Supporting Time-Sensitive Applications on a Commodity OS

A. Goel, L. Abeni, C. Krasic, J. Snow and J. Walpole

Proceedings of USENIX 5th Symposium on Operating System Design and Implementation (OSDI)

December 2002

Introduction

- Multimedia applications time-sensitive
  - Ex: periodic execution with low jitter (e.g. soft modem)
  - Ex: quick response to external event (e.g. frame capture in videoconference)
- OS must allocate resources at appropriate times
- Needs:
  - High precision timing facility
  - Well-designed preemptible kernel
  - Appropriate scheduling
- Most commodity OSes don't (Windows, Linux)
- Special OS enhancements can support real-time
  - But *hard* real-time, s.t. degradation of non-real-time applications suffer

Approach

1) *Firm timers* for efficient, high-resolution timing
2) *Fine-grained kernel preemptibility*
3) *Priority* and *Reservation–based CPU scheduling*

- Integrate into Linux kernel
  - → Time-sensitive Linux
- Show benefits real-time application, but not degrade performance of other apps

Outline

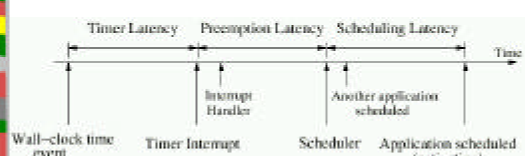
- Introduction (done)
- Related Work (next)
- Requirements
- Implementation
- Evaluation
- Conclusions

Related Work

- Illustration of real-time implementation difficulties [6,15,16]
- Mathematical real-time scheduling [10,19]
  - But ignore practical issues such as non-preemptibility
- Practical real-time scheduling [12,17,22]
  - But performance of non-real-time suffers
- Real-time micro-kernelish [4]
  - But hard-timers add more overhead
- New OSes [9]
  - But different API so hard to port apps

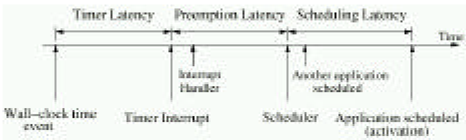
Time-Sensitive Requirements

- From time need to handle event until actual dispatch is *kernel latency*



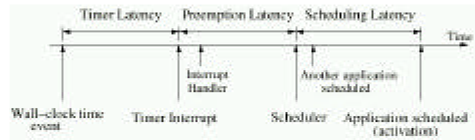
- Need: *Timing Mechanism*, *Responsive Kernel*, *CPU Scheduling Algorithm*

Timer Mechanism



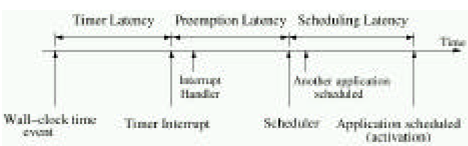
- Accurate timer the largest add to kernel latency
- Can use:
  - *One-shot timer* – on x86, use on-chip CPU Advanced Programmable Interrupt Controller (APIC). Needs to be reprogrammed each time.
  - *Soft Timer* – check for expired timers at strategic locations, reduce the number of interrupts
- Solution: Combine to call *firm timers*

Responsive Kernel



- If timer is accurate, might still not have low kernel latency if kernel cannot respond
  - (Traditionally, thread in kernel runs until done)
- Solution: reduce size of non-preemptible regions

CPU Scheduling Algorithm



- Need to schedule the right process as quickly as possible
- Solutions:
  - *Priority-based scheduler* – pre-assign priorities and schedule in that order
  - *Proportion-period scheduler* – schedule with an upper-bound on delay

Misc

- Note, any one alone *not* sufficient!
  - High-resolution timer doesn't help if kernel not preemptible or:
  - Responsive kernel not useful without accurate time
- Note, tasks may *not* be independent:
  - X server operates (and is scheduled) in FIFO order
  - Video application with higher priority than X server will have *priority inversion* (waiting on low priority) (will address)

Outline

- Introduction (done)
- Related Work (done)
- Requirements (done)
- Implementation
  - Firm Timers (next)
  - Fine-Grained Preemptibility
  - CPU Scheduling
- Evaluation
- Conclusions

Periodic Timers

- Commodity OSes implement timing with *periodic* timers.
  - Ex: on Intel x86, interrupts generated with Programmable Interval Timer (PIT)
  - Ex: is 10 ms on Linux, thus is max latency
- Can reduce latency by reducing period, but adds more interrupt overhead
- Instead, move to *one-shot* timer
- Ex: two tasks, period 5 and 7 ms, timer period 1 ms, 35 ms running time
  - Periodic: 35 interrupts generated
  - One-shot: 11 interrupts generated (5, 7, 10, 14 ...)
  - Plus, one-shot timer reduces timer latency

Firm Timer Design

- One-shot timer costs: timer reprogramming and fielding timer interrupts
  - Reprogramming cost has decreased in modern hardware (P2+)
    - PIT on x86 used to use slow out on bus
    - Newer APIC resides on CPU chip
  - Thus, last cost is interrupt cost
- Reduce by soft-timers
  - Poll for expired timers at strategic points where context switch is occurring
    - Ex: system call, interrupt, exception return
- Two new problems: poll cost and added timer latency
- Can solve 2nd problem with *timer overshoot*
  - Provides upper bound on latency
  - Tradeoff between accuracy and overhead
    - 0 → hard timers, large → soft-timers
    - At 100 MHz, theoretical accuracy of 10 nanoseconds

Firm Timer Implementation

- Timer queue for each queue, sorted by expiry
- When timer expires
  - execute callback function for each expired timer
  - Reprogram APIC
- Global overshoot value (but could be done per timer)
- Accessible through: `nanosleep()`, `pause()`, `setitimer()`, `select()` and `poll()`

Outline

- Introduction (done)
- Related Work (done)
- Requirements (done)
- Implementation
  - Firm Timers (done)
  - Fine-Grained Preemptibility (next)
  - CPU Scheduling
- Evaluation
- Conclusions

Reasons Scheduler Cannot Run

- Interrupts disabled
  - Hopefully, short
- Another thread in critical region
- Commodity OSes have no preemption for entire kernel period
  - Ex: when interrupt fires or duration of system call
  - Unless known it will be long (ex: disk I/O)
  - Preemption latency under Linux can be 30 ms

Enabling More Preemption

1) Add more preemption points
   - Must be done manually
2) Allow preemption anytime not using shared data structures
   - Protect shared structures with locks
   - Can still result in long latencies

- Combine 1) and 2) works best
  - (Done by Robert Love [11])
  - (Authors evaluted in [1])

Outline

- Introduction (done)
- Related Work (done)
- Requirements (done)
- Implementation
  - Firm Timers (done)
  - Fine-Grained Preemptibility (done)
  - CPU Scheduling (next)
- Evaluation
- Conclusions

CPU Scheduling

- Priority CPU scheduling is simple, POSIX compliant
  - But assumes applications well-behaved
- So, combine with proportion-period on top to give protection

Proportion-Period CPU Scheduling

- For single independent tasks, assign highest priority task
  - Mis-behaving task can consume "too much"
  - Use *temporal protection*
- *Proportion-period* provides by allocating fixed CPU amount each period
  - Task executes as "real-time" (highest priority) for time *Q* every *T*
  - Period determined by application requirements (Ex: 30ms for video)
- Implemented using Earliest Deadline First (EDF)

Priority CPU Scheduling

- *Priority inversion* occurs when an application has multiple tasks that are independent
  - Example: Video application uses X
  - Video is highest since time-sensitive
  - Sends frame to X server and blocks
  - X server may be preempted by other medium priority task, hence delaying Video client
- To solve, use highest-locking priority (HLP) [19] in which task inherits priority when using shared resource
  - Example: display is shared resource so X server gets highest priority of blocking clients

Outline

- Introduction (done)
- Related Work (done)
- Requirements (done)
- Implementation (done)
- Evaluation (next)
- Conclusions

Evaluation

1) Behavior of time-sensitive applications running on TSL
2) The Overheads of TSL

- Setup:
  - Software
    - Linux 2.4.16
    - Robert Love's lock-breaking preemptible kernel patch
    - Proportion-period scheduler
  - Hardware
    - 1.5 GHz Intel P4 with 512 MB RAM

Latency in Micro Benchmarks

- Test low-level components of kernel latency: timer, preemption and scheduling
  - Time-sensitive process that sleeps for a specified amount of time (using nanosleep())
  - Results: 10 ms in standard Linux, few microseconds in TSL
- Test preemption latency under loads
  - Results: Linux worst case 100 ms (when copying data from kernel to user space), but typically less than 10 ms and is hidden by timer latency. TSL is 1 ms.

(Result details in [1])

Latency in Real Applications

- Tested two applications:
  - mplayer – a open-source audio/video player
  - Proportion-period scheduler - a kernel-level "application"

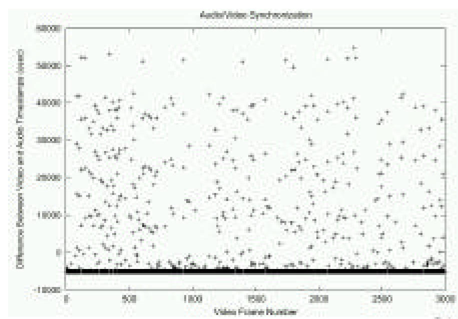
Mplayer Details

- Synchronizes audio and video using time-stamps
- Audio card used as timing source
- When video frame decoded, time stamp compared with audio clock.
  - If late, then play
  - If early, then sleep for time then play
- If kernel not responsive or has coarse timing, will be poor audio/video synch and high inter-frame display jitter

Testing MPlayer

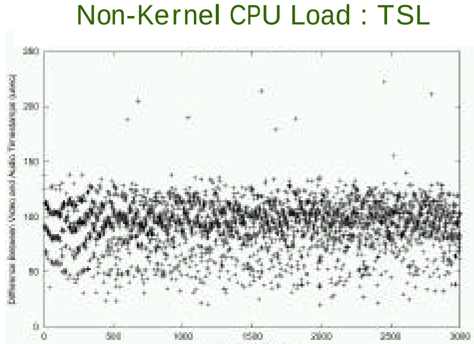
- Compare Linux with TSL under:
  - Non-kernel CPU load – run user-level stress test
  - Kernel CPU load – large (8 MB) mem buffer copied to a file (one `write()` call), 90% in kernel mode
  - File-system load – large dir (linux src, 13000 files, 180 MB data, ext2) copied (via DMA) recursively and flushed
- Fore ach test, run mplayer for 100 seconds at real-time priority

Non-kernel CPU Load : Linux



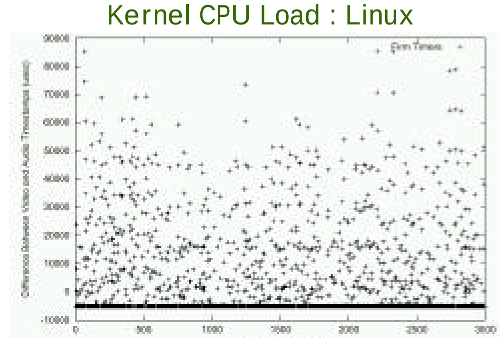
-5 ms to 50 ms when X server run normal prio

Non-Kernel CPU Load : TSL



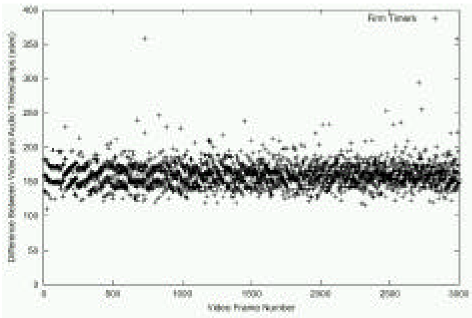
(X server at real-time, 250 microseconds (not y-axis))
(This config used for all others)

Kernel CPU Load : Linux



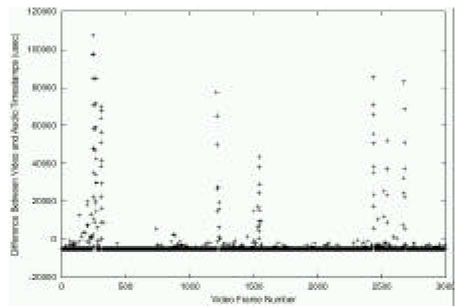
(90 msec for Linux, since done in non-preemptible section)

Kernel CPU Load : TSL



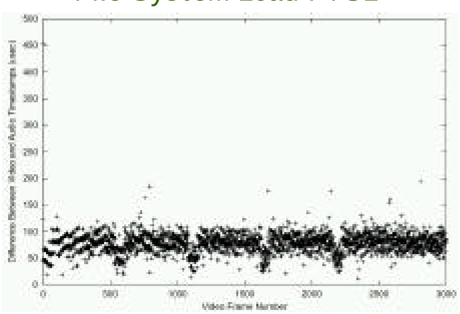
(Skew improves to less than 400 microseconds)

File System Load : Linux



(Skew often low, but as high as 120 msec)

File System Load : TSL

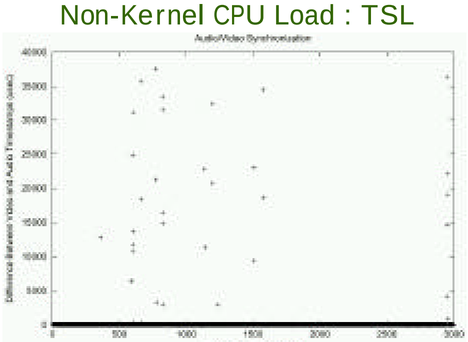


(Skel less than 500 microseconds, often lower)

Comparison with Real-Time Kernel

- Linux-SRT [6], includes finer-grained timers and reservation scheduler
  - (See figure 5a, 5b, 5c)
- Non-kernel CPU load skew less than 2ms, but as high as 7 ms (compare w/TSL of 250 microsec)
- Kernel CPU load worst case was 60 ms (compare w/TSL of 400 microsec)
- File-System load worst case was 30 ms (compare w/TSL of 500 microsec)
- Shows real-time scheduling and more precise timers insufficient. Responsive kernel also required.

Non-Kernel CPU Load : TSL



(Much lower, but can still be 35 msec)

Proportion-Period Scheduler

- Simultaneously ran 2 time-sensitive apps with proportions of 40% and 20% and periods of 8192 microsec and 512 microsec
- Each process records time via `gettimeofday()` and records in array
- Measure performance by differences in array compared with period

Maximum Deviation

	No Load		File System Load	
	Max Proportion Deviation	Max Period Deviation	Max Proportion Deviation	Max Period Deviation
Thread 1 Proportion: 40%, 3276.8 μs Period: 8192 μs	0.3% (≅25 μs)	5 μs	6% (≅490 μs)	534 μs
Thread 2 Proportion: 20%, 102.4 μs Period: 512 μs	0.7% (≅3 μs)	10 μs	4% (≅20 μs)	97 μs

Deviations low. Higher when load is high.
Maximum gives you bounds. Example: soft-modem needs CPU every 4 to 16 ms so could be supported.

System Overhead

- Costs of executing code at newly inserted preemption points
- Costs of executing firm timess

Cost of Preemption

- Memory access test (sequentially access 128 MB array), fork test (create 512 processes) and file-system access test (copy 2 MB buffers to 8 MB file)
  - Designed in [1], should be worst case
- Tests hit additional preemption checks
- Measure ratio of completion time under TSL / Linux
- Result: memory .42%+-.18%, fork .53%+-.06%, file sys had no overhead

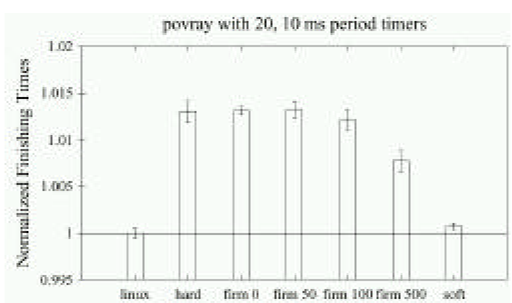
Firm Timers

- Firm timers user hard and soft timers. Costs:
  - Hard timers costs only – interrupt handling and cache pollution
  - Hard and soft timers common costs – manipulation timers from queue executing preemption for expired thread
  - Soft timers costs only – checking for expired timers

Firm Timers : Setup

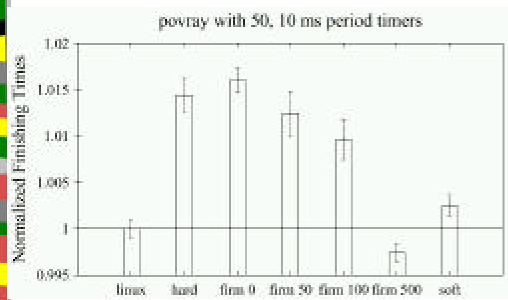
- Timer process - time-sensitive process is periodic task wakes up via `setitimer()` call, measures time, goes to sleep
- Throughput process – `povray`, a ray-tracing program rendering skyvase benchmark, measure elapsed time
- Run timer with 10 ms period since is supported by Linux

Firm Timer Overhead



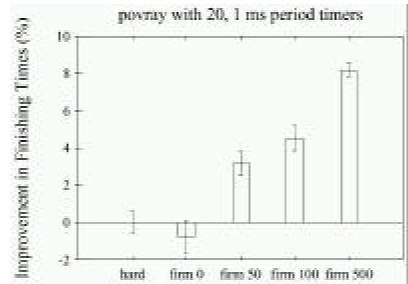
(Different overshoot values. 8 times w/ 95% confidence intervals)
(Only small decrease in overhead with larger overshoot)

Firm Timer Overhead



(Larger decrease in overhead since more timers)
(Linux slower with 500 since synchronizes 500 procs. Artifact of setup)

Firm Timer Overhead High Frequencies



(Compare with hard timers only since Linux cannot do 1 ms)

Discussion

- Firm timers lower overhead when soft-timer checks find timers
- Firm timers higher overhead when soft-timer checks find nothing and timer goes off
  - From their work, firm timers lower when more than 2.1% of timer checks find timer

Conclusions

- TSL can support applications needing fine-grained resource allocation and low latency response
  - Firm timers for accurate timing
  - Fine-grained kernel preemptibility for improving kernel response
  - Proportion-period scheduling for providing precise allocation of tasks
- Variations of less than 400 microseconds under heavy CPU and file system load
- Overhead is low