Achieving Predictable Timing and Fairness Through Cooperative Polling

Anirban (Ani) Sinha

Department of Computer Science
University of British Columbia

November 2, 2007

Outline

The Changing Computing Environment

- Multimedia capable (soft) realtime applications are increasingly becoming common.
- Many of these applications are *adaptive*:
  - They consume as much CPU resources as are available.
  - Adaptive tasks keeps the system overloaded at all times (when adaptation is active).
- They are *peculiar*:
  - They are *time-sensitive*:
    - Have specific deadlines for doing certain jobs.
  - They are often *both* IO intensive with considerable CPU requirements.
- Other kinds of mixed workloads (e.g., security enabled web-servers, databases) are also common.

The New Challenges

Challenges for a task scheduler therefore are as follows:

- Provide a good balance of overall throughput and timeliness.
- Uphold work conservation
  - Maximum utilization of available CPU resources.
- Allow graceful & coordinated adaptation.
- Avoid starvation.
- Use an effective strategy for load balancing in SMP environments.

We do not address the last issue in this work.

The space of multimedia scheduling for general purpose OS is well explored:

Related Works

- SMART: Jason Nieh and Monica S. Lam. The design, implementation and evaluation of SMART: a scheduler for multimedia applications. SOSP 1997.
- BVT: Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. SOSP 1999.
- BEST: Scott A. Banachowski, Scott A. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. MMCN 2002.

The drawbacks

- Use of complicated schedulability analysis and/or kernel-userspace interaction mechanism.
- Use of some notion of priorities that can lead to starvation.
- Dependence on CPU reservations (or assumption of underloaded system) for providing better timing.
  - Throw away work conservation.

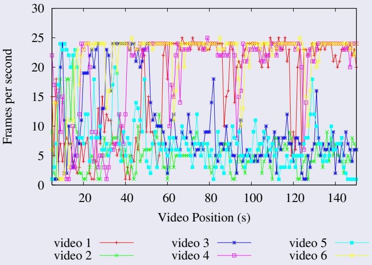
The $O(1)$ Scheduler Overview

- Uses multi-level feedback queue scheduling algorithm.
- Uses static (nice levels) and dynamic priorities.
    - Affected by starvation and live locks.
- Is not particularly effective for mixed IO and CPU bound workloads.
- Has rather large timeslices for high priority IO bound jobs (800 ms).
- Uninformed preemptions leads to poor adaptations.
    - No mechanism to achieve coordinated adaptation for adaptive workloads.

We do not discuss the new 2.6.23 CFS scheduler in this work.

The $O(1)$ Scheduler Performance

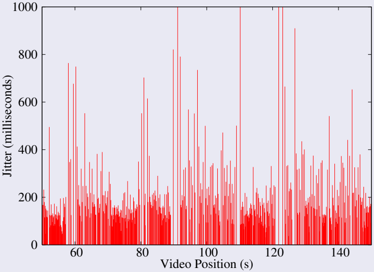
Six VLC players playing one video each on Vanilla 2.6.20 Kernel



Frame Rate

The $O(1)$ Scheduler Performance

Six VLC players playing one video each on Vanilla 2.6.20 Kernel



Frame Jitter

Outline

Our Scheduler Design Objectives

Our scheduler tries to satisfy the following objectives:

- Have overall long term fairness in the system.
- Have predictable timeliness (within the bounds of fairness) even in overload.
- Allow time sensitive applications to cooperate.
  - Cooperation helps to achieve coordinated adaptation.
  - Cooperation provides better timeliness.
- Uncooperative, misbehaving cooperative tasks should be policed.
- Achieve a good balance of throughput and responsiveness.

Outline

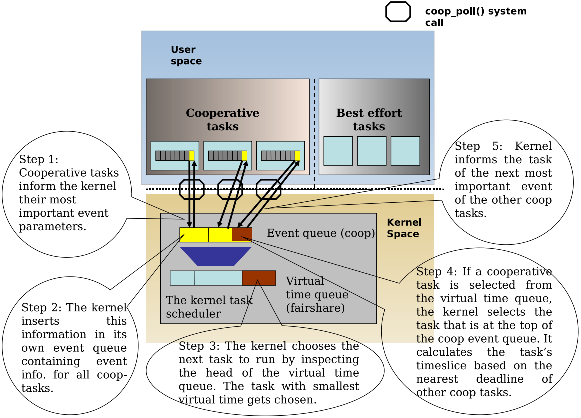
Design Highlights

- Fairshare scheduler based on virtual time to schedule all tasks.
  - Ensures long term fairness.
  - *Borrowing* prevents accumulation of virtual time.
- All time sensitive tasks form a cooperation group.
  - A common virtual time for the whole group.
  - No fairsharing or allocation enforcement within the group.
  - Tasks in the cooperation group cooperate with one another through kernel using `coop_poll` primitive.
  - Tasks within cooperation group are scheduled based on their deadlines and best effort priorities.
- Preferential treatment and policing of cooperative tasks by fairshare scheduler.

The coop_poll() Primitive

- coop_poll(IN,OUT)
  - IN: Most important deadline and best effort priority event of the current task.
  - OUT:
    - Most important deadline of all the external time sensitive tasks *or* the fairshare policing deadline, whichever is earlier.
    - Best effort event of all the external time sensitive tasks.
- Kernel Responsibility:
  - Resume the task when:
    - IN parameter deadline has expired (preferential treatment) or
    - IN parameter best effort is most important.
- Task Responsibility:
  - Treat the OUT events as it's own
  - yield back to kernel using coop_poll when they fire.

Scheduling Overview



The Scheduling Overview

The Main Kernel Scheduler

Algorithm: schedule()

```
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);
     }
}
```

Choosing the Next Task

Algorithm: choose_next_task()

```
choose_next_task() {
   nextTask = q_head(Wfq);
   if (nextTask.sched_dom == COOP_DOM) {
          nextTask = choose_next_coop_task();
   }
   return nextTask;
}
```

Choosing the Next Time Sensitive Task

Algorithm: choose_next_coop_task()

```
choose_next_coop_task() {
    if (head_expired(CoopDomain.dead_ev)){
        nextDeadEv = q_head(CoopDomain.dead_ev);
        return task(nextDeadEv);
    }else if(q_not_empty(CoopDomain.be_ev)){
        nextBeEvent =
        q_head(CoopDomain.be_ev);
        return task(nextBeEvent);
    } else {
        return ERR;
    }
}
```

Calculating Timeslice

Algorithm: cal_Tslice()

```
cal_Tslice(nextTask, &Tslice) {
  fsPrd = find_fs_prd();
  coopPrd  = earliestCoopDead - now;
  if (coopPrd < 0) coopPrd = 0;
  nextDeadTask = find_earliest_deadline_task();
  if (nextTask.virtual_time + coopPrd <
      nextDeadTask.virtual_time) {
         timeDelta = nextDeadTask.virtual_time
                   - (nextTask.virtual_time
                   + coopPrd);
         coopPrd = coopPrd + timeDelta;
   }
   Tslice = max(min(fsPrd,coopPrd),minTslice);
}
```

Implementation Overview

- Implementation on 2.6.20 kernel + highres timers
  - High resolution timers for timeslice enforcement.
  - Use of fine grained time accounting in the kernel.
- Binary heaps for our runqueue.
  - Tasks sorted based on their virtual time.
  - Also two heaps for sorting the time sensitive tasks based on deadlines and best effort priorities.
  - Heaps implemented with existing kernel runqueue - no separate locking mechanism needed.
- A new system call `coop_poll()`.
- We override the vanilla kernel scheduling decision with ours.

Outline

Evaluation Strategy

- Use a broad spectrum of load conditions: underloaded to fully overloaded.
- Vary the # of Qstream applications for varying the load.
  - 6 players => CPU just saturated. 12 players => complete saturation.
  - Qstream is a mixed CPU and IO intensive workload.
  - The challenge => achieve coordinated adaptations with graceful degradation.
- Qstream server run on a different machine.
  - Server load has no impact on the client performance.
- Enough memory & network bandwidth to handle 12 players - no memory pressure.
- Stray applets and services on client disabled.

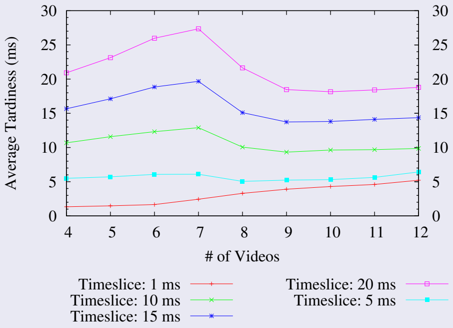
Evaluation of Fairshare Scheduling

Salient Points of the Experiment

- Qstream applications run as best effort task under the fairshare scheduler.
- No cooperation between applications.
- Frame display disabled.
  - Xserver has coarse grained event dispatch mechanism - perturbs our results.
  - Effects of Xserver eliminated.

Results

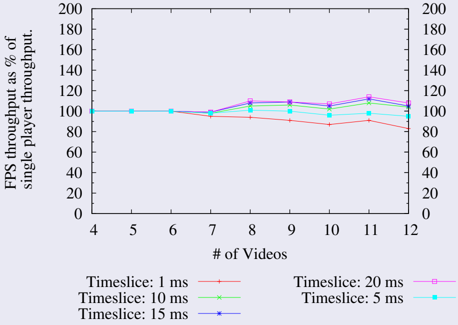
Dispatcher Latency



Dispatcher Latency

Results

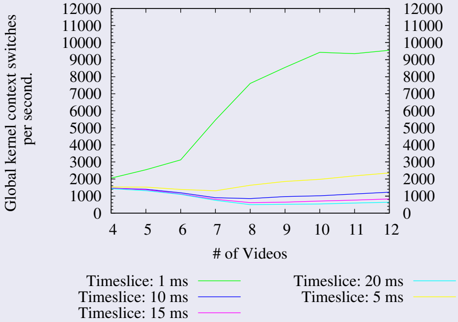
Throughput vs Monolithic (single player case)



Relative Throughputs

Results

Context Switch Rate



Context Switches

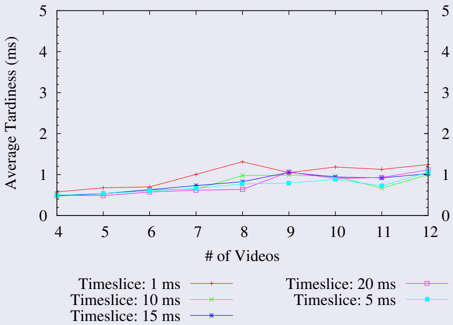
Evaluation of Cooperative Polling Algorithm with Policing

Salient Points of the Experiment

- Qstream applications cooperate with each other through kernel using `coop_poll()` system call.
- Its a homogeneous environment - all of the applications are well behaved.
- Frame display disabled.
  - Effects of Xserver eliminated.

Results

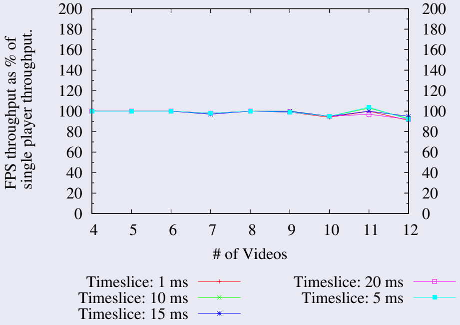
Dispatcher Latency



Dispatcher Latency

Results

Throughput vs Monolithic (single player case)

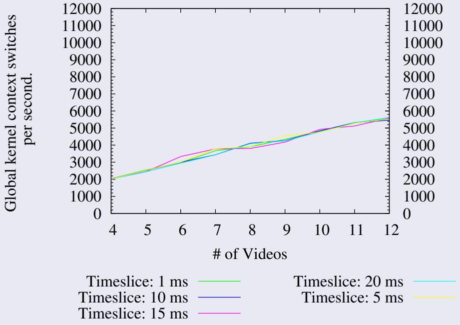


Relative Throughputs

Cooperative Polling (+ policing) Evaluation

Results

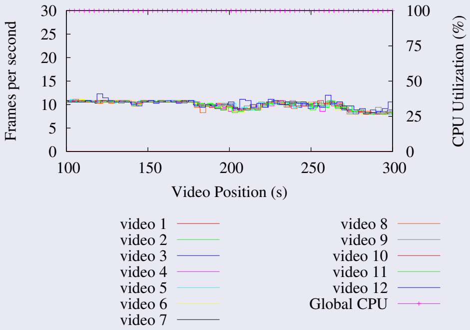
Context Switch Rate



Context Switches

Results

Frame Rates



Frame Rate for 12 videos with Xserver run as best effort task.

Cooperative Scheduling is Better Than Pure Fairsharing

Results with 10 players

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

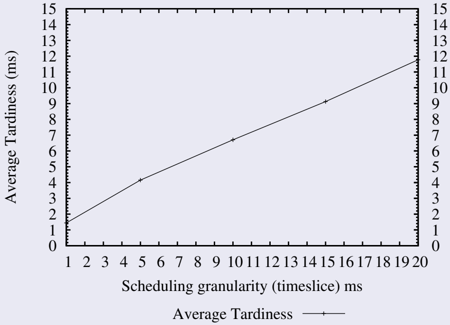
Performance Evaluation with High Definition Video

Salient Points of the Experiment

- A single Qstream player playing a 1080p high definition video, 679.2 kbyte/s bit-rate and 25 FPS.
  - The single video alone can take 70% of CPU.
- A best effort video encoding job run in parallel to completely saturate the CPU.
- Represents a common scenario where users watching a high definition video perform some video/audio encoding work in parallel.
- Xserver run as a best effort task in our fairshare scheduler.
  - Scheduled according to vanilla heuristics on the vanilla kernel.

Results

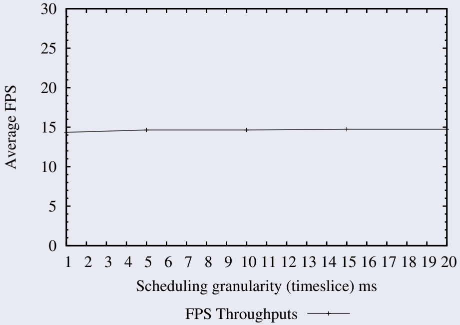
Dispatcher Latency as a function of global scheduling period



Dispatcher Latency

Results

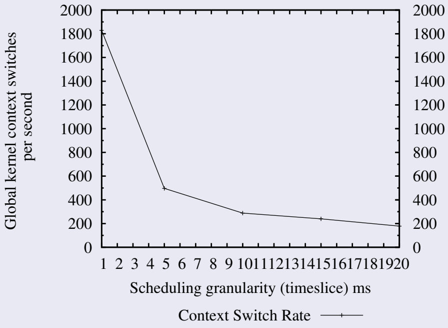
FPS throughput as a function of scheduling period



FPS Throughput

Results

Context switch rate as a function of scheduling period



Context Switch Rate

Outline

Summary

- Fairshare scheduling alone provides a baseline performance proportional to global period.
- The cost of smaller period and finer grained scheduling is high context switch overhead.
- Cooperative polling can provide improved timeliness with reduced context switch overheads.
  - Informed context switches are less expensive.
  - Helps to achieve coordinated adaptation.
- Policing through fairsharing ensures long term fairness in the system with no starvation.

Project Resources

Everything is Open Source!

- Project URL: `http://dsg.cs.ubc.ca/coopfsched`
  - Contains project updates, publications and code repository checkout URLs.
- Qstream source: `http://Qstream.org`
  - Has all the benchmark scripts.

Acknowledgements

I sincerely thank the following people who have helped me to proceed in this work:

The Indispensable

- Dr. Charles 'Buck' Krasic, my supervisor.
- Dr. Ashvin Goel, my *unofficial* co-supervisor.
- Dr. Norman Hutchinson, my second reader.
- All anonymous reviewers of our papers and those who gave us valuable feedback at SOSP.

Acknowledgements

The Support

- The DSG lab and all the wonderful people therein - special mention to Geoff, Brendan, Andy.
- My wonderful labmates: Andrei, Gang, Mike Wood, Mike Tsai, Gitika, Mayukh, Cuong, Camilo and Brad.
- My three good friends: Kan, Meghan and Abhishek.
- All the CSGSA folks.

The Hidden Support

My family who has always been there for me, unconditionally.

Questions

Questions ...