Fair and Timely Scheduling via Cooperative Polling

Charles 'Buck' Krasic[1] Mayukh Saubhasik[1]
Anirban Sinha[1] Ashvin Goel[2]

[1]Department of Computer Science
University of British Columbia

[2]Department of Electrical and Computer Engineering
University of Toronto

Outline

Outline

1. Introduction
   - Problem Description
   - Previous Approaches
2. Our Approach
   - Design
   - Implementation
3. Results
   - Timeliness
   - Fairness

Introduction

- Scheduling in commodity operating systems traditionally favors throughput over timeliness
  - Time sensitive applications are poorly served unless they have low requirements.
- Our approach improves timeliness while preserving benefits of the best effort model
  - Application model for time sensitive applications
  - Kernel scheduler the provides fairness and timeliness
  - New system call called coop_poll that supports cooperation between application and kernel level schedulers
- *Timing improvements of up to two orders of magnitude*

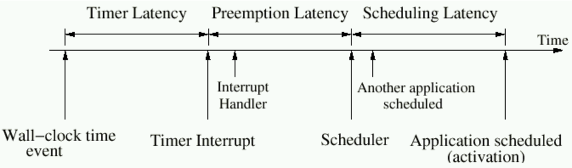
Time Sensitive Applications

- Hard real-time
  - aircraft controllers, airbag controllers
- Soft real-time
  - games, graphical animation (visualizations, desktops, etc.)
  - continuous media (audio and video)
  - distributed computing services (e.g. SLAs)
  - user level drivers

Elements of good scheduling

- Throughout
  - work conserving
  - low overhead
- Fairness
  - Max-min fairness is common in best effort systems.
  - Can be resource centric (QoS: CPU time, bandwidth, etc.) or application centric (QoE: PSNR, MOS, etc.)
- Timeliness
  - Release-Time, Deadline, Jitter
  - *Tardiness:* difference between release time and corresponding activation.

Critical Path of Tardiness



- Timer Latency
  - High resolution clock, timers.
- Preemption Latency
  - Fully preemptable kernel.
- Scheduling Latency
  - Our approach.

Outline

1. Introduction
   - Problem Description
   - Previous Approaches
2. Our Approach
   - Design
   - Implementation
3. Results
   - Timeliness
   - Fairness

Classic Real-time

- Priority Based.
  - Starvation, inversions.
- Reservation based.
  - Very hard to estimate resource requirements.
- Tune the reservation parameter via Feedback.
  - Can lead to instability for adaptive applications.
  - Composing feedback controllers is hard.

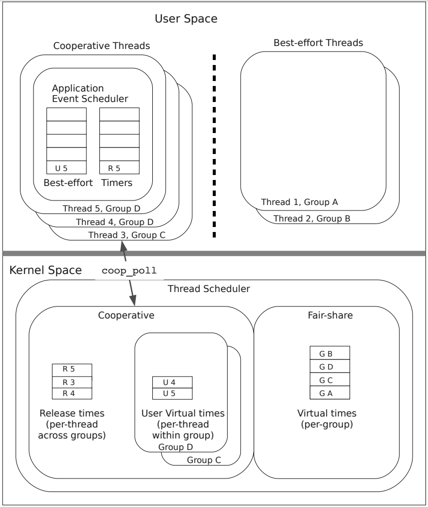
Outline

1. Introduction
   - Problem Description
   - Previous Approaches
2. Our Approach
   - Design
   - Implementation
3. Results
   - Timeliness
   - Fairness

Key Ideas

- Time-sensitive applications can cooperate with kernel and each other
    - Applications include a user level scheduler
- Inform kernel of timing needs
    - new system call: `coop_poll()`
- Kernel facilitates and coordinates this information exchange
- Kernel offers protection against mis-behaving applications

Architecture



User Level Programming Model

- Reactive event loop
  - Two types of events - Best Effort, Timer
  - Short running events
    - stack-rip loops or use coroutines
  - Use non-blocking I/O as much as possible.
- Adaptive applications
  - reduce events (best-effort) during *overload*

Application Example

```
recv_video_frame(player, frame) {
  frame.decode_event = {
    type = BEST_EFFORT,
    user_virtual_time = decoder_get_virtual_time(frame),
    callback.fn = decode_video_frame };
  submit(frame.decode_event)
  frame.expire_event= {
     type = TIMER,
     release = decoder_get_release_time(frame),
     callback.fn = expire_video_frame };
  submit(frame.expire_event);
}
```

Application Example (cont'd)

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

Kernel Fairshare Scheduler

- Weighted fairshare scheduler.
- Virtual time:
  - Use high-resolution accounting to measure execution time.
  - Vitual time = weight $\times$ measured.
  - Not allowed to accumulate virtual time by sleeping.
- Task with lowest virtual time picked for execution.
- Timeslice $=$ Period $/$ Number of runnable tasks.
  - lower bound enforced to prevent excessive context switches

Coop_Poll Call

- Coop_Poll connects user level scheduler to kernel scheduler.
  - Input $\leftarrow$ Earliest local release-time & user virtual time.
  - Output $\rightarrow$ CPU-wide earliest release-time & group-wide earliest virtual time.

Coop_Poll in the kernel scheduler

- Timeslice calculation (amended)
  - Timeslice = min(Period/N, Time till next release-time)
  - Sets output param of coop_poll.
- fairness vs timeliness?
  - If release-time is due override fairness choice, *but* force task to yield quickly: set output release-time = now.
  - Allows temporary unfairness, subject to following limit.

Mis-behaving Applications

- Un-cooperative behavior:
  - Does not yield on time (now $-$ release time $>$ coop slack).
  - Non-cooperative yield (page fault, IO, sleep, i.e. not coop_poll).
  - Exceeds unfairness threshold (Task VT $-$ Min VT $>$ Unfairness Threshold).
- Kernel demotes task to best-effort status
  - Temporary effect, status regained next time coop_poll is called.

Coop_Poll in the user level event scheduler

- The event loop yields the CPU *only* via coop_poll.
- The output parameters of coop_poll are translated into proxy events:
  - Two proxies: timer and best-effort.
  - Proxy events yield the CPU via coop_poll.
- Application defined fairness via thread groups.
  - Whole group shares the same virtual time.
  - User defined fairness within group.
    - Best-effort events contain application specified virtual time.
    - e.g. cumulative fps, cumulative utility

Outline

1. Introduction
   - Problem Description
   - Previous Approaches
2. Our Approach
   - Design
   - Implementation
3. Results
   - Timeliness
   - Fairness

Scheduler Implementation

- Implemented scheduler and coop poll in Linux kernel
  - Two versions: pre and post CFS (2.6.22 and 2.6.25).
  - Simple tickless design:
    - one shot high resolution timers.
    - High resolution accounting.
  - Previously prototyped approach at user level.

Cooperative Applications

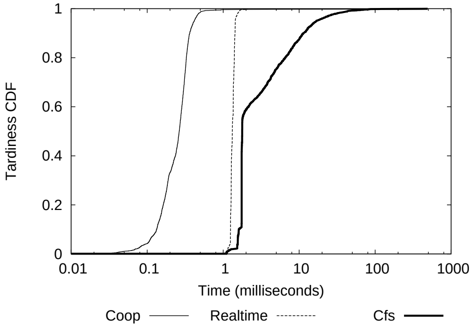
- QStream Adaptive Video Streaming
  - ideal candidate - event based, adaptive, short running events.
- X11 display server.
  - Event based, but non-adaptive.
  - Extend Xsync to support high res timers
  - Incorporate integrated Coop_Poll
  - 0.3% LOC changed (Excluding extensions).

Outline

Experimental Setup

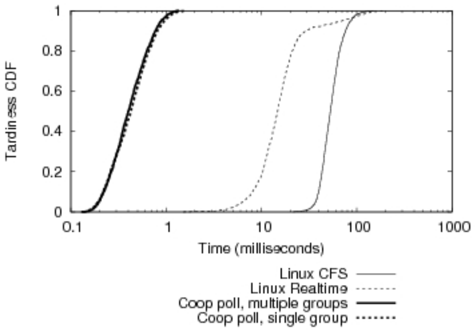
- Pentium 4 3.0 Ghz, Nvidia NV43 GPU
- Kernel based on Linux 2.6.25, Preemption, High-Resolution Timers, SMP enabled
- 20 ms global period, 20 ms unfairness threshold, 100 us min timeslice, 2 ms coop_slack
- Compare with CFS and Linux real-time.

Baseline Timeliness



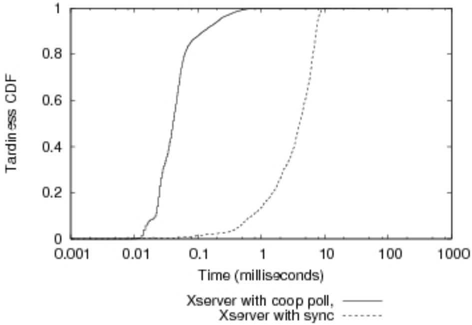
- Single time-sensitive thread + 4 Background loads.
- Gives reference point for application granularity and best-case timeliness.

Timeliness with Multiple Adaptive Applications



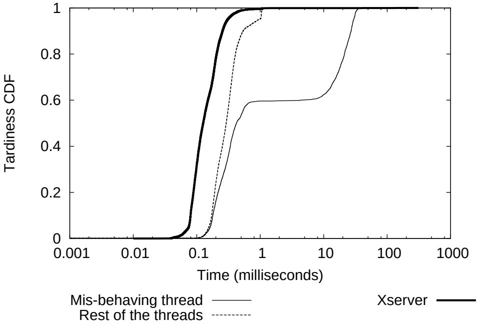
- 8 time-sensitive threads + 4 Background loads.
- Linux real-time priority doesn't help here.

Non-Adaptive Application(X11 Display Server)



- 8 Video Players(X11 enabled) + 4 Background loads.
- Player performance (not shown) similar to previous case.

Mis-behaving Application



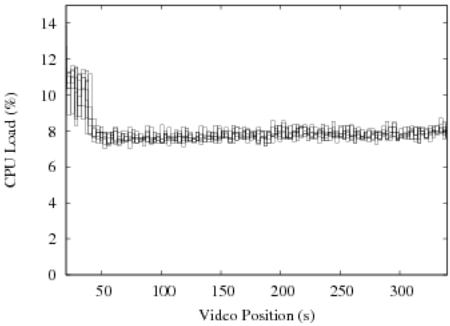
- Same workload as before, 1 player delaying yields with probability of 1%
  - Random delay ranging from 0 . . . 10 milliseconds.
- Misbehaving task is the only one to suffer!

Limits of frequent context switching

- Q: Why don't we just use a high granularity periodic scheduler?
- e.g. Global period $=$ 1 millisecond.
- Coop tardiness is 5x better with 4x fewer context switches.
  - Tardiness is still 5ms due to context switch plateau.
  - 9348 Context Switches/Second *vs* 2211 Context Switches/Second.

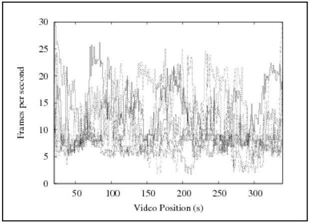
Outline

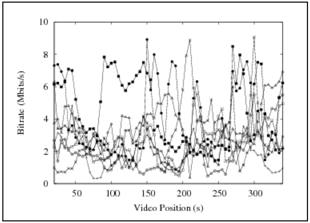
Fairness for Adaptive Applications



- 8 Video players + 4 Background loads.
- Coop fairshare scheduler shown, results with Linux CFS (not shown) are similar.
- Coop achieves timeliness *and* fairness.

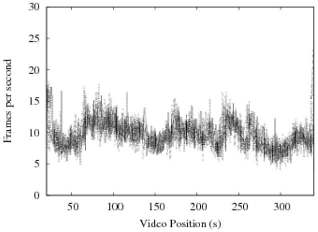
CPU fairness != Quality Fairness

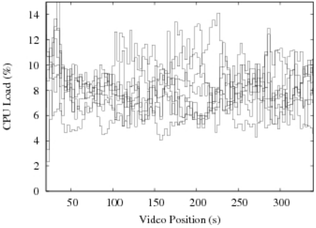




- FPS per video (left graph) is chaotic.
- Video Bitrates (right graph) are indicative of videos' busty requirements.

Application-centric fairness





- Equal user defined quality (fps) via user scheduling
- All players run in same Coop group
  - CPU within group allocated according to user specified virtual time.

Summary

- Cooperative Polling is:
  - Split level user-kernel scheduling
  - Kernel combines fair sharing base with timeliness through cooperative-polling
  - Kernel facilitates cooperation and protects against misbehaviour.
  - Supports resource and application centric fairness
- Results indicate sub-millisecond timing requirements are attainable.
- *Reconciles the conflict between best-effort and time-sensitive applications.*

Future Work

- Multiprocessor evaluation
- Integrate with thread library such as GNU Pth
- Other resource types—storage, network, memory.
- Implementing the concept in a hypervisor.
- Move of fast-path `coop_poll` to Linux *vsyscall*.
- Support for Linux scheduler groups/cgroups.

Questions?

- All of our code is open source: `http://qstream.org/`
- Please visit our poster/demo at the poster session.