tasks, the one whose virtual time is smallest. However, in that case, since the deadline has already expired, `coopPeriod` will be 0 and all these tasks will run for an amount of time that is just enough to make their virtual time greater than this realtime task. Ultimately, all of them will catch up with the virtual time of the realtime task whose deadline expired.

Note that this policy increases the event dispatch latency for the time sensitive task. However, this also ensures that the fairness across all the tasks is not violated. The *minimum virtual time first* ensures that all tasks get a fair opportunity to execute on the CPU.

Routine in algorithm	Implemented kernel function
`safely_charge_running_times()`	`charge_running_times`()
`update_running_times()`	`update_virtual_times`()
`choose_next_task`()	`choose_next_bvt`()
`choose_next_coop_task`()	`choose_next_coop`()
`calculate_timeslice`()	`calculate_bvt_period`()
`schedule_timer`()	`schedule_dynamic_bvt_timer`()

Table 2.1: One to one relationship between algorithm routine and our implemented kernel function

Table 2.1 shows the relationship between the routines previously described in our algorithm and our implemented kernel functions. We hook our scheduling algorithm with the main kernel scheduler function `schedule()` using two of our own functions - `update_bvt_prev()` and `prepare_bvt_context_switch()`. `update_bvt_prev()` charges the previous task its own running time. Therefore it calls the routine `update_running_times`() as described in Section 2.3.3.
`prepare_bvt_context_switch()` then performs the rest of the scheduler functions - choosing the next task to execute, calculating timeslice and scheduling the timer for the newly chosen task.

Figure 2.18 shows the algorithm used by the native Linux 2.6.20 scheduler with the two functions mentioned above. Instead of removing the entire logic in `schedule()`, we simply override its native decision by the decision of our algorithm.

```
schedule()
{
        if (in_atomic_context())
                panic();
        rQ = currentRunQueue();
        disable_preemption();
        irq_disable();
        spin_lock(rq);
        prevTask = currentTask;
        update_running_time(prevTask);
        if (non_runnable(prevTask) &&
                preemption_enabled()) {
                if (signal_pending(prevTask) &&
                        sleep_type(prevTask) ==
                                SLEEP_INTERRUPTED) {
                        set_task_state(prevTask,RUNNABLE);
                }
                else
                         deactivate_task(prevTask);
        }

        update_bvt_prev(prevTask);

        if (rq.active_arr.nr_active == 0)
                switch_arr(rq.active_arr, rq.expired_arr);
        nextTask = find_next_task(rq.active_arr);

        prepare_bvt_context_switch(prevTask, nextTask);

        update_sleeping_times(prevTask);
        prio = recalc_task_prio(nextTask);
        if (prio != nextTask.prio) {
                requeue_task(nextTask, prio);
        }
        if (prevTask!=nextTask) {
                context_switch(prevTask, nextTask);
        }
```

```
        spin_unlock(rq);
        irq_enable();
        enable_preemption();
}
```

Figure 2.18: Modified Linux 2.6.20 scheduler with our kernel functions

Borrowing

In our algorithm, we implement virtual time borrowing. Any task that wakes up from sleep gets a fresh virtual time that is equal to the minimum of the virtual times of all runnable tasks. Since we treat cooperative tasks as belonging to one single domain, all the tasks in this domain have a common virtual time. The cooperative domain borrows as a whole when the first task in that domain wakes up. Any subsequent wakeups of other tasks will cause them to inherit their common domain specific virtual time and hence they need not borrow. Borrowing makes two tasks (one that woke up and the other that was previously at the top of the queue) have same virtual time. To break the tie, we employ a FIFO policy. The task that was previously at the top of the queue still remains at the top by virtue of its earlier entry into the queue. This FIFO policy helps to prevent starvation.

Borrowing ensures that:

1. No task can accumulate virtual time. When a task sleeps, it gives up the virtual time it has accumulated during the course of its execution. When it wakes up, it gets a fresh virtual time and begins anew. This prevents IO intensive tasks from getting an occasional big shot at CPU by virtue of its accumulated virtual time.

2. Interactive tasks that sleep often, wake up and get assigned a minimum virtual time across all the currently running tasks in the runqueue. Thus there is a bounded latency between the time the task wakes up and when it gets to execute. This latency is solely dependent on the number of tasks that were there earlier in the queue with exactly the

same virtual time. In practice, the number of such tasks will be very small and the interactive tasks will have a high responsiveness.

In the kernel implementation, borrowing of virtual time is implemented by calling our function `bvt_borrow`() from the Linux kernel function `activate_task`(). `activate_task`() is called whenever a task is reinserted into the runqueue after waking up. The implementation follows directly from the algorithm. The code corresponding to borrowing is shown in appendix B.3. One thing significant to note from the code is that when a cooperative task wakes up, we reinsert its deadline event information back into the kernel event queues and remove its nodes from the `coop_sleep_queue` (see Section 2.3.2).

Policing

Policing is imposed on two distinct cases:

1. When a cooperative task fails to honor the deadlines of other tasks and overshoots its timeslice. In this case, the timer fires and enforces policing. Note that if a cooperative task yields by calling `coop_poll`() before its timeslice expires, we cancel the running timer from within `coop_poll`().

2. If a cooperative realtime task goes to sleep other than the cooperative sleep described in Section 2.3.2. In this case, we enforce policing by checking whether the `is_well_behaved` flag for that task is true. Recall that tasks go to cooperative sleep from within `coop_poll`() where this flag is true.

In the kernel implementation, we implement the first through the timer handler. In the timer handler, we check to see if the current running task is cooperative realtime. If it is, we impose policing by removing the task's event information from the queues and calling `do_policing`() (described later). For the second, we implement policing in the Linux kernel function `deactivate_task`() (called when a task becomes non-runnable). In

`deactivate_task`(), we check to see if a task is a cooperative realtime task and it is not well behaved. If both the conditions are TRUE, we impose policing. Note that we do not impose policing when a cooperative realtime task sleeps cooperatively from within `coop_poll`(). In the later case, the task is marked as well behaved (as it is executing within `coop_poll`) and policing is skipped. `deactivate_task`() however does one more thing. It cancels the running timer for the current task, regardless of whether its realtime or not. The code for `deactivate_task`() with our modifications is shown in Appendix B.5

The actions performed during policing are as follows:

1. It demotes a realtime task to a best effort task, changing membership of domains.

2. It charges the running time of the task to the task itself and not to the cooperative domain. This ensures that the other members of the cooperative domain remain unaffected from the misbehaving task and they still get preferential treatment.

Therefore, at the end of policing operation, the cooperative realtime task loses its realtime status and is also pushed away from the head of the priority queue (because the running time was charged to the task).

In our kernel implementation, the first of the above steps is performed by the function `do_policing`. It also assigns the virtual time of the coop-domain to the task so that the task has correct virtual time and the end of step 2 above. The actual charging of the virtual time is done by the routine `safely_charge_running_times()` as has been discussed in Section 2.3.3. The code for the `do_policing`() is shown in Appendix B.4.

2.4 Scheduling Best Effort Tasks

From Figure 2.18 it clear that in our current implementation, we do not throw away the entire vanilla 2.6.20 kernel scheduler code. Instead we override its scheduling decisions by ours. Further, we implement a new system

call called `set_bvt`() through which processes can request to be scheduled under our fairshare scheduling algorithm. In `schedule()`, we enforce our scheduling decision only for these tasks. In order to test the performance of our fairshare algorithm as a general purpose scheduler for all best effort tasks, it is important to bring all Linux tasks under our fairshare scheduling regime. We have cleaned up a large portion of our code towards this end. The ultimate goal of running all tasks under our fairshare scheduling regime was finally completed by another student, Mayukh Saubhashik. The complete performace analysis of our fairshare scheduler with all best effort tasks remains a future work (see Chapter 5, Section 5.1.1). A full rewrite of the current scheduler and replacing its algorithm with ours is unnecessary under the new 2.6.23 pluggable scheduler (see Chapter 5, Section 5.1.7).

So far, we have given a detailed description of our scheduling algorithm in the kernel. In the next section, we describe how, with a simple extension, non-adaptive applications can also be accommodated within our scheduling framework.

2.5 Accommodating Non-Adaptive Time Sensitive Applications

Our original cooperative polling scheme takes into account two kinds of event information - the deadline event information and the best effort event information. The best effort events are used by the adaptive applications to share the CPU according to their own requirements. Hence they are able to achieve coordinated adaptation at the user level. However, the notion of adaptation is application centric and is specified via best effort event priorities. The algorithm we have described so far does not accommodate applications that are non-adaptive. Further, even with adaptive applications using some form of best effort events, there is no guarantee that they use the same notions of best effort event priority. With a mix of adaptive and non-adaptive applications, the situation becomes even more complex.

We wanted to extend our algorithm in order to decouple the issues of

achieving predictable timelines from coordinated adaptation. As a first step, we made the following extensions to our algorithm:

1. We modified our domain data structure in Figure 2.10 to include multiple cooperative domains. Each of the cooperative domains have deadline and best effort event priority queues for all the tasks belonging to that domain.

2. We modified the `coop_poll`() interface to add one new IN parameter, the `dom_id`. This is a simple integer variable that denotes which cooperative domain the task wants to be a member of.

3. The `find_earliest_deadline`() routine in Figure 2.17 is modified to find the earliest deadline across all the realtime tasks in all the domains. This is done by iterating through the head of the priority queues of all the deadline events of all the domains. This is an $O(N)$ algorithm where N is the number of coop domains. In our current implementation, this is a fixed number (15 domains).

4. In `coop_poll`(), when we report the most important external best effort event to the user space, we only inspect the best effort events from the coop-domain of which the current task is a member. This enables members of the same coop-domain to cooperatively use the timeslice allocated for that specific domain. Members of different coop-domain do not interact with each other. Further, there is isolation and protection between coop-domains and not within the members of the same domain as before.

Thus, our multiple cooperative domain algorithm is just a simple extension of the single domain case. When all tasks seek membership to one specific cooperative domain, it reduces to our previous algorithm having one cooperative domain. However, we have an interesting case when all realtime tasks seeks membership in different domains.

With all realtime tasks belonging to different domains, scheduling decisions and yieldings are made solely on the basis of deadline events. Since

there is only one cooperative realtime task in one coop domain, best effort events for other cooperative realtime tasks are not considered (see point 4 above). Thus, realtime tasks having only deadline events can get better timeliness even with only their deadline information. Also note that the best effort event parameter is strictly not necessary in this algorithm. However, they are used to decide whether a task can be put to cooperative sleep (since it has no outstanding work pending) or it should be left runnable. Lack of best effort events signify the former.

One issue with the above algorithm is that each of the (non-adaptive) realtime tasks has to be a member of a different coop-domain if they want to get timeliness benefits. In our present algorithm there are fixed number of cooperative domains. A task, in general, does not know which domains other realtime tasks are the members of. In essence, we need variable number of domains that can be created by the realtime tasks themselves. Alternatively, we need some kind of mechanism for the kernel to automatically decide which domain the task should be a member of. Addressing some of these issues are left as a future work.

2.6 Chapter Summary

In this chapter, we have discussed our task scheduling approach, both as a complete task scheduler in the Linux kernel as well as a prototype implementation in the user level. In this respect, we have also discussed the new data structures and other parameters related to our algorithm. We have reviewed some of the basic ideas of the event driven reactive model and have shown how this work fits into this model. Our approach, which is a combination of a fair share scheduling with cooperative polling is designed so as to satisfy the main objectives of our work. Through fairsharing we ensure that there is an overall long term fairness in the system. Fairsharing also helps us to implement policing for realtime tasks. Whereas the use of fairshare scheduling approach is not new, the combination of fairsharing with cooperation is a novel aspect of our work. Even though our algorithm was originally designed for adaptive multimedia workloads, we show that

with a simple extension, our algorithm can accommodate non-adaptive time sensitive applications as well.

In this chapter, we have also discussed some of the main implementation issues of our algorithm in the Linux kernel. Our implementation uses some of the recent infrastructural components in the kernel, such as fine grained preemption, high resolution time accounting and timers, etc. Though we have a basic kernel implementation of our algorithm, it is far from complete. We do not as yet have support for multicore and hyperthreaded processors in our implementation. We do not take any optimization steps for the kernel codepath.

In the next chapter, we discuss the evaluation our implementation. We show that our scheduling approach can provide predictable timeliness for all realtime tasks without sacrificing long term fairness in the system, even in overload situations.

Chapter 3

Performance Evaluation

In this chapter, we provide evaluation and benchmark results from a prototype of the scheduling algorithm implemented in the user level and from the actual scheduling algorithm implemented in the kernel. In this thesis, we focus on the performance of our algorithm for multimedia real time applications. Further evaluations for various kinds of best effort tasks and other workloads and comparison with the CFS scheduler [27] are in progress but are outside the scope of this thesis.

3.1 Experiments with Multiple Instances of VLC and MPlayer

To emphasize the relevance of our work and in order to show the shortcomings of the Linux 2.6.20 kernel scheduler, we first describe our preliminary experiences of running multiple independent players on the vanilla Linux 2.6.20 kernel. We use two of the most popular open source video players available today: *MPlayer*[13] and the VideoLan Client (*VLC*)[2]. This work was done as a part of our our previous work involving priority progress decoding [23] and the results were reported in the paper. Since this work was done at a very early stage, the experimental setup for this experiment is different from the rest of all other experiments. We do not expect that the differences in the setup to effect the substance of the results. These experiments were performed on a Dell Inspiron 1300 laptop PC, with a 1.8 GHz Pentium-M CPU and 512 MB memory running the Ubuntu Linux 6.06 distribution. We chose MPlayer and VLC because they are very popular video applications of very high overall quality, the result of very large teams

of talented and dedicated developers. The open source nature of these applications allowed us to add our own instrumentation code for collecting data.

We measured the performance of MPlayer and VLC by playing 3 and 6 videos (the same video) simultaneously. Each video is played in a separate player. The CPU usage remains just below saturation with 3 videos while the CPU is fully saturated with 6 videos. The video is a movie taken from a DVD and converted to MPEG-4 format. We measured the frame rate and jitter of the videos.

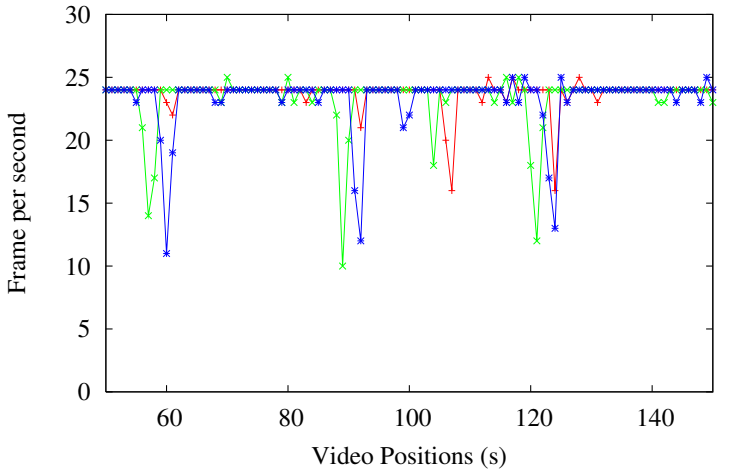
Figure 3.1 shows the frame rates of the three videos for the VLC and MPlayer video players during underload. Both players maintain close to the full frame rate of 24 fps, and subjectively, the videos play with normal smoothness and no noticeable pauses.

Figure 3.2 shows the performance of VLC and MPlayer during overload (CPU usage is pegged at 100%) when the frame-rate adaptation mechanism was active in both the players. The graphs on the left show that the frame rates of the videos varies dramatically. Both players exhibit bi-modal fairness with many of the videos experiencing zero or low frame rates and some that have almost full frame rate. Figure 3.2(b) shows that jitter reaches up to one second for VLC, which experiences several pauses. MPlayer is able keep jitter below 200 ms.

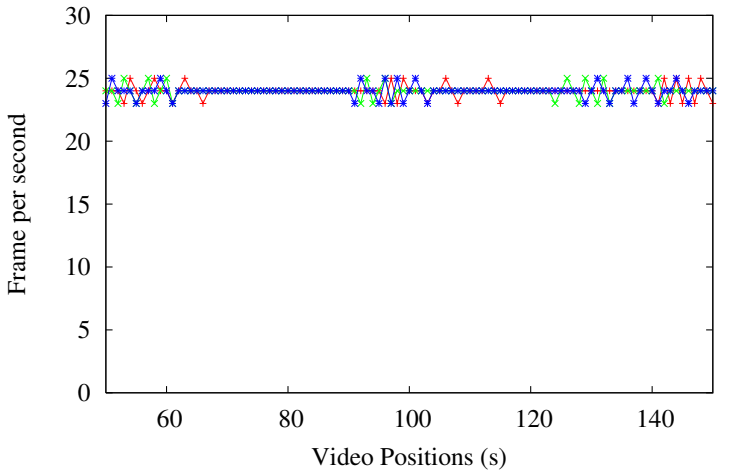
To conclude, these players maintain acceptable performance during underload but not during overload. Interestingly, the event-driven MPlayer is more consistent than the threaded VLC player even though both players adapt quality similarly during overload. While we have not analyzed this difference in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

3.2 Benchmarking Tools Used

For the rest of all our experiments, we use the adaptive media streaming framework, Qstream [21] to gather data and analyze the results. Qstream

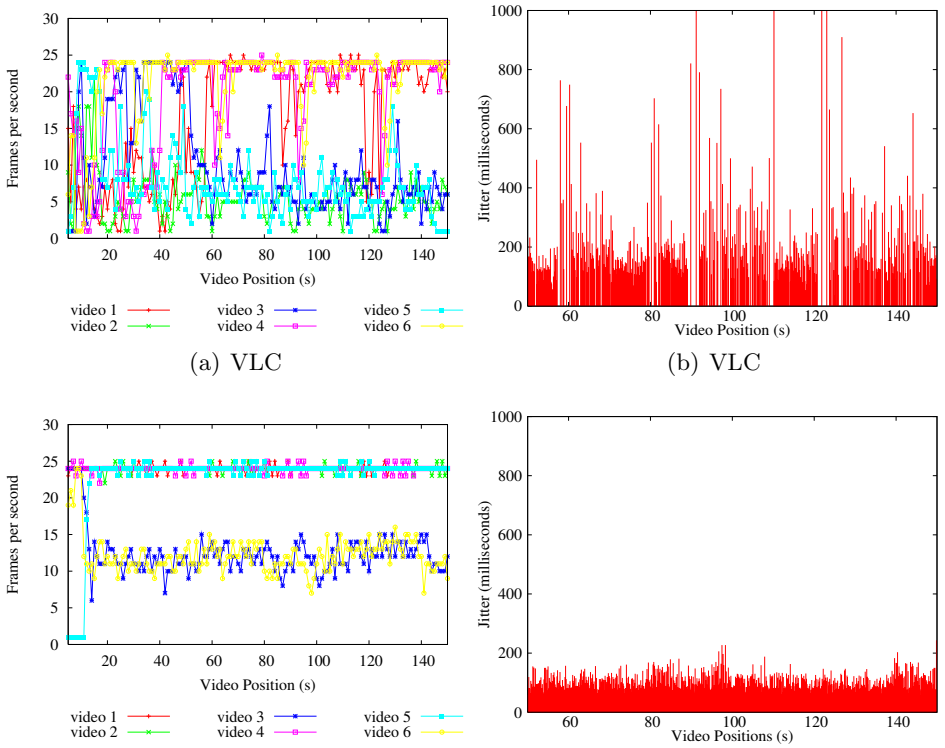


(a) VLC



(b) MPlayer

Figure 3.1: VLC and MPlayer video players in underload



(a) VLC (b) VLC

(c) MPlayer (d) MPlayer

Figure 3.2: VLC and MPlayer video players in overload.

is an adaptive media streaming application based on the event driven programming style that was the inspiration for our cooperative model. It has built-in tools and options for tracing, profiling and performance diagnosis. Gathering data using Qstream was much more automatic and seamless than if we had used any other multimedia applications.

In the next section, we describe the details of the setup we use for our experiments. We only discuss the basic configuration details. For other finer details of our experimental configuration, please refer to Appendix A.

3.3 Experiment Description

3.3.1 Basic Setup

Qstream has a client-server architecture. In all our benchmarks, we run the server side of Qstream on a separate physical machine connected to the client machine through gigabit ethernet (thus the network does not become a bottleneck in our experiments). This separation of client and server in different physical machines ensures that the server side disk IO and processing does not affect our results. We run all our video benchmarks on the client side. Figure 3.3 shows the basic client server setup. A detailed description of our experimental methodology is in Section 3.3.4. For all the benchmarks, we use scripts to automate the entire process of running the workload, data collection and plot generation. The scripts have different command line options through which we can automate running several different sets of experiments. We generate the plots from the results using Gnuplot [47]. All the scripts that are used to generate the results are released open source and can be viewed and downloaded from the Qstream public repository [22].

3.3.2 Hardware Configuration

The hardware configuration of the client is as follows: A Pentium IV, 3 Ghz machine with 1 Gigabyte RAM and a NVIDIA GeForce 6200 display card with 256 MB video memory. We disable CPU hyperthreading from

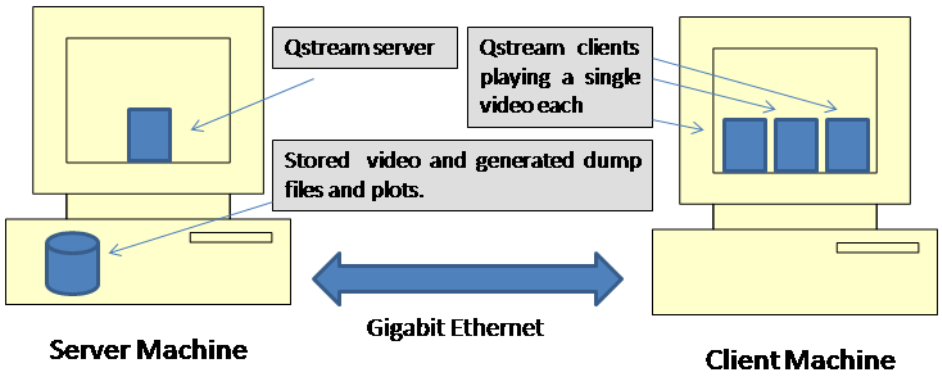


Figure 3.3: Basic experimental configuration

the BIOS for all our experiments since we do not support multiprocessor scheduling. The server side consisted of a 3 Ghz Pentium IV machine with one two way hyperthreaded CPU, 1 Gigabyte of RAM. It ran debian Linux with the 2.6.20 SMP kernel.

3.3.3 Kernel Configuration

For all our experiments, we use Linux 2.6.20 as our base kernel downloaded from `http://kernel.org`, patched with high resolution timer support. All our developments are done on the top of this kernel. In the rest of the thesis, the word *vanilla* is used to describe the stock Linux kernel as directly available from `http://kernel.org` without any modification or patches applied. The Linux 2.6.20 kernel uses the traditional $O(1)$ scheduler based on the well known multi-level feedback queue scheduling algorithm [5, 34]. Very recently, a new kernel task scheduling algorithm (named the Completely Fair Scheduler [27]) based on fairshare scheduling approach was incorporated into the mainstream Linux 2.6.23 kernel. We do not do any benchmarking over this kernel as it was released after the completion of this work.

We disable SMP and multicore options in our kernel for all the experiments. We also make sure that all the debugging and kernel performance measurement options remain disabled since they often add extra overhead.

Fine grained preemption (kernel preemption when executing in non-critical section codepath) was enabled in our kernel for all the experiments.

3.3.4 Experimental Methodology

Through our experiments, we aim to characterize the performance of our scheduling approach across a broad spectrum of system load, from an underloaded system to a completely overloaded system. Our main focus though is to evaluate the performance of our algorithm in the extreme overload condition. In our experiments, we perform a set of runs with the same configuration. We vary the load by adjusting the number of players in each run, increasing it in successive runs, from four players to twelve players. We note that with six players, in our setup, the CPU just becomes saturated. With twelve players, the CPU remains completely saturated at all times. Running a variable number of players therefore gives us a broad spectrum of the experimental condition, from fully underloaded to completely overloaded. Since our algorithm is designed to provide predictable timeliness both for underload and overload situations, this setup can effectively demonstrate the strength (or weaknesses) of our algorithm.

Qstream has a command line option to disable or enable frame display for the videos. We repeat the above set of experiments once with frame display disabled in Qstream and once with frame display enabled. Disabling frame display helps us to eliminate the effects of the Xserver from our results. This is important because the Xserver has a coarse grained event dispatching mechanism (we have observed that a single frame display operation can take several milliseconds) as compared to Qstream. Therefore, it is important to separate the effects of the Xserver from our results so that we may better understand the effectiveness of our approach. Further, comparing the results of the experiments with frame display enabled with those of the frame display disabled helps us to get a better idea of the extra overhead introduced by the Xserver alone.

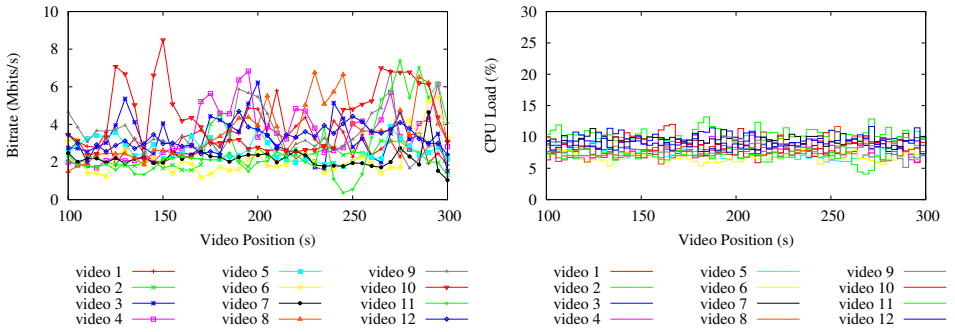
All our benchmarks use a video taken from a movie DVD converted to MPEG-4 format. The movie has a bitrate of 2907 kbits/s, 704x352, 12bpp

resolution and a frame rate of 24 frames per second. Each of the players plays different sections of the movie. Since the video is of variable bit rate and each section has a different bitrate profile, this setup is equivalent to playing different videos in different players. Each stream was played for 300 seconds (5 minutes). For all our results, we discard the first few seconds of the run and report the result from 100th to the 300th second.

As the movie has variable bit rate (thus variable processing requirements) over the duration of the run, running multiple videos represents a fairly complex workload - the complexity arising from the following issues:
(a) The CPU requirements of each of the players varies considerably with time. No specific mathematical model is known to represent their requirement to a reasonable accuracy. For example, Figure 3.4(a) shows the dynamics of stream bitrate for twelve Qstream players playing twelve different streams. Clearly, there is considerable variation in bitrate among the twelve players. The CPU usage, as shown in Figure 3.4(b) varies in strong coherence with bitrate.
(b) Each of the players has specific timing requirements for its numerous events.
(c) All the players are partly CPU intensive and partly IO intensive, representing the kind of a mixed workload that poses a significant challenge for the vanilla 2.6.20 kernel scheduler.
(d) Each of the players have adaptive capability with respect to CPU and network resources. Thus there is an inherent need to coordinate the adaptations across different players for uniform fidelity.

3.4 Benchmark Parameters and Terminology

In each of our experiments, we use certain specific terms to describe specific parameters of our results. We provide the meaning and explanations of these terms below:
(a) **Tardiness:** This can also be termed as the dispatch latency. Recall from Section 2.1.2 that our event dispatcher only invokes the callback for deadline



(a) Per-video bit rate

(b) Per-video CPU usage

Figure 3.4: Dynamics of multi video playback

events after the deadline has passed. The dispatch latency is the difference between the time when a deadline event is actually serviced and the deadline value specified in the event by the application. At regular intervals (a few milliseconds), the worst case difference is calculated for all the events fired in that interval and then this value is averaged for the duration of one second. This metric is very useful in showing how well the kernel is able to schedule the applications such that they are able to hit their deadlines in a timely fashion. A good scheduling algorithm should ensure small values of tardiness, even under overload. However, it is important to note that tardiness values are also limited by application granularity. For example, if in the worst case a single event in the application takes 1 ms to execute, the worst tardiness values of the application can never be smaller than 1 ms. Qstream has a worst case granularity of 1 ms. This was verified by the WCET trace feature in Qstream. This feature measures the worst case execution time (WCET) of all the functions for a given interval of time and then dumps the names of the worst N longest executing functions, where N is provided by the user.

(b) **Average frames per second:** This is the number of frames displayed per second by all the players. This metric gives us an indication of throughput. It is important to note here that throughput is affected by context switches and other overheads. With an increase in the number of players,

we expect the context switches to rise resulting in a decrease in throughput. This is true for *all* cases.
(c) **Frame jitter:** This is the absolute time difference between displaying two consecutive frames. Frame jitter gives us the measure of visible pauses during the video playback (as opposed to frame rate which gives us an idea of the overall smoothness of the video). We timestamp every frame displayed by Qstream. Jitter is then simply calculated as the difference between two successive timestamps. At regular intervals, the worst case difference is calculated for all the frames displayed in that interval and then this value is averaged for the duration of one second.
(d) **Kernel context switch rate:** It is the number of context switches performed by the scheduler per second. This value is directly read off from the `/proc` filesystem and then averaged for every second.

The next few sections describe the various stages of our evaluation. First, we evaluate the results of playing a single video application on a vanilla kernel. This gives us the base case (with no extra overhead of context switches) against which we would compare all the rest of our results. Next, we revisit the scenario of playing multiple instances of a video application over the vanilla kernel. This time we use Qstream as a performance measurement tool. In the later sections we evaluate the performance of our user level and kernel implementation of the fairshare scheduling and our cooperating scheduling algorithm. In each case, we compare our results with that of the single player performance. We also analyze the performance of our algorithm when playing a single high definition video together with running a best effort video encoding job. Lastly, we describe the preliminary results of our implementation involving multiple cooperative domains. This is still a work in progress and a complete implementation of this algorithm with the kind of results we expect to have from it remains a future work.

3.5 Single Player Playing Multiple Videos

Before we describe our results with multiple players, we discuss the results of the simplest case where we let one single Qstream application play multiple

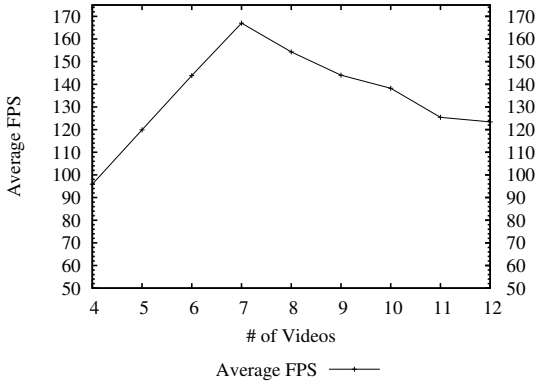
videos. The results in this case reflects the basic overhead of playing multiple videos but has no extra context switch overhead. Thus, this case serves as our *base-case* for comparing all other cases. Figure 3.5 shows the result of playing an increasing number of videos, from four videos to twelve in each run, when frame display has been disabled. Figure 3.6 shows the same results when frame display has been enabled.

There are some interesting points to note from the figures. First, in the overload, the context switches in both cases attains a steady value. For frame display disabled case, its about 100 context switches per second. When frame display is enabled, this value evens out at about 700 context switches per second.

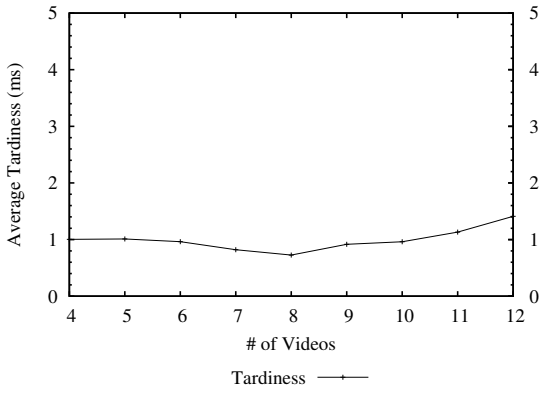
In the underload, for both cases, we see very large number of context switches per second. We have observed this result for all the cases of our experiments. It can be attributed to the *opportunistic nature* of the kernel in the underload. In the underload, the kernel services the interrupts and softirqs as and when they fire. There is no batching up of requests. As a result, context switches increase. In the overload, the kernel batches up requests together and this results in a constant number of context switches.

A look at the throughput (frames per second) also reveals interesting facts. First, when the frame display is disabled, it is clear that application adaptation (frame dropping) does not kick in until we play eight videos. Even with seven videos, all players play full framerate (24 fps $\times$ 7 = 168 fps). However, as we increase the number of videos, overload increases and all players start to drop frames. With twelve players, the overall FPS drops to 124 fps. When frame display is enabled, we see a similar trend. However, due to the extra overhead of the Xserver, we see a greater impact on the throughput. With twelve players, the throughput reduces to 118 fps.

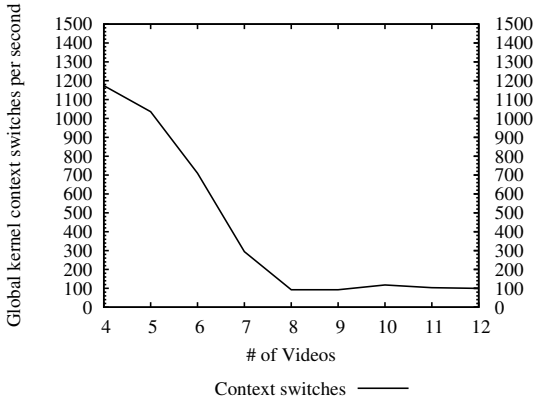
The tardiness starts with higher values in the underload, continues to decreases as the CPU reaches saturation and then increases again at overload. This can be attributed due to the soft-timer effect. In the underload, Qstream players sleep while waiting for data from the socket. This IO sleeping decreases the soft timer granularity (while it sleeps, some deadlines may expire which the application may not be aware of). As the load increases,



(a) FPS Throughput



(b) Tardiness



(c) Context Switch Rate

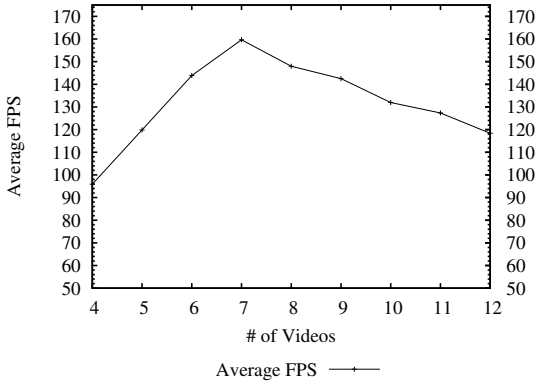
Figure 3.5: Single Qstream: without frame display.

the frequency of sleeping decreases, increasing the soft timer granularity. At overload of course, more and more events gets stacked on to one another (they have deadlines pretty close to each other). Hence the corresponding delay in servicing these deadlines results in increasing tardiness. We do see however that the overall tardiness values tends to be higher when frame display is enabled than when it is disabled.

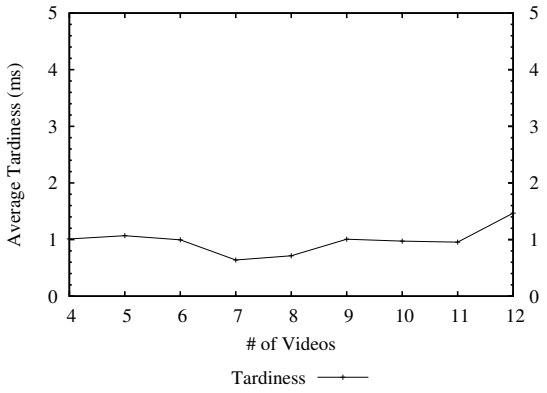
In Figure 3.7 we report the combined frame rate plot for all videos when playing twelve videos at a time. We observe that the videos have uniform frame-rate during the duration of playback. Thus, in spite of having extremely variable bitrate and CPU requirements across them and over the duration of playback (see Figure 3.4), our adaptation technique is able to achieve uniform fidelity across all the videos. This is a formidable scheduling challenge for any conventional scheduler as the CPU requirements of each of the videos are extremely variable even though their frame-rates are uniform. In Section 3.8.2, when we describe our results with multiple Qstream players cooperating using kernel `coop_poll`(), we show that we can achieve very similar results.

3.6 Multiple Qstream Applications Under the Vanilla Kernel.

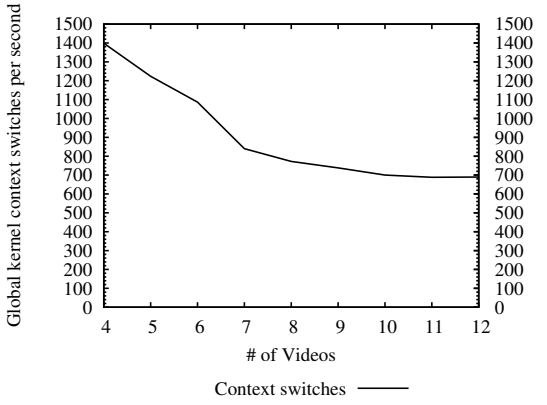
This section describes our experiences with running multiple instances of Qstream over the vanilla kernel. In this experiment, we ran an increasing number of Qstream players, from four players to twelve players in the client machine running the vanilla 2.6.20 kernel and measured the average tardiness for each run. The CPU is underloaded with four players. The load gradually increases until at six players the CPU is just saturated and the adaptation mechanism in the players kicks in. At twelve players, the CPU is completely saturated and all the players are adapting to the available CPU. Figure 3.8 shows the values of average tardiness plotted against increasing number of players. Clearly, as the number of players increases, the tardiness value increases quickly. With twelve players, the tardiness value reaches 1.4



(a) FPS Throughput

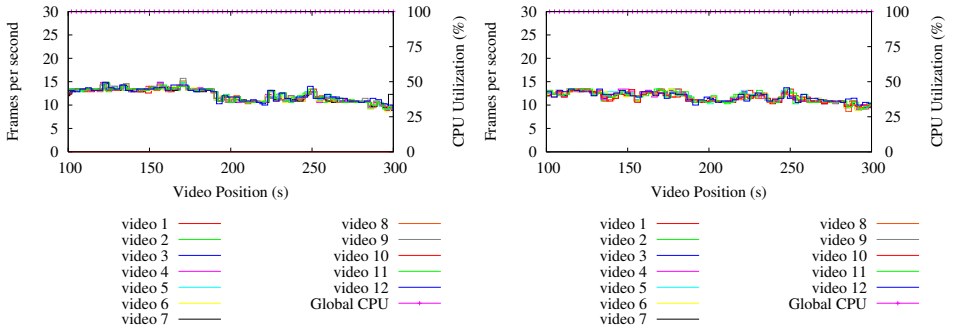


(b) Tardiness



(c) Context Switch Rate

Figure 3.6: Single Qstream: with frame display on.



(a) Frame Display Disabled

(b) Frame Display Enabled

Figure 3.7: Single Qstream: FPS and CPU load for all videos.

seconds resulting in visible long stoppage of the video and very poor fidelity across all the video players.

Figure 3.9 shows the FPS of each of the players and the overall CPU load when running ten independent players at a time. The CPU load is shown on the right hand side scale and the FPS on the left hand side. As is expected, the CPU is fully saturated all the time. However, it is clear that the FPS varies widely between streams and even for the same stream, varies considerably with time. There are periods when one of the streams is completely stuck. This clearly brings forward the weaknesses of the CPU scheduling algorithm of the vanilla 2.6.20 Linux kernel.

So far, we have described the results of running a single Qstream player playing more than one video and multiple Qstream players, each playing a single video over the vanila kernel. In the next section, we talk about our fairshare scheduling algorithm. First, we evaluate the prototype of the algorithm designed at the user level. Then we describe the results of the implementation of the algorithm in the kernel.

3.7 Fair Share Scheduling

In this section, we analyze the performance of our fairshare scheduling algorithm. We first benchmark the performance of the fairshare algorithm that

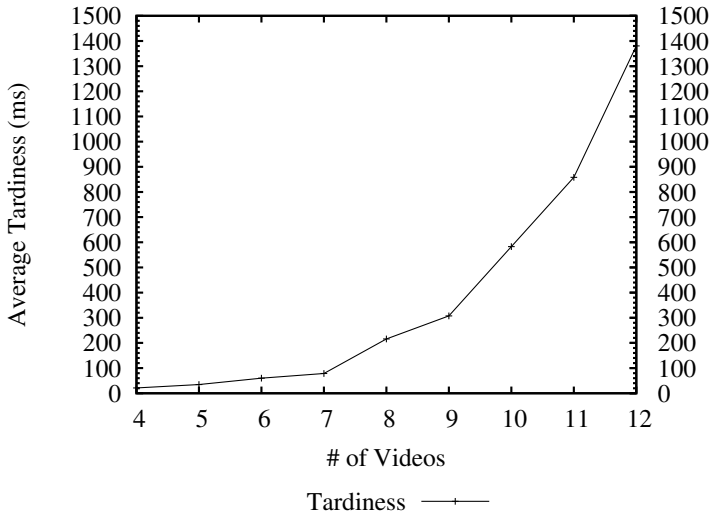


Figure 3.8: Multiple independent Qstream applications: average tardiness on the vanilla kernel

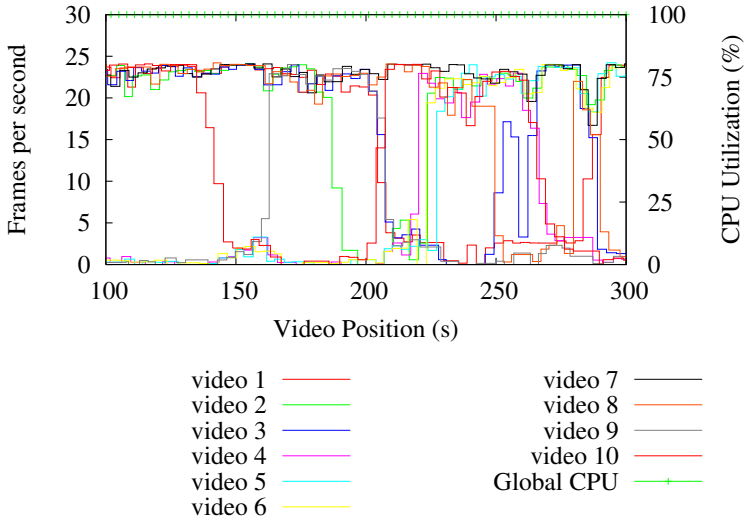


Figure 3.9: FPS of all ten players and the global CPU usage under the vanilla Linux kernel.

was already available as a user level prototype and then do it for the kernel scheduler.

3.7.1 User Level Shared Memory Prototype Implementation.

In this section, we provide a brief overview of the results obtained after benchmarking the user level fairshare algorithm prototype. We believed that the user level implementation will have a higher overhead than the corresponding implementation in the kernel. Our experimental results verifies our belief.

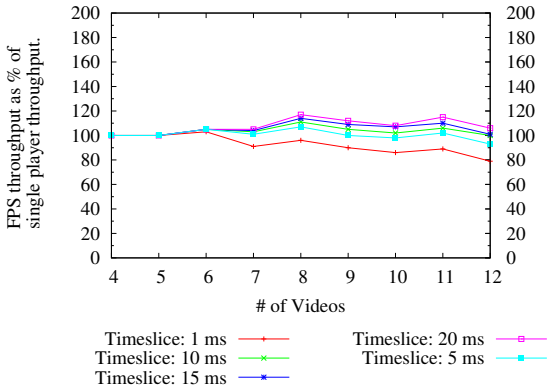
The experimental setup is the same as before. The kernel however does not play any role in each of the following results. For simplicity, we only show the results with frame display disabled. Figure 3.10 shows the plots for throughput, tardiness and context switch rates.

From Figure 3.10(c) we see that, in the user level, the context switches are extremely high. We believe that there are two main reasons for this.

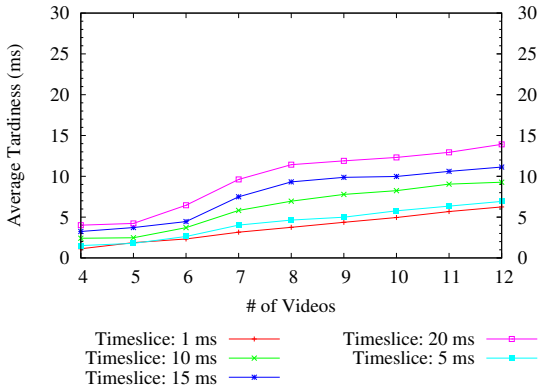
First, in the user level, we use condition variables for implementing the yielding logic. After careful analysis, we found that user level condition variable sleep/wakeup yield does *not* have a 1:1 relationship with kernel context switches. In fact, for every user level yield, there are more than one kernel context switches. We elaborate this point later in Section 3.8.1 where we analyze the user level cooperative scheduling performance.

Secondly, in the user level, we use a slightly different formula for calculating the timeslice for a particular process. Instead of the formula given by Equation 2.2 for the kernel scheduler, we use the Equation 2.1 for the user level implementation. This has already been discussed in Section 2.2.2.

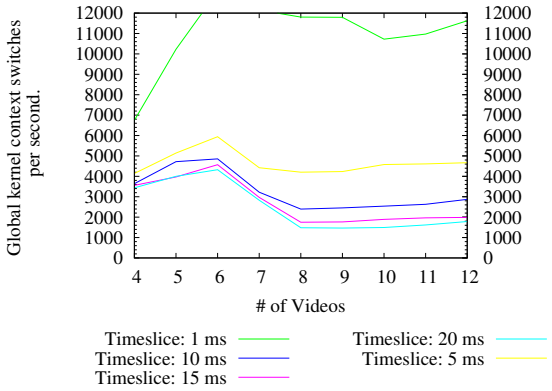
Thus, there would be roughly two times more context switches per second for the user level implementation as there would be in the kernel implementation. This is exactly what we see from the figures. For example, from Figure 3.12(c), we see that in the kernel implementation with timeslice of 10 ms and 12 players, there are about 1300 context switches per second. In the userlevel case however, for the same period and number of players, there



(a) Throughput vs Monolithic (single player case)



(b) Tardiness



(c) Context switches

Figure 3.10: Multiple Players: User level fairshare without frame display.

are 3000 context switches (Figure 3.10(c)).

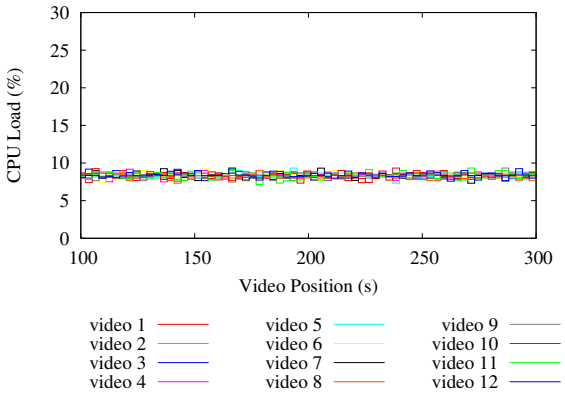
The increased number of context switches has a direct impact on throughput. Comparing Figures 3.10(a) and 3.12(a), we see that the decrease in throughput with respect to the single player case is more for the user level implementation than for the kernel implementation.

For completeness, in Figure 3.11 we show the cpu usage and fps rates for 12 videos under our fairshare scheduler. We see that each player gets a uniform share of the CPU as can be expected from a fairshare algorithm. The frame rate varies widely across the 12 players. This is because each of the videos have different bitrates and hence different processing requirements. Hence, giving fair allocation of CPU resource does not translate to uniform quality (in terms of frame rate) across all the applications.

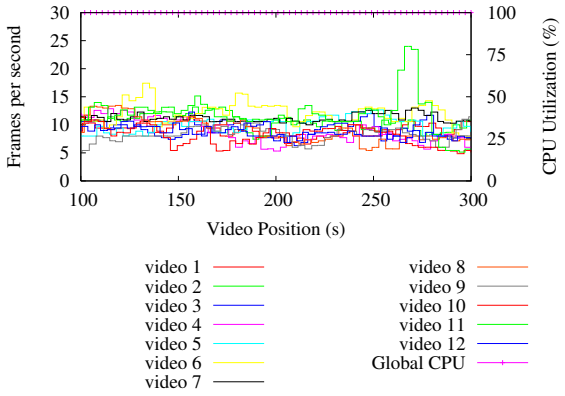
In the next section, we evaluate the performance of the fairshare algorithm implemented in the kernel.

3.7.2 Kernel Implementation.

In this section, we analyze the results of our kernel fairshare implementation. In each of the experiments, we vary the number of Qstream players, from four players to twelve players. We also vary the scheduling granularity (the scheduling period) along with it. Recall from Chapter 2, Section 2.3.3 that our implementation allows the user to change the scheduling period through the `/proc` filesystem. Through our scripts, we write an appropriate value (in microseconds) of the period in `/proc/sys/kernel/bvt_sched_period_us`. For these experiments, we use the following values for the scheduling granularity: 1 ms, 5 ms, 10 ms, 15 ms and 20 ms. It is important to note that in this experiment, the Qstream applications are all running independently and are neither exposing their deadline information to each other through the kernel nor through shared memory. Figure 3.12 shows the combined results for all the experiments when the frame display is disabled. Figure 3.13 shows the same results when the frame display is enabled. Also note that when the frame display is enabled, the Xserver is also run as a fairshare task.

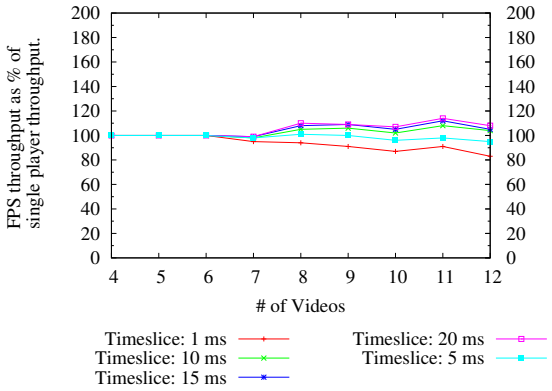


(a) CPU load for all videos

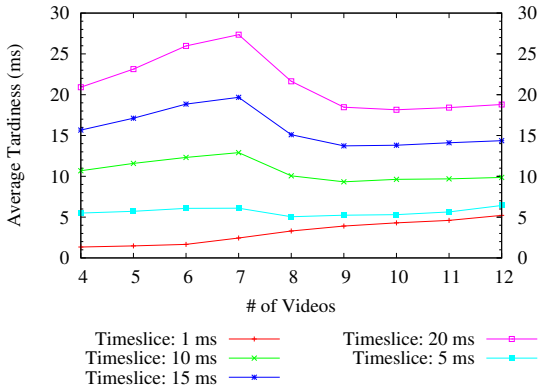


(b) FPS and CPU load for all videos

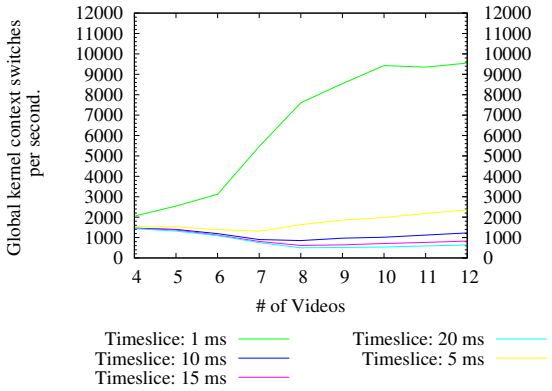
Figure 3.11: Userlevel fairshare algorithm with 12 players and 10 ms period: FPS and CPU load for all videos without frame display.



(a) Throughput vs Monolithic (single player case)



(b) Tardiness



(c) Context switches

Figure 3.12: Multiple Players: Kernel Fairshare without frame display.

Figures 3.12(a) and 3.13(a) show the overall throughput normalized with respect to the throughput of the corresponding single player case (Figure 3.5(a) and Figure 3.6(a)). Since we have multiple Qstream applications, we expect that the increased number of context switches would effectively reduce the overall throughput. We see that this is indeed the case when the scheduling granularity is set at 1 ms. In all other cases, we see that the throughput is in fact *more* than that of the single player case. This we believe, is the result of amortization of the cost of context switches at a relatively large period as we explain below. Recall that in the single player experiment, the player switches between the videos at a much more fine grained scale. This translates to very low tardiness values as we have observed in Figures 3.5(b) and 3.6(b). For the kernel fairshare scheduling, the Qstream applications context switch at a much coarser level depending upon the scheduling period. On one hand, this results in higher tardiness values (compare Figures 3.5(b) and 3.12(b)) but on the other hand, this also results in better cache and TLB utilization, increasing the throughput. Note that when kernel is switching between different applications, each context switch is potentially more expensive as this results in TLB flushes. In single player case, since all the videos are running under one address space, there is no TLB flush. Thus, larger periods in a multi-application scenario amortizes the cost of context switches resulting in a perceptible increase in throughput. When we decrease the scheduling period however, the context switches take place at a finer level, resulting in better tardiness but at the same time, less efficient cache/TLB utilization. Hence, the results with a 1 ms scheduling period shows better tardiness but worse FPS throughput.

Figures 3.12(b) and 3.13(b) shows that the tardiness of the runs in the overload are a direct function of the scheduling period. Each player runs for the duration of their timeslice and then context switches to the next player. With scheduling period T and N players (we assume that all the cpu time is allocated for the players only), fairshare timeslice is given by Equation 2.2 as has been discussed in Section 2.3.3.

In the worst case, when an event deadline of a player expires at the moment when it is context switched out, the event gets delayed by the

tardiness value given by

$$tardiness_{worst} = ((N-1)/N) \times T \tag{3.1}$$

Since, N is constant for a particular run (in overload, the players do not sleep), we see that the tardiness is directly proportional to the period T. However, when T is sufficiently small, say 1 ms, the overhead of a very large number of context switches (in terms of cache pollution and TLB flushes) effects the tardiness. In that case, Equation 2.2 no longer holds. In Figures 3.12(b) and 3.13(b), we can see that with period 1 ms, the tardiness gradually rises until it reaches 5 ms with twelve players. If the above equation were to hold, tardiness would have been roughly equal to 1 ms at all overload conditions.

In the underload however, since the players sleep on IO, they miss deadlines and we see high tardiness values. This situation is the same as the single player case and has been described in Section 3.5. However, it is important to note that the scheduling period will also have some impact on the tardiness even at underload. A player may not be scheduled to run until the timeslice for the currently executing process expires. The timeslice of a task can be as large as the period since players spend most of their time sleeping in underload and are hence out of the runqueue.

Comparing the tardiness results with the user level implementation case (see Figure 3.10), we see that the later has a better result than the kernel implementation. In our belief, this is a direct consequence of a finer grained context switch coupled with the slightly different way we implement the algorithm in the user level. At the user level, when a cooperative task sleeps[5], we take into account the nearest deadline among all the tasks and adjust the sleeping duration accordingly. This is different from the kernel implementation where we do not take into account any deadline information at all. All context switches take place based on the fairshare allocation between tasks. The tasks do not have any information about their mutual deadlines. Thus, whereas in the kernel fairshare implementation in the underload case, the

[5]Sleeping occurs when none of the tasks have any unserviced asap events left.

sleep duration is based on their own deadlines; for the user level implementation, the sleep duration is adjusted based on the nearest overall deadline event. Hence, the result.

Looking at the figures for the context switches we see that, as expected, the context switches are directly proportional to the period. Since, according to our implementation, the minimum CPU time allocation per task is 100 micro seconds, the ideal context switch rate is given by the following formula:

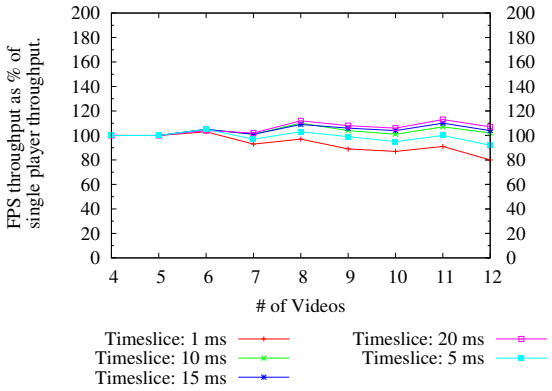
$$c\_rate_{ideal} = min(N \times 1000/T, 10000) \tag{3.2}$$

where c_rate$_{ideal}$ is the ideal context switch rate per second, N is the number of players and T is the scheduling period in ms.

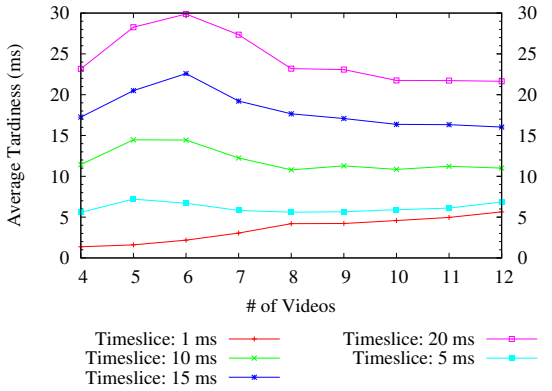
Note that with one context switch at every 100 micro seconds, we have 10 context switches per 1 ms, or 10,000 context switches per second. This is as high as it can get. With 10 players and a period of 1 ms, each player gets the minimum allocation. With an increasing the number of players, the context switch remains the same as the players still continue to get an allocation of 100 micro seconds. Thus, in the figures, we see that with 1 ms period, the context switch rate becomes flat after 10 players. It is also important to note that in practice, we never reach 10,000 context switches. The maximum we get is about 9,600 context switches. This, we believe is due to the high resolution timer latency issues. With 9600 context switches, a single context switch occurs at every 104 micro seconds. Thus, the timer latency turns out to be 4 micro seconds. We consider this to be a very good performance for the high resolution timers and a major achievement by the kernel designers.

We also observe that the context switches with frame display enabled is slightly higher than that when frame display is disabled. This is due to the same reasons as was described for the single player case in Section 3.5.

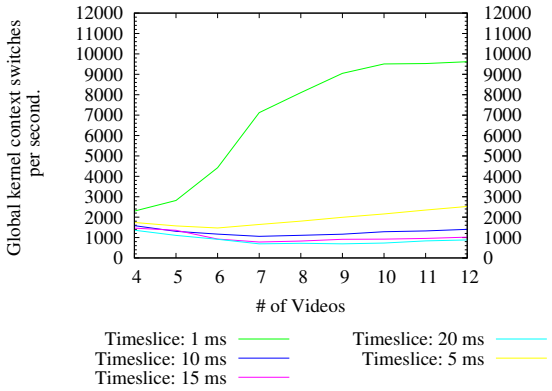
Finally, if we compare Figures 3.5(c) and 3.12(c), we see that with 12 players, 1 ms scheduling period and frame display disabled, fairshare scheduling results in 9500 more context switches than the single player case (9600 context switches as opposed to 100 context switches per second). How-



(a) Throughput vs Monolithic (single player case)



(b) Tardiness



(c) Context switches

Figure 3.13: Multiple Players: Kernel fairshare with frame display on.

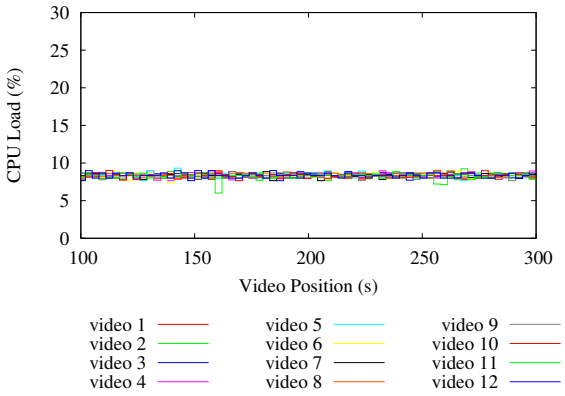
ever, if we observe Figure 3.12(a), we see only a 20% reduction in throughput. Hence, 9500 more context switches only decreases the throughput by a mere 20%. It indicates that context switches are not as expensive in a modern kernel on a modern hardware as it used to be. This is an important result from our experiments. In Section 3.8.2, when we discuss kernel cooperative polling, we will show that it is possible to get higher throughput and better tardiness even with the same amount of context switches. It is possible to do so if the context switches occur in an *informed* fashion, which is the essence of our cooperative scheduler.

For completeness, in Figure 3.14 we show the cpu usage and fps rates for 12 videos under our fairshare scheduler. We see that when playing 12 videos under our fairshare scheduler with 10 ms period, each player gets a uniform share of the CPU (Figure 3.14(a)) whereas their FPS rates vary widely (Figure 3.14(b)). This is because of the same reasons as has been described in Section 3.7.1 while analyzing the same results for the user level implementation.

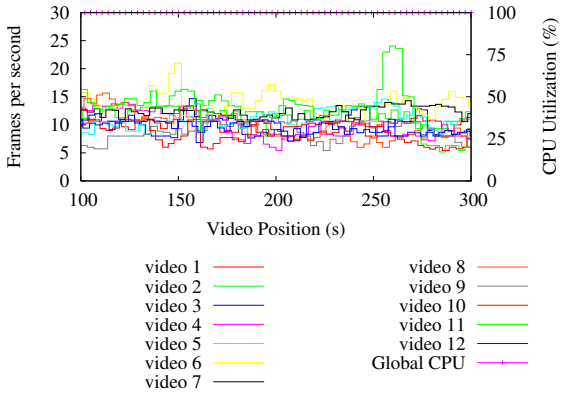
In the preceding section, we evaluated our fairshare algorithm. There was no cooperation between the tasks in this algorithm. In the next section, we evaluate our cooperative polling algorithm. In this algorithm, the cooperative time sensitive tasks cooperatively share the CPU among themselves through the new primitive `coop_poll`().

3.8 Cooperative Polling

In this section, we analyze the performace of our cooperative polling algorithm. First we give a brief overview of the performance of our user level prototype. The prototype was already available to us and was useful for evaluating the performance of the cooperative scheduler at work before we implement it in the kernel. In the later section, we describe the performance of our kernel implementation. Note that, in the kernel, we combine our cooperative polling approach with the fairshare scheduler. The fairshare scheduler ensures overall fairness across all the tasks in the system. This we refer to as policing. The cooperative scheduler preferentially treats real time



(a) CPU load for all videos



(b) FPS and CPU load for all videos

Figure 3.14: Kernel fairshare with 12 players and 10 ms period: FPS and CPU load for all videos without frame display.

tasks and together with their deadline information ensures that they get predictable timeliness within the limits of the fairness imposed by the fairshare scheduler. However, note that in our experiments, we use a homogeneous set of exactly the same application(Qstream). Therefore the policing never actually kicks in. All our Qstream applications are actually well behaved. In general however, in any system we will have a mix of heterogeneous cooperative applications and therefore, policing will play an important role in ensuring fairness in the system. Modification of any other off the shelf application to fit in our cooperative polling framework is beyond the scope of this thesis (see Chapter 5). At the user level, we can not enforce policing and therefore cooperative polling was implemented without any policing mechanism.

3.8.1 User Level Shared Memory Prototype Implementation

In this section, we give a brief overview of our user level implementation results. Each of the players in this experiment share their event deadline and asap event priorities with each other through a shared memory file. They yield to the appropriate process using the condition variable semantics. Figure 3.15 gives the overall results for running increasing number of players (from four to twelve). Comparing Figures 3.15(b) and 3.5(b), we see that the tardiness results are comparable. However, the context switches are extremely high. We were first confused by observing this high level of context switches. To get a better idea, we instrumented Qstream to report the rate of user level yields, which is the number of times `qsf_coop_yield()` is called per second. `qsf_coop_yield()` in the user level prototype replaces calls to the `coop_poll` system call in the kernel implementation and has been described in detail in Section 2.2.1. Figure 3.17 shows the average number of yields per second for each number of players. We see that this figure matches reasonably with Figure 3.19(c), the number of context switches in the kernel implementation. We conclude that the extra number of context switches results due to the way condition variable sleep/wakeup semantics

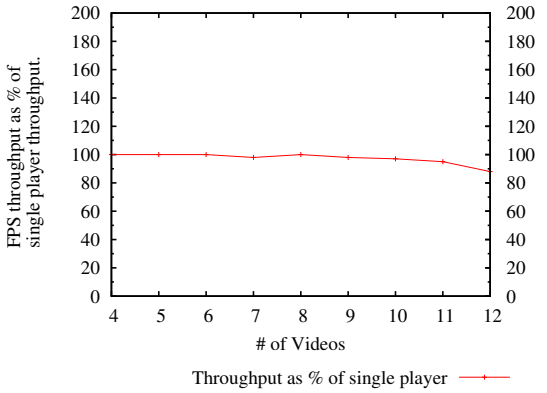
has been implemented in libc. Figure 3.18 shows the ratio of kernel context switches to user level yields. Ignoring the underload cases and the basic amount of context switches in overload (as shown in Figure 3.5(c)), we see that the ratio varies approximately between 2.25 and 2.5.

A high value of context switches decreases the throughput by as much as 20% from the single player case as shown in Figure 3.15(a).

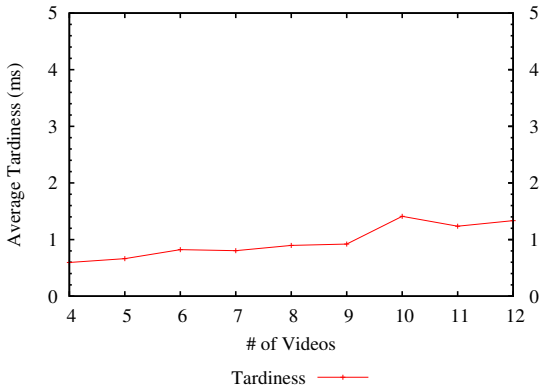
Figure 3.16 shows the overall frame rate of all the players and the global CPU usage. It is clear that in our user level implementation, cooperative scheduling maintains uniform fidelity across all the applications through sharing of event information. We were excited to see this result. If the same cooperative logic could be implemented in the kernel, we hoped to see a similar (or even better) performace. Further, a kernel implementation through a system call interface will be able to actually enforce policing when tasks misbehave or fail to cooperate properly. In the next section, we describe our results with the kernel implementation.

3.8.2 Kernel Implementation With Policing

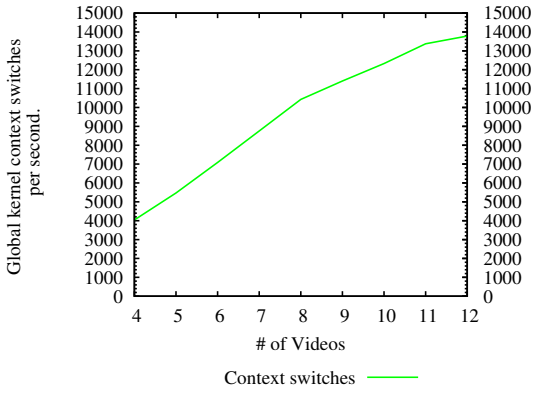
In this section, we analyze the performance of our cooperative polling mechanism working in combination with fairshare scheduling. This combined approach is the most important contribution of this thesis. In this experiment, we vary the number of players, from four players to twelve. We also vary the fairshare scheduling granularity: 1ms, 5 ms, 10 ms, 15 ms and 20 ms. Unlike Section 3.7.2 where we discussed fairshare scheduling without any mutual cooperation between the tasks, in this experiment, each of the players share their event information (deadline and asap priority) with each other. Each of the players cooperatively yield to each other with the help of the kernel through the use of the `coop_poll` system call. However, the overall allocation of CPU resources in the system is decided by the fairshare scheduler. This ensures that no task gets more than its fairshare allocation. Further, please recall from the discussion in Section 2.3.3 that all the cooperative processes gets an allocation as a group. They do not have individual allocations.



(a) Throughput vs Monolithic (single player case)



(b) Tardiness



(c) Context switches

Figure 3.15: Multiple Players: User level cooperative scheduling without frame display.

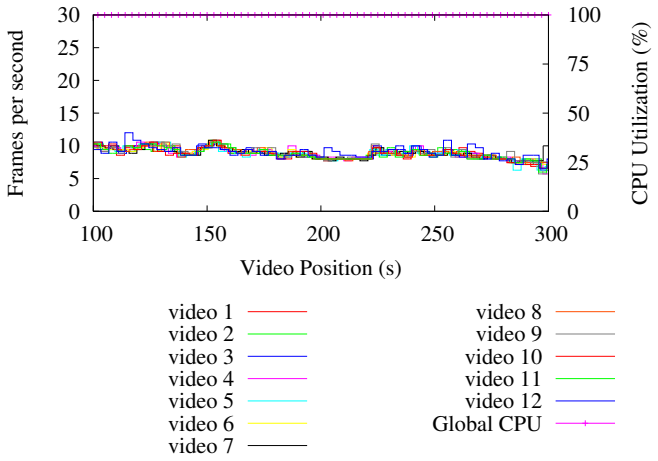


Figure 3.16: Multiple Players: User level cooperative scheduling, overall frames per second without frame display.

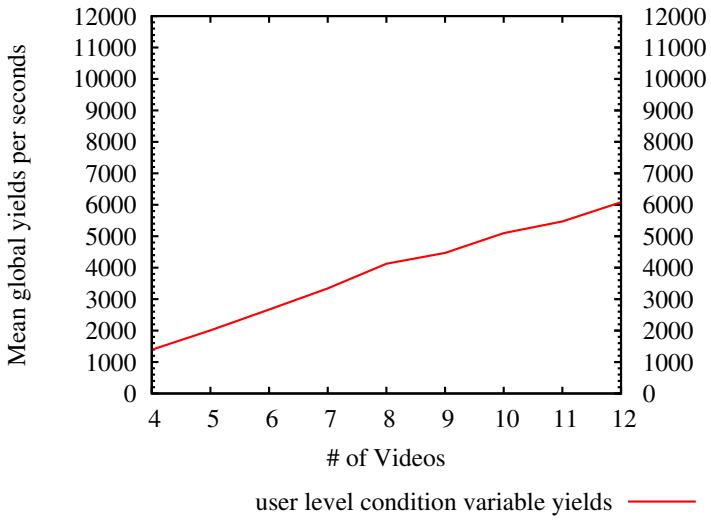


Figure 3.17: Multiple players: Number of user level yields without frame display.

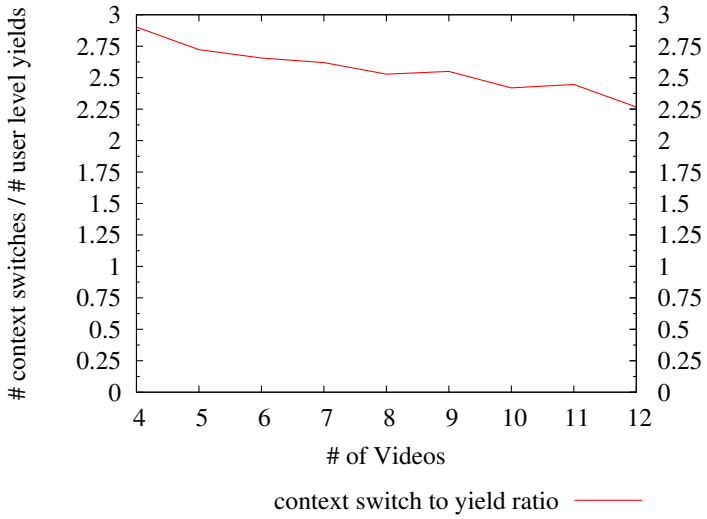


Figure 3.18: Multiple players: Ratio of kernel context switches to user level yields without frame display.

We run two sets of experiments, one with frame display disabled and the other with frame display enabled. With frame display enabled, we run the Xserver as a fairshare task as before. Figure 3.19 shows the combined result of our experiments when frame display is disabled. Figure 3.20 shows the same results with frame display enabled.

If we observe tardiness plots from the above figures, we can see that their values are much closer to the single player baseline (compare Figure 3.5(b) with Figure 3.19(b)) than that of the fairshare scheduler alone, even with the smallest scheduling period (Figure 3.12(b)). Further, comparison of the context switch rates from Figure 3.19(c) with that for the best tardiness case (1 ms period) of the fairshare scheduler in Figure 3.12(c), reveals a very important result. Whereas the fairshare scheduler has almost twice as many context switches compared to cooperative scheduler with policing, its tardiness is almost five times worse (5 ms as opposed to 1 ms).

Whereas a very high number of context switches reduces the throughput by as much as 80% in the fairshare only scheduler, our cooperative polling mechanism coupled with fairshare scheduler does admirably well. We ob-

Comparison	Fairshare Scheduler(1 ms period)	Cooperative Scheduler
Dispatcher Latency	4.3 ms	0.9 ms
Context Switches	9430 per sec	4766 per sec
Throughput as % of single player	87%	95%

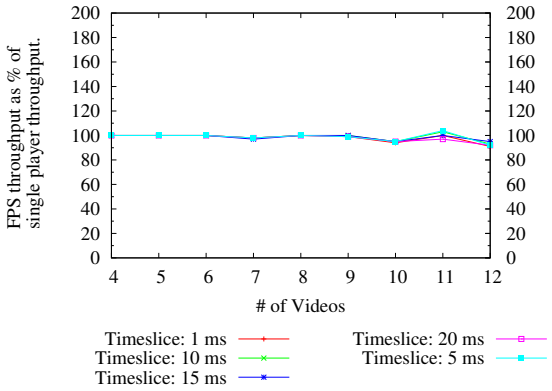
Table 3.1: Comparison of Fairshare Scheduling with Cooperative Polling with 10 Qstream players

serve from Figures 3.19(a) and 3.20(a) that the reduction in throughput is as low as 5% in this case compared to the single player case . We believe that such a small reduction in throughput is not only the result of fewer context switches but also a direct consequence of *informed* context switching. Informed voluntary switching amortizes the cost of cache pollution and TLB flushes, resulting in an increase of throughput. In a pure fairshare scheduler, applications are not aware of the kernel scheduling decisions, resulting in untimely involuntary context switches which is more expensive.

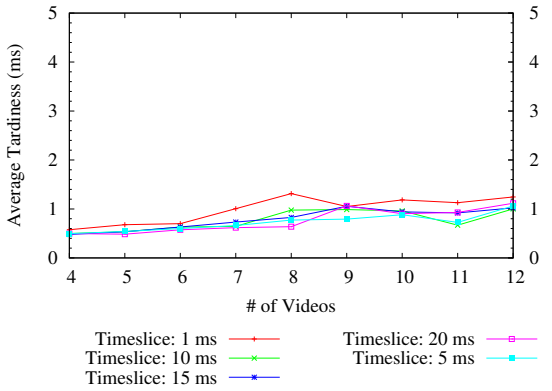
One another interesting thing to note from the context switches is that they are independent of the scheduling period, unlike in the pure fairshare scheduling case. This is because all the real time cooperative processes have a CPU allocation as a group and they are free to share this allocation among themselves cooperatively, according to their needs as long as they do not cross the limits of the group allocation.

Table 3.1 clearly brings out the advantages of cooperative polling over pure fairshare scheduling. In this table, we compare fairshare scheduler with 1 ms period using cooperative scheduler. In both cases, we take the example of playing 10 videos at a time. Clearly, cooperative polling has five times better tardiness than pure fairshare scheduling. However, its throughput penalty compared to single player is 8% better than pure fairshare. The context switching in pure fairsharing is also almost twice as much as in cooperative polling.

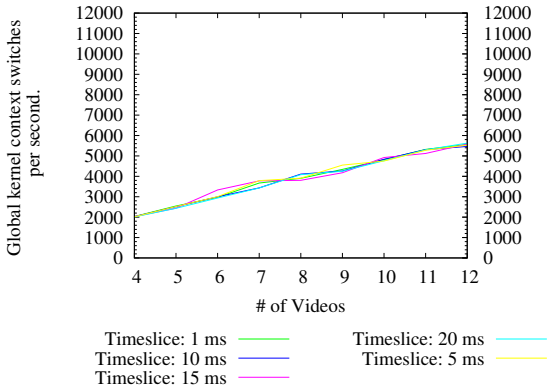
Figure 3.21 shows the combined framerate of all the twelve players playing with scheduling period set to 10 ms and frame display enabled. Clearly,



(a) Throughput vs Monolithic (single player case)

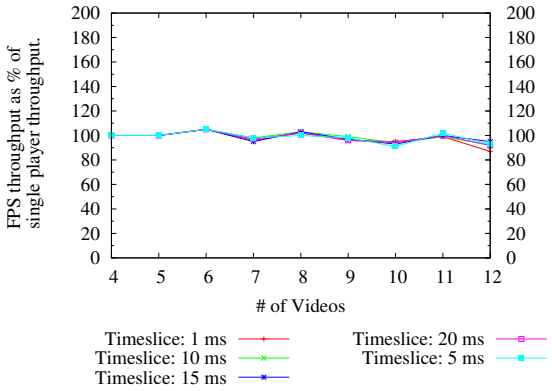


(b) Tardiness

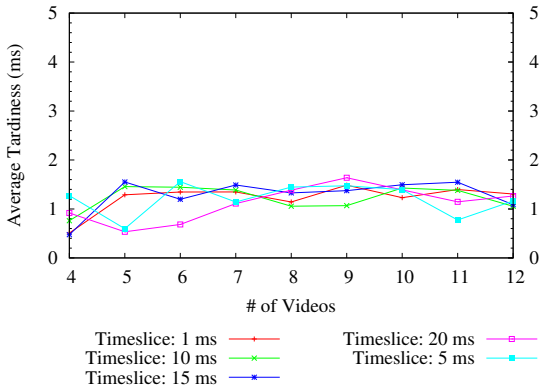


(c) Context switches

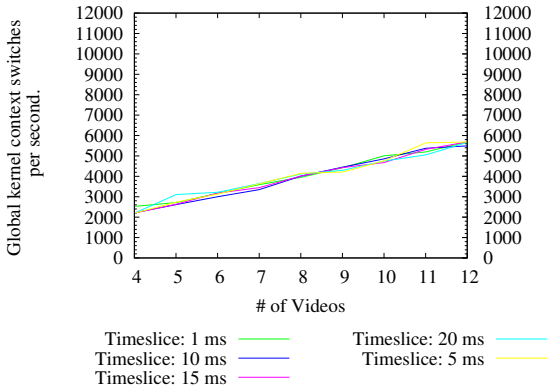
Figure 3.19: Multiple Players: Kernel cooperative polling with policing and without frame display.



(a) Throughput vs Monolithic (single player case)

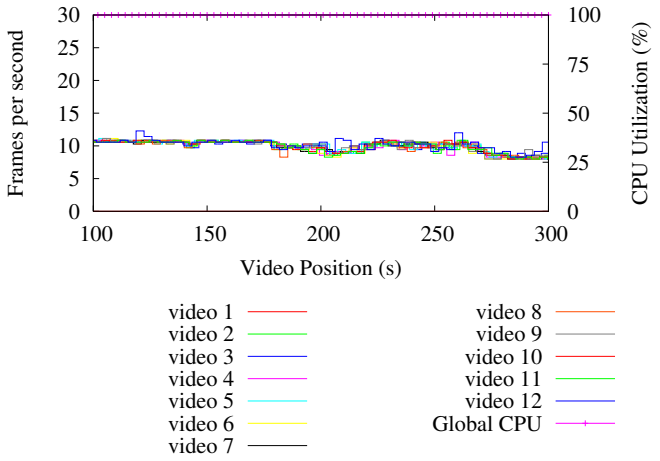


(b) Tardiness



(c) Context switches

Figure 3.20: Multiple Players: Kernel cooperative polling with policing and with frame display on.



(a) FPS and CPU load for all videos

Figure 3.21: Kernel cooperative fairshare algorithm with 12 players and 10 ms period: FPS and CPU load for all videos with frame display.

since each of the players are cooperating using `coop_poll` through the kernel, they have coordinated adaptations with uniform frame rate across all of them.

So far, we have described the results of our fairshare scheduling algorithm as well as our cooperative polling algorithm (with policing). In each of the experiments, we have run variable number of players to vary the CPU load from underloaded to extremely overloaded condition. In the next section, we describe our experiences with running a single high definition video (a single such video in our experiment takes as much as 70% of the CPU) by a single Qstream application. We run our experiment both on the vanilla kernel as well as on our cooperative-fairshare scheduler. We run a best effort video encoding work in parallel to completely saturate the CPU.

3.9 Results of Playing a Single High Definition Video in CPU Saturated Condition

In this section, we describe our experiments with playing a single 1080p high definition video in Qstream and running a best effort video encoding task in parallel. Together, they completely saturate our client. The goal of this experiment is to evaluate our algorithm for a more common scenario where users watching a high definition video perform some video/audio encoding work in parallel.

We found it difficult to get a freely downloadable long 1080p high-def video. The one that we use has a bitrate of 679.2 kbyte/s, resolution of 1440x1080 with 24 bits per pixel and frame-rate of 25 fps. When running alone, the single video requires roughly 70% of the CPU. The sample video that we encode is a standard MPEG-2 video ripped from a movie DVD, resolution 720x480, with 24 bits per pixel, bitrate of 743.5 kbyte/s and frame rate of 24 fps. The encoding job converts this video to SPEG format (Qstream compatible video) of equivalent quality.

We run our experiments both in the vanilla 2.6.20 kernel and on our cooperative-fairshare scheduling algorithm. Please note that since this is a single video running with no other real time tasks in parallel, there is no other real time process to cooperate with. Hence, effectively, the real time task gets a fairshare allocation of CPU along with the other best effort task. Also note that when running the experiment under our scheduler, we run the Xserver as a best effort process. When running the experiment under the vanilla kernel, we leave the responsibility of scheduling the Xserver totally on the vanilla kernel heuristics.

3.9.1 Vanilla Kernel Performance

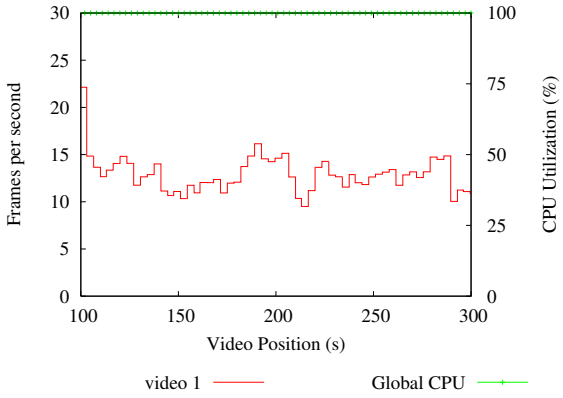
In this section we describe the results of our experiments on the vanilla 2.6.20 kernel. Figure 3.22 shows the results of our experiment. From Figure 3.22(c) we see that the vanila kernel heuristic is treating the Qstream player as a CPU intensive task and is therefore equally sharing the CPU

between both the tasks. Since the video itself requires about 70% of CPU, the player is forced to adapt, resulting in frames dropping, as is evident from Figure 3.22(a). Figure 3.22(b) shows the average value of tardiness (or dispatcher latency) for the duration of playback. We observe that the tardiness consistently reaches 100 ms which is an extremely high value. Moreover, the vanilla kernel has no adjustable scheduling parameter with which one can obtain a better timeliness for the real time task if so desired.

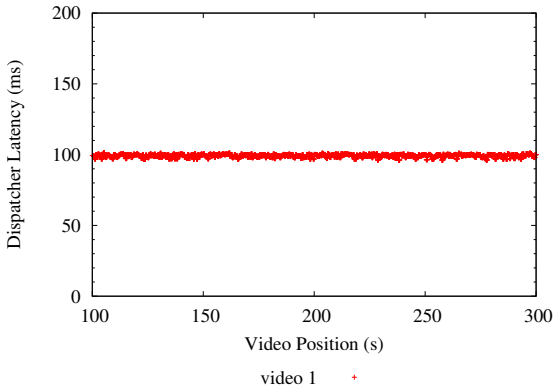
3.9.2 Performance of Our Kernel Fairshare Algorithm

In this section, we run the above described experiment under our cooperative-fairshare scheduler. Note that the Xserver is also run as a best effort task under our fairshare scheduler. Figure 3.23 shows the combined results of the experiment as the scheduling period is varied. From the Figure 3.23(b) we see that the tardiness is a direct linear function of the scheduling granularity (the period). As the scheduling period is decreased, the kernel context switches at a finer level, resulting in improved tardiness. Hence, our scheduler gives the users a tuning knob in the form of the scheduling granularity value (which can be set through the `/proc` filesystem) that can give the desired tardiness required. Moreover, even with a large value of the period, 20 ms say, the tardiness we can achieve is many times better than the vanilla kernel (12 ms as opposed to 100 ms). However, as the downside of decreasing the scheduling period, the number of kernel context switches increases (Figure 3.23(c)). However, it is interesting to note from Figure 3.23(a) that even though context switches increases with increasing scheduling granularity, the overall FPS throughput does not decrease considerably as a consequence of excess context switch overhead.

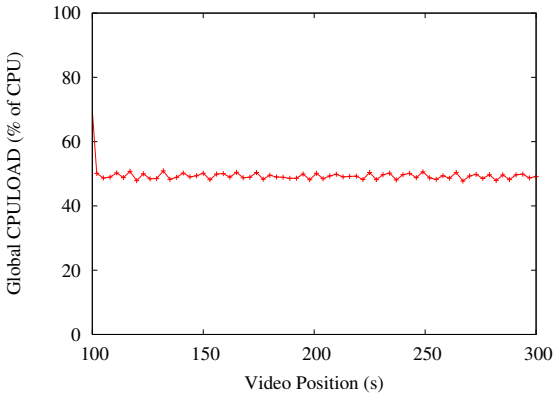
The main evaluations of our implementation ends here. However, we did perform some more experiments to do initial evaluations of our multiple cooperation domain implementation. The next section describes the results of these experiments.



(a) FPS rate and global CPU

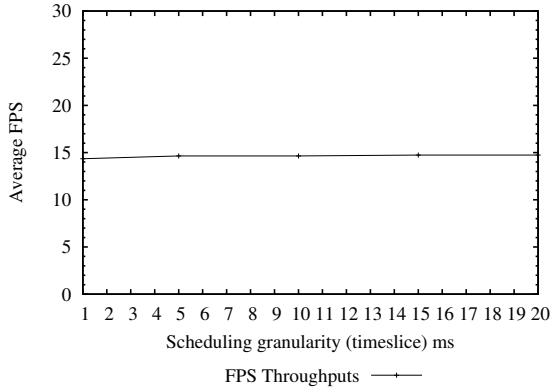


(b) Tardiness

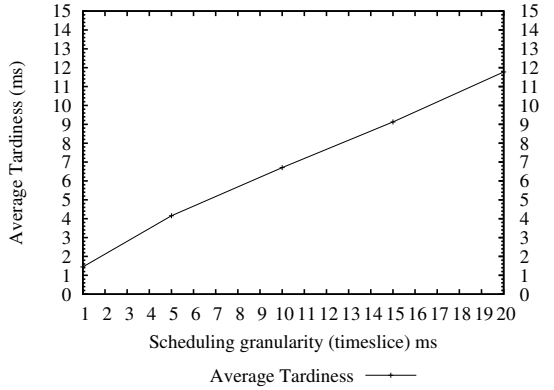


(c) Global CPU usage

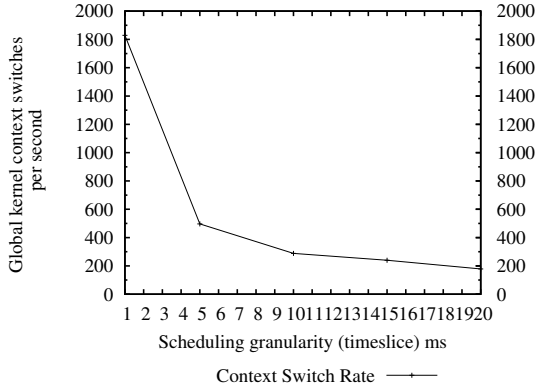
Figure 3.22: Single player with one high definition video on vanilla kernel with frame display on.



(a) FPS throughput as a function of scheduling period



(b) Tardiness as a function of scheduling period



(c) Context switch rate as a function of scheduling period

Figure 3.23: Single player with one high definition video on kernel fairshare algorithm with frame display on.

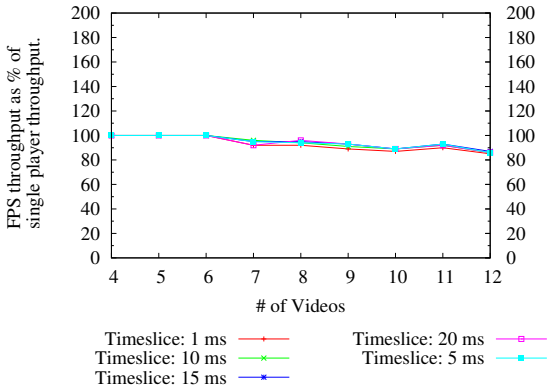
3.10 Multiple Coop-domain Implementation Results

In this section, we provide a brief overview of the results of our experiments over our multiple cooperative domain implementation. This implementation and its purpose has been explained in detail in Section 2.5. This work remain unfinished and the results reported here are only preliminary.

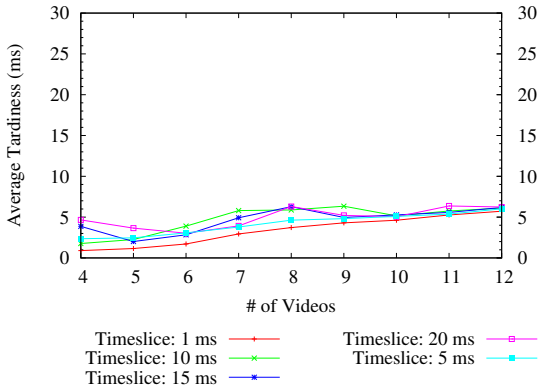
For this experiment, we modified the `coop_poll` interface and added a new integer argument representing the coop domain id. We had to modify the user level Qstream code in order to make use of the new system call interface. We added a new command line parameter to Qstream through which users can specify the coop-domain in which the application should become a member. Further, we modified the benchmark scripts so that each of the different Qstream players seek membership in separate coop domains. The rest the benchmark setup remained the same as before. We performed experiments with both frame display enabled and disabled. However, for brevity, in this thesis, we only report results of the experiments that were performed with frame display disabled.

Figure 3.24 shows the combined results of all our experiments. We see at once that the number of context switches becomes independent of the period at bigger scheduling periods. This is because, with a large scheduling period, the event granularity remains finer than the fairshare period. Since, in this algorithm, the scheduling decision is also based on the earliest deadline in each of the domains and each domain has exactly one task in our benchmarks, context switches solely depend on the event granularity and not on the period. At smaller periods however, the fairshare allocation becomes smaller than the event granularity. In this scenario, as we see in the 1 ms period case, the context switches become a function of the period.

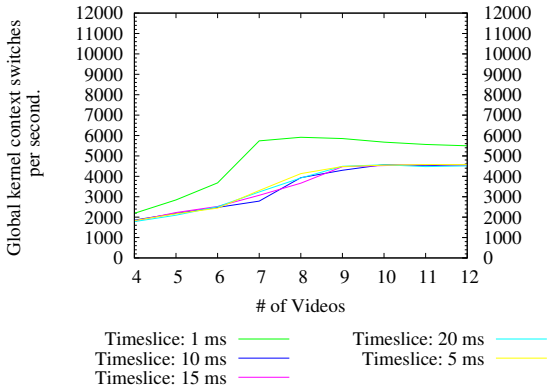
From Figure 3.24(b), we observe that the tardiness values are independent of the scheduling granularity and solely depends on the number of Qstream applications. The reason for this is the same as explained above. If the scheduling period is too large, the event granularity will be much more fine grained than the individual allocations. Hence, tardiness (which



(a) Throughput vs Monolithic (single player case)



(b) Tardiness



(c) Context switches

Figure 3.24: Multiple Players: Kernel multi-domain cooperative scheduling with policing and without frame display.

depends on how fine grained the context switches are compared to the spacing of the event deadlines) will also be independent of the period. However, with a small enough period, the event granularity is coarser than the fairshare allocation per task. We see from the figure that this is indeed the case with a 1 ms period. With a 1 ms period, we see more context switches though not as many as in the pure fairshare algorithm (Figure 3.12(c)).

In terms of throughput, we see that this algorithm does experience a reduction of overall FPS with an increase in the number of players. However, its performance is comparable to that of the cooperative fairshare implementation (see Figure 3.19(a)).

It is interesting to observe that the tardiness in this implementation is not as good as the cooperative fairshare implementation. We believe that a part of this reason could be the mismatch between the task that spent the longest time waiting in the runqueue and the one whose deadline just expired. Recall from Section 2.3.3 that in this scenario we prioritize running the former task as opposed to the later task to ensure fairness in CPU allocation. Due to time constraints, we were unable to complete a full and rigorous analysis of our multi-coop domain implementation and ascertain some of these proposed beliefs through concrete numbers. This work is currently being persued by some other students but nevertheless, it remains a future work for this thesis.

In summary, even though our analysis and design fine tuning were incomplete as far as the multi-coop domain algorithm goes, we were encouraged to see that it did give us a middle ground performance - in between a pure fairshare algorithm and our cooperative fairshare implementation, both in terms of tardiness and throughput. Further, the algorithm inherently does not use asap events for scheduling. Hence, it has the potential to accommodate non-adaptive applications as well.

3.11 Chapter Summary

In this chapter, we have analyzed the performace of the vanilla 2.6.20 Linux scheduler in scheduling multiple real time multimedia applications. We have

shown that this vanilla kernel heuristics is incapable of providing acceptable timeliness for the real time tasks. Further, we have shown that multimedia workloads that are both CPU and IO intensive impose a formidable scheduling challenge to the the existing scheduler.

Next, we gave an overview of the performance of the userlevel prototype of the algorithms available to us prior to our venture with the kernel. We show that our user level prototype gives us the results we anticipated. However, they exhibit extra overhead along with having weak scheduling isolation between tasks which the kernel implementation is capable of addressing.

Next, we analyzed the performace of the various algorithms implemented by us in the kernel. We have shown that a simple fairshare algorithm is capable of providing acceptable timeliness for time sensitive applications, if the scheduling period is properly chosen. However, a very small value of scheduling period can adversely effect the overall throughput as the number of context switches increases to extremely high levels. The overhead of the large number of context switches imposes a practical limit on the best possible timeliness one can get out of a simple fairshare scheduler. We show that a combination of cooperative polling and fairshare algorithm can provide timeliness many times better than a simple fairshare scheduler with much less overhead. Through voluntary preemption, it is possible to amortize the cost of context switching and get better throughput. Preferential treatment of realtime tasks using the information provided by them in the form of event deadlines and priorities can help in getting excellent timing performance for real time tasks. At the same time, cooperative scheduling, when supervised by the fairshare scheduler, ensures fairness of CPU allocation among all the tasks. It also facilitates policing of the misbehaving/uncooperative real time tasks that violate the allocation constraints. Thus, this combined approach ensures predictable timeliness without starvation concerns.

In our experiments, we show that the Xserver in itself imposes a significant overhead and introduces untimely behavior in the system. We believe that it is possible to alleviate the problem to a large extent by modifying the Xserver so as to use our cooperative polling mechanism. This however

remains a future work for this thesis.

Finally, we analyzed the performace of our multi-coop domain algorithm. We propose that, with proper implementation, the multi-coop domain algorithm can provide equivalent performance compared to our cooperative-fairshare algorithm. Yet this new approach would give more portability and flexibility across many other applications as it will bring non-adaptive time sensitive applications under our scheduling regime as well. However, this work remains incomplete in this thesis.

Chapter 4

Related Work

In this chapter, we review some of the related works in this space and compare our work with them. We also cite the relevant papers and references from where we have been encouraged to work on this idea.

4.1 Cooperative Polling and Kernel-Userspece Interactions

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

Our cooperative polling at the user level is largely inspired by the soft timers [4] and the firmtimers [16] kernel-based polling approach. They use trigger states such as kernel entry points to efficiently schedule events (e.g., packet transmission). Cooperative polling extends the benefits of these approaches into the user level. Both cooperative polling and soft/firm timers aim to avoid unnecessary preemption or interrupts.

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

domains: activations are mainly designed for throughput-oriented applications where the upcall model is easier to use, while cooperative polling is mainly designed for time-sensitive applications where polling is a commonly used method to meet timing requirements. Also, with activations, the user level only informs the kernel when it yields the processor, [6] while with cooperative polling, applications also inform the kernel about their deadlines or priorities.

4.2 Operating System Support for Time Sensitive Applications

There is a large body of work that aims to provide operating system support for multimedia and real-time applications [9, 16, 17, 18, 26, 30, 36, 41]. We believe that our work is closely related to the borrowed virtual time (BVT) scheduling algorithm [10] and the SMART scheduler [30]. Like SMART, our algorithm uses a notion of urgency based on application-supplied timing values, while BVT uses a notion of urgency based on *warp* value that is difficult to calculate. However like BVT and unlike SMART, our approach does not require an application estimate of service time and leaves admission control to the user-level. SMART requires service times to perform schedulability analysis, this analysis can be conservative, when SMART sends overload notifications, they are probably too late for the application to react. The comparison of our work against BVT and SMART is described in tabular form for better clarity in Table 4.1.

4.3 Event Driven and Multi-threaded Programming

Although the relative merits of event-driven and multi-threaded architectures remain highly debated over a long period of time [1, 25, 32, 33, 44, 46], generally events are considered to offer better performance while threads are

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

Comparison	`coop_poll`	BVT	SMART
Application Model	Event Driven	Regular	Regular + Event Driven
Specifications	Wake up times and (optionally) priorities	warp (priority) values	Deadlines and service time estimates and (optionally) priorities.
Use of WFQ to ensure fairness	yes	yes	yes
Notion of urgency	wake up time (which we call deadline in this thesis)	warp value	time of completion of work (referred to as deadline in the paper)
At overload	Application informed at wake up time when it needs to yield next	external admission control module	scheduler notifications when system is in overload.

Table 4.1: Comparison of our work with BVT and SMART.

considered to offer ease of programing. In the QStream framework which is used extensively to benchmark this work, Krasic uses events because he feels they are the best match for time-sensitive applications that must quickly respond to external input. For example, the worst-case execution time (WCET) of a job or a response is an important metric for time-sensitive applications. This metric is quite easy to instrument with events. Nevertheless, we believe that it is possible to use non-preemptive threads libraries such as Pth [11] as an alternative for implementing cooperatively-polled applications.

Currently, our model supports applications that are primarily single threaded. Zeldovich et. al. [48] provide multiprocessor support for event-driven programs. Their approach could be directly applied to our work.

Even though events have been used extensively, most event systems have focused on the efficiency and scalability advantages of events rather than

predictable timing. As a result, most event systems do not distinguish between deadline and best-effort events. The PulseAudio [35] sound server uses events that are closest to our model. It uses separate event types corresponding to our deadline and best-effort types, but unlike our model, applications cannot control the order of best effort events.

4.4 Adaptive Multimedia Applications

Our model focuses on time-sensitive applications that can adapt during overload. The QStream video streaming application uses a technique called Priority-Progress to adapt to available network bandwidth [24] and CPU [23]. This technique was inspired by several other works on quality-adaptive streaming [12, 37, 40]. Seda [46] provides a framework for performing overload management primarily for throughput-based applications such as Internet services.

4.5 Recent Kernel Developments

Recently, there has been some activity in the space of redesigning the kernel scheduling algorithm for the Linux kernel. Ingo Molnar and others have proposed and designed a new scheduling algorithm which they have named the Completely Fair Scheduler [27] or CFS scheduler. Certain portions of the CFS scheduler is itself derived from the previous work by Con Kolivas in his Rotating Staircase Deadline (RSDL) Scheduler [19]. The concept of providing fair allocation of CPU time to all tasks in the system is close to our fair share scheduler. However, the idea of cooperative polling which allows cooperation between real time tasks through the kernel and voluntarily yielding the CPU based on deadlines, is still a novel aspect of our work. As far as our knowledge goes, no one as yet has proposed a similar idea. The CFS scheduler was in the implementation and testing phases and began to be incorporated into the mainstream kernel as of version 2.6.23. The CFS scheduler has been better tested and debugged and run by many more users than our code.

For supporting various time sensitive applications, certain new features have been implemented into the Linux kernel. Fine grained kernel preemption, high resolution kernel timer and time keeping mechanisms and tickless kernel design with dynamic ticks [38] are some of them. John Stultz and Thomas Gleixner [15, 42] have done work in the space of kernel timers and timer mechanisms. In our work, we complement and make use of some of these new kernel infrastructures for fine grained timekeeping and process accounting. We also avoid using any kernel features that make use of regular timer ticks, thus making our design consistent for using with tickless kernels.

4.6 Chapter Summary

In this chapter, we have reviewed some of the related work in this space and compared them with ours. Whereas the need for running more and different types of adaptive multimedia applications has increased over time, the kernel scheduling algorithm has not changed significantly to support these newer kinds of workloads. Most of the existing works on multimedia scheduling either depends on CPU time reservations or has complicated heuristics to achieve predictable timeliness. However such approaches are impractical for commodity environments. Recently Ingo Molnar and others have proposed and designed new kernel scheduling heuristics to address some of the challenges we seek to solve. However, as far as our knowledge goes, our idea of cooperative polling based on application deadlines still remains a novel idea in this space.

Chapter 5

Future Work and Conclusion

This chapter reviews some of the important future directions of this work and then finally concludes.

5.1 Future Work

While this work involved a novel approach for scheduling time sensitive workloads along with traditional best effort ones, it was only a good beginning. The work is hardly complete. There are several avenues of future work in this space, some of which has already been discussed in the earlier sections in this thesis. In this chapter, we enumerate a few of the very important ones.

5.1.1 Evaluation of the Fairshare Scheduling Algorithm

We have not performed a thorough evaluation of our fairshare algorithm as a general purpose scheduler. A complete and detailed evaluation of the algorithm is needed. In particular, evaluations with various kinds of pathological workloads e.g., `massive_intr` [43], `HBench-OS` [8], `ocbench` [45], `pipetest` [28], `ringtest` [29], etc., and comparing the results with those of the CFS scheduler [27] would be of great interest. It would also be interesting to quantify the throughput figures for running multiple best effort workloads and comparing them with the vanilla kernel scheduler.

5.1.2 Accomodating Non-Adaptive Time Sensitive Applications

In Section 2.5, we discuss our approach towards accommodating non-adaptive applications within our scheduling regime. In Section 3.10, we discuss our preliminary evaluation of this algorithm. However, this work remains incomplete due to time constraints. Though the results did give us some broad idea on the potential of the algorithm, we were not able to exactly reason out some of the performance figures we obtained from benchmarking the design. A deeper level analysis and understanding with possibly modifying the implementation is necessary.

5.1.3 Implementation of Cooperative Xserver.

Our benchmark results from Chapter 3 show that the Xserver imposes significant overhead and cause poor timeliness for time sensitive tasks, even when scheduled as a best effort application in our fairshare scheduler. Scheduling Xserver has always been a challenge to system programmers. Since the Xserver is based on an event driven programming model, it would be an interesting work to modify it so that it can use our `coop_poll` semantics. With Xserver cooperating with the rest of the Qstream applications, it is our belief that we can achieve much better results. However, since it is not an adaptive application like Qstream, our kernel has to support non-adaptive time sensitive applications. This work was discussed in the previous section. Unfortunately, due to time constraints, both of these two are left as future work.

5.1.4 Benchmark on a More Heterogeneous System

In our performance analysis, we use the Qstream application throughout, as the only cooperative application. It will be definitely an interesting experience to modify certain other applications to use our cooperative polling scheme. Then, we can benchmark our scheduling framework on a more heterogeneous system comprising of different cooperating tasks. This work will become more attractive if these modifications can be performed with

minimal effort. Unfortunately, due to time constraints, we also leave this as future work.

5.1.5 Load Balancing for Chip-Multithreaded Processors.

We have already described in Chapter 2 that our implementation does not support multi core and hyperthreded systems. Even though we have per-CPU data structures that should automatically scale to multi-processor scenarios, we have not been able to get our code running on SMP systems (possibly because of subtle bugs in the code). Support for chip multithreaded processors and a mechanism to load balance the tasks across the processors would be an important future work. One way to load balance is to tackle the problem in the application itself. Event driven applications can assign different colors to each of their events such that events of the same color can be executed in different CPUs without concurrency issues. This idea is very much similar to the Zeldovich et.al., work [48] on supporting event driven programs on multi-processors. However, this idea is still in the early stages and a complete working implementation remains to be done.

5.1.6 Integration of `coop_poll` with `epoll`

An interesting extension of the work is to integrate our `coop_poll` system call with the existing `epoll` system call. Even though the `epoll` system call interface will change significantly as a result of this integration, no new system call will be introduced into the kernel. Further, this will help us to incorporate the status of open file descriptors as one of the parameters of the `coop_poll` yielding logic. Thus, the modified yielding logic will prioritize a cooperative task whose deadline has expired or the task that has the highest priority best effort event ready. The important difference is that now, the best effort events of a task would include its ready file descriptors with priority values attached to them. This would facilitate more fine grained adaptation mechanism with improved timeliness across all the cooperative tasks.

5.1.7 Integration with the 2.6.23 Pluggable Scheduler.

As of writing this thesis, the new CFS scheduler [27] has been merged with the 2.6.23 kernel source tree. The scheduler code of this new kernel uses the notion of pluggable CPU schedulers [20]. It might be possible to integrate our scheduler into the new kernel as a pluggable module. This would help us in getting more and more people involved with this work (since the scheduler can be tested with minimal effort) and further fine tuning and regression testing of our scheduling algorithm.

5.2 Conclusion

In this thesis, we have introduced a new and novel approach to bring time sensitive and traditional best effort applications within a single unified scheduling framework. We have shown that the traditional preemptive scheduling introduces unpredictable timing and poor adaptation behavior for adaptive timesensitive workloads. Cooperative polling aims to reduce unpredictable timing by minimizing involuntary preemption, and it facilitates cooperation between applications by sharing event information such as deadlines and priorities across applications. Our evaluation has shown that this approach together with a simple deadline-based scheduling policy achieves overall predictable timing, and it allows applications to make adaptation decisions during overload that cooperate with rather than get overwhelmed by the kernel's scheduling policy.

However, cooperative scheduling alone can not ensure fairness and avoid starvation. We combine our cooperative algorithm with a fairshare scheduler in order to achieve fairness. This preemptive scheduling prevents the possibility of our cooperative mechanism being abused (either intentionally or otherwise) to gain unfair advantage. Further, unlike existing approaches that have attempted to integrate conventional real-time scheduling algorithms into general-purpose operating systems with limited success, our approach allows time-sensitive and best-effort tasks to co-exist in a tightly unified framework.

We have implemented our scheduling algorithm in the Linux kernel and have performed a series of experiments to evaluate our approach. In our evaluation, we use a quality-adaptive video playback application to demonstrate how complex adaptation policies may be realized. We have also performed comparisons that show the performance and timing benefits of cooperative polling. Our experiments show that cooperative polling leverages the inherent efficiency advantages of voluntary context switching versus involuntary preemption. In CPU saturated conditions, we show that the scheduling responsiveness of cooperative polling is five times better than a well-tuned fair-share scheduler, and orders of magnitude better than the best-effort scheduler used in the mainstream Linux kernel.

All our source code is Open Source and is freely available for download from the repository, `http://dsg.cs.ubc.ca/viewsvn`. Details of the project, current status, related publications along with the checkout URLs of the codebase can be seen from our project homepage: `http://dsg.cs.ubc.ca/coopfsched/`. My research log can be viewed from: `http://cs.ubc.ca/~anirbans/research/` (access restricted to UBC Computer Science faculty and graduate students only). Complete set of results for all our implementations (including experimental ones and those which we discarded in the course of our work) is available for viewing from `http://dsg.cs.ubc.ca/~anirbans/`. Qstream is completely Open Source and can be freely downloaded from `http://www.Qstream.org`. Benchmark scripts used to perform all the experiments are available along with the Qstream source.

Bibliography

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

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

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

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

[5] M.J. Bach. The Design of Unix Operating System. Prentice Hall Inc.,New Jersey.

[6] A. Bavier and L. Peterson. The power of virtual time for multimedia scheduling. In *NOSSDAV 2000: Proceedings of the Tenth International Workshop for Network and Operating System Support for Digital Audio and Video*, pages 65–74, june 2000.

[7] Daniel Pierre Bovet and Marco Cassetti. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005.

[8] Aaron Brown et al. HBench-OS. `http://www.eecs.harvard.edu/vino/perf/hbench/index.html`.

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

[10] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 261–276, 1999.

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

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

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

[14] Thomas Gleixner and Douglas Niehaus. High resolution timer patches for linux kernels prior to 2.6.21. `http://www.tglx.de/projects/hrtimers/`.

[15] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Linux Symposium*, volume 1, pages 333–346, Ottawa, Canada, 2006.

[16] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on a commodity os. *SIGOPS Oper. Syst. Rev.*, 36(SI):165–180, 2002.

[17] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, October 1996.

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

[19] Con Kolivas. The Rotating Staircase Deadline Scheduler. `http://ck.kolivas.org/patches/staircase-deadline/`.

[20] Kon Kolivas. Pluggable cpu scheduler framework. http://lwn.net/Articles/109049/.

[21] Charles Krasic. *A Framework for Quality-Adaptive Media Streaming*. PhD thesis, OGI School of Science and Engineering at OHSU, October 2003.

[22] Charles Krasic et al. Qstream. `http://www.qstream.org/`.

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

[24] Charles Krasic, Jonathan Walpole, and Wu chi Feng. Quality-adaptive media streaming by priority drop. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 112–121, New York, NY, USA, 2003. ACM Press.

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

[26] 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.

[27] Ingo Molnar. Completely Fair Scheduler. `http://people.redhat.com/mingo/cfs-scheduler/`.

[28] Ingo Molnar. Pipe test. `http://people.redhat.com/mingo/cfs-scheduler/tools/pipe-test.c`.

[29] Ingo Molnar. Ring test. `http://people.redhat.com/mingo/scheduler-patches/ring-test.c`.

[30] Jason Nieh and Monica S. Lam. The design, implementation and evaluation of SMART: a scheduler for multimedia applications. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 184–197, 1997.

[31] Nvidia Corporation. NVIDIA Quadro Plex. `http://www.nvidia.com/page/quadroplex.html`.

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

[33] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton. Comparing the performance of web server architectures. In *EuroSys '07: Proceedings of the 2007 conference on EuroSys*, pages 231–243, New York, NY, USA, 2007. ACM Press.

[34] Darwyn R. Peachey, Richard B. Bunt, Carey L. Williamson, and Tim B. Brecht. An experimental investigation of scheduling strategies for unix. In *SIGMETRICS '84: Proceedings of the 1984 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 158–166, New York, NY, USA, 1984. ACM Press.

[35] Lennart Poettering, Pierre Ossman, Shahms E. King, et al. PulseAudio Sound Server. `http://0pointer.de/lennart/projects/pulseaudio/doxygen/`.

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

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

[38] Suresh Siddha, Venkatesh Pallipadi, and Arjan Van De Ven. Getting maximum mileage out of tickless. In *Linux Symposium*, volume 2, pages 201–207, Ottawa, Canada, 2007.

[39] Anirban Sinha, Charles Krasic, and Ashvin Goel. Achieving predictable timing and fairness through cooperative polling. In poster session, ACM Symposium on Operating Systems Principles (SOSP), October 2007.

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

[41] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 145–158, February 1999.

[42] John Stultz, Nishanth Aravamudan, and Darren Hart. We are not getting any younger: A new approach to timekeeping and timers. In *Linux Symposium*, volume 1, pages 219–232, Ottawa, Canada, 2005.

[43] Satoru Takeuchi. massive_intr. `http://people.redhat.com/mingo/cfs-scheduler/tools/massive_intr.c`.

[44] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for Internet services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 268–281, 2003.

[45] Jeremy Weatherford. orbitclock. `http://linux.1wt.eu/sched/`.

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

[47] Thomas Williams et al. Gnuplot. `http://www.gnuplot.info/`.

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

Appendix A

Finer Details of the Experimental Setup

A.1 Client Configuration

In all our experiments, we used the older Nvidia driver version 1.0-8776 uniformly for all runs. We had to apply a very simple patch in order to make the driver compatible with newer Linux 2.6.20 kernel. This was necessary because in the newer 9629 and above drivers with NVIDIA Quadro Plex[31] support, we observed that the Xserver was consuming unusually large percentage of CPU cycles. We do not know the exact reason for this behavior since there has been some major changes in between driver versions 8776 and 9629[7]. However, when we reverted back to the old legacy 8776 driver version, Xserver seemed to behave normally.

A.2 Kernel Configuration

In all our experiments, high resolution timers and hpet drivers were enabled during kernel configuration. However, vanilla Linux 2.6.20 kernel does not come with drivers that can provide high resolution timer support. We had to patch the kernel with high resolution patches [14] in order to enable high resolution timer support. These patches have been integrated into the mainstream 2.6.21 kernel. While enabling high resolution timers, we have been careful not to enable the timer monitoring and profiling options as these add significant additional overhead.

[7] details can be seen here:`http://www.nvidia.com/object/Linux_display_ia32_1.0-9629.html`

For all our experiments, we configure the jiffies to be of 1000 HZ.

Some additional fine tuning is performed on the top of this basic setup so that our benchmark results are not affected by other external factors. We make sure that we have enough memory on the client side so that running multiple players on the client does not cause undue memory pressure. All unnecessary daemons and applets running on the client are killed so that our client is free from interference from other stray applications when we run our benchmarks. In the scripts, we echo 0 to `/proc/sys/vm/swappiness` so as to ensure that the kernel does not swap Qstream pages to disk[8].

[8] We saw that having enough free physical memory at any given time does not prevent Linux kernel from swapping pages to disk. This, we believe, is a bad design choice in the Linux memory management subsystem.

Appendix B

Source Codes

B.1 The coop_poll System Call Code

```
/* sys_coop_poll - The coop_poll system call interface
 * @i_param: input deadline parameters
 * @o_param: output deadline parameters
 *
 * This version of the coop_poll does not force a process to
 * sleep on a completion variable/waitqueue. Instead, it just
 * calls schedule() to yield the
 * processor to another coop task.
 * This system call is thus, very similar to sched_yield() call
 * that also yields the processor for a very small amount of
 * time. In a combined fairshare & coop heuristic, this is
 * expected to result in a much more simpler code and is
 * conceptually more consistent with what we want it
 * to do.
 */

asmlinkage long
sys_coop_poll(struct coop_param_t __user *i_param,
              struct coop_param_t __user *o_param){

        struct coop_param_t ki_param;
        struct coop_param_t ko_param;

        int ret;
        unsigned short yields = 0;
        unsigned long flags;
        int flg;
```

```
struct timeval tv_now, tv_diff;
struct timespec tomono;
coop_queue *cq;
struct bvtqueue *bq;
struct task_struct *w_asap;
struct timeval tv_zero = {.tv_sec = 0,.tv_usec = 0};

ko_param.t_deadline = tv_zero;
ko_param.t_asap     = tv_zero;
ko_param.p_asap     = 0;
ko_param.have_asap  = 0;

/* copy input values to kernel space
 * this is where the kernel checks for
 * memory access violations
 * This might put the process to sleep
 */
ret = get_user_arg(&ki_param, i_param);
if (ret)
        return ret;

/* if a task has no asaps or deadlines to report,
 * we pretend as if this guy is only a besteffort guy
 */

if (unlikely(!GET_HAVE_ASAP(ki_param)) &&
    unlikely(timeval_compare(&ki_param.t_deadline,
                             &tv_zero) == 0)) {
        goto yield;
}

/* acquire the coop queue lock.
 * this disabled preemption and local
 * IRQs and acquires
 * the runqueue spinlock
 */
```

```
cq = task_cq_lock(current, &flags);

bq = cpu_bq(task_cpu(current));

/* update the number of coop count in coop queue */

cq->num_coop_calls++;

if ((current)->cf.bvt_t.bvt_timer_active) {
        bvt_timer_cancel(&bq->bvt_timer);
        current->cf.bvt_t.bvt_timer_active = 0;
}

if (!is_coop_realtime(current)) {
        set_tsk_as_coop(current);
}

current->cf.coop_t.is_well_behaved = 1;

do_gettimeofday(&tv_now);

/* remove my stale nodes from the coop heaps
 * and re-insert new
 * nodes based on my updated information
 */
remove_task_from_coop_queue(current, cq,0);

/* Insert my info into the heap */

if ( likely(timeval_compare(&ki_param.t_deadline,
                            &tv_zero) != 0) ) {
        ret = insert_task_into_timeout_queue(
                                &ki_param.t_deadline,
                                cq,
                                current);
        if (unlikely(ret < 0)){
                remove_task_from_coop_queue(current,
```

```
                                                    cq,0);
                cq_unlock(cq, &flags);
                return ret;
        }
} /* if */

flg = 0; /* will be = 1 only when there are asaps */

if ( likely(GET_HAVE_ASAP(ki_param))) {
        ret = insert_task_into_asap_queue(
                                &ki_param.t_asap,
                                ki_param.p_asap,
                                cq,
                                current);
        if (ret < 0) {
                remove_task_from_coop_queue(current,
                                                    cq,0);
                cq_unlock(cq, &flags);
                return ret;
        } else {
                flg = 1;
        }
} /* if */

/* if there are no asaps and the deadline is in the
 * future, make the process sleep in coop_poll() until
 * the deadline expires
 */
if (!flg &&
        timeval_compare(&ki_param.t_deadline,
                        &tv_now) > 0) {
        /* calculate the time difference */
        set_normalized_timeval(&tv_diff,
                ki_param.t_deadline.tv_sec
                - tv_now.tv_sec,
                ki_param.t_deadline.tv_usec
                - tv_now.tv_usec);
```

```
                cooperative_highres_sleeper(cq,
                                        &flags,
                                        timeval_to_ktime(tv_diff));
                goto wakeup;
        }

        cq_unlock(cq, &flags);
yield:
        schedule(); /* this is how we now yield */
wakeup:
        yields++;

        cq = task_cq_lock(current, &flags);

        /* update the current running task node */

        cq->num_yields += yields;

        w_asap = NULL;
        flg = 0;

        /* remove, obtain most imp asap and then reinsert */
        if (current->cf.coop_t.coop_asap_heap_node) {
                remove_task_from_coop_queue(current,cq,2);
                flg = 1;
        }

        find_nearest_asap(cq, &w_asap);

        if (flg)
                __insert_into_asap_heap(cq,current);

        if (w_asap) {
                ko_param.t_asap =
                w_asap->cf.coop_t.asap_p.t_asap;
                ko_param.p_asap =
```

```
                w_asap->cf.coop_t.asap_p.priority;
                SET_HAVE_ASAP(ko_param);
        }

        cq_unlock(cq, &flags);

        /* adjust for wall time, monotonic time difference */
        tomono = wall_to_monotonic;
        set_normalized_timespec(&current->cf.coop_t.deadline,
                        current->cf.coop_t.deadline.tv_sec
                        - tomono.tv_sec,
                        current->cf.coop_t.deadline.tv_nsec
                        - tomono.tv_nsec);

        ko_param.t_deadline.tv_sec  =
        current->cf.coop_t.deadline.tv_sec;
        ko_param.t_deadline.tv_usec =
        current->cf.coop_t.deadline.tv_nsec / NSEC_PER_USEC;

        /* reset the well behaved flag */
        current->cf.coop_t.is_well_behaved = 0;

        /* send it to user process */
        return copy_to_user(o_param,
                    &ko_param,
                    sizeof(struct coop_param_t))? -EFAULT:0;
}
/* sys_coop_poll */
```

B.2 The cooperative_sleeper() Code

```
/* cooperative_highres_sleeper:
 * arms a high resolution timer to sleep for the
 * requested interval of time in ktime_t
 * Also releases the corresponding runqueue lock.
 * @cs: The pointer to coop queue
 * @flags: The IRQ flags saved previously.
 * @time: The amount of time to sleep in ktime_t.
 */
static ktime_t cooperative_highres_sleeper(coop_queue *cq,
                                           unsigned long *flags,
                                           ktime_t time)
{
        struct hrtimer_sleeper t;
        ktime_t remain;

        hrtimer_init(&t.timer, CLOCK_MONOTONIC,
                     HRTIMER_MODE_REL);
        hrtimer_init_sleeper(&t, current);

        set_current_state(TASK_UNINTERRUPTIBLE);
        hrtimer_start(&t.timer, time, HRTIMER_MODE_REL);

        cq_unlock(cq, flags);

        if (likely(t.task))
                schedule();

        hrtimer_cancel(&t.timer);

        remain = hrtimer_get_remaining(&t.timer);

        if (ktime_to_ns(remain) < 0)
                return ktime_set(0, 0);
        else
                return remain;
```

```
}
```

B.3 The Borrowing Code

```
/* bvt_borrow: implement borrowing of virtual times.
 * It is called from activate_task in sched.c
 */
void bvt_borrow(struct task_struct *p, struct bvtqueue *bq)
{
        struct fairshare_sched_param *top_node = NULL;
        struct timeval tv_zero = { .tv_sec = 0,
                                   .tv_usec = 0};

        p->cf.bvt_dom->num_tasks++;

       /* handle real coop wakeups seperately here */

        if (is_coop_realtime(p)) {
                /* reinsert the coop nodes
                 * into the coop heap */
                if (timeval_compare(
                    &(p->cf.coop_t.dead_p.t_deadline),
                    &tv_zero) > 0) {
                        /* insert into timeout heap */
                        insert_into_coop_heaps(&bq->cq,
                        p,COOP_TIMEOUT_HEAP);
                        /* and remove from coop sleep
                         * queue */
                        remove_task_from_coop_sleep_queue(
                        p, &bq->cq);
                }
                /* if there are other coops running,
                 * do not borrow and do not
                 * insert my nodes into the heap
                 */
                if (p->cf.bvt_dom->num_tasks >1) return;
        } /* if */
```

```
        if (likely(!heap_is_empty(bq->bvt_heap))) {
                top_node =
                (struct fairshare_sched_param*)
                heap_min_data(bq->bvt_heap);
        }

        if (top_node) {
                p->cf.task_sched_param->bvt_virtual_time =
                top_node->bvt_virtual_time;
        }
        else{
                set_normalized_timespec(
                &(p->cf.task_sched_param->bvt_virtual_time),
                0, 0);
        }

        /* insert task into the bvt queue
         */
        if (likely(!p->cf.task_sched_param->bheap_ptr)) {
                insert_task_into_bvt_queue(bq,p,HEAP_NO_GROW);
        }
}
```

B.4 The Policing Code

```
/* do_policing: This is the main policing function.
 * Important: It only goes one way, i.e., demotes
 * a real coop task to a best effort
 * task.The assertion at the
 * very beginning of the function
 * ensures that this is not violated.
 */
void do_policing(struct bvtqueue *bq, struct task_struct *tsk)
{
        g_assert(tsk);
        g_assert(bq);
```

```
        g_assert(is_coop_realtime(tsk));

        /* We need to set the private virtual time of the
         * policed task to the coop-domain's virtual time.
         * This will ensure that when the task has correct
         * virtual time once the scheduler charges its
         * running time to itself.
         */
        tsk->cf.bvt_t.private_sched_param.bvt_virtual_time =
                tsk->cf.task_sched_param->bvt_virtual_time;

        /* register this task as belonging to the
         * best effort domain. Once this is done, the
         * scheduler routine will correctly charge the running
         * time of this task to the task itself and not to the
         * coop-domain.
         */
        tsk->cf.bvt_dom = &(bq->bvt_domains[DOM_BEST_EFFORT]);

        /* task virtual time is going
         * to be the virtual time of the
         * individual task
         */
        tsk->cf.task_sched_param =
                        &tsk->cf.bvt_t.private_sched_param;

        clear_coop_task(tsk);

} /* do_policing */
```

B.5 The Modified Kernel `deactivate_task()` Function

```
 /*
 * deactivate_task - remove a task from the runqueue.
 */
static void deactivate_task(struct task_struct *p,
```

```
                                struct rq *rq)
{
#if defined (CONFIG_SCHED_COOPREALTIME)
        struct bvtqueue *bq = &(rq->bq);
#endif
        dec_nr_running(p, rq);
        dequeue_task(p, p->array);
        p->array = NULL;

#if defined (CONFIG_SCHED_COOPREALTIME)

        if(!is_bvt(p)) return;

        bq->running_bvt_task = NULL;

        bvt_timer_cancel(&bq->bvt_timer);
        p->cf.bvt_t.bvt_timer_active = 0;

        if (is_coop_realtime(p))
        {
                /* note that policing has
                 * not yet taken place, so
                 * no matter whether we are
                 * cooperatively sleeping or
                 * not, we remove our nodes
                 * from coop heap because
                 * the task is getting
                 * deactivated.
                 */
                test_remove_task_from_coop_bvt_queues(
                                p,&bq->cq);

                if(!p->cf.coop_t.is_well_behaved &&
                        !p->exit_state) {
                        do_policing(bq,p);
                } /* if */
                else {
```

```
                        insert_into_coop_heaps(&bq->cq,
                                p,COOP_SLEEP_HEAP);
                } /* else */
        } else  {
                remove_task_from_bvt_queue(bq,p);
                p->cf.bvt_dom->num_tasks--;
        } /* else */
#endif
}
```