

Inferring Scheduling Behavior with Hourglass

John Regehr
School of Computing
University of Utah
regehr@cs.utah.edu

<http://www.cs.utah.edu/~regehr>

Abstract

Although computer programs explicitly represent data values, time values are usually implicit. This makes it difficult to analyze and debug *real-time* programs whose correctness depends partially on the time at which results are computed. This paper shows how to use Hourglass, an instrumented, synthetic real-time application, to make inferences about what is happening on a computer at millisecond and microsecond granularities. These inferences are possible because Hourglass records a very fine-grained map of when each of its threads runs, and because Hourglass supports a variety of *thread execution models* that model the properties and requirements of non-synthetic real-time applications. We conclude that between measurements and inferences, surprisingly detailed knowledge about scheduling behavior can be obtained without modifying, or even explicitly interacting with, the operating system kernel.

1 Introduction

Real-time applications such as games, audio playback, video display, and voice recognition need to finish computations by certain times, in addition to producing the right result, in order to operate correctly. Task execution can become complex because tasks share resources, because time-sharing schedulers such as the ones in Linux and FreeBSD execute complex algorithms, and because kernel activity such as hardware and software interrupt handlers can interfere with application execution. These factors often make it difficult to figure out what is really happening at microsecond and millisecond granularities when a real-time application runs. The situation is complicated by the many modified Linuxes that provide improved real-time services, since each has different performance characteristics. These real-time enhanced kernels fall into two main categories. First, there are those that improve the real-time performance of basic mechanisms, such as high-resolution timers [1], Robert Love's

preemptible kernel patch [12], Andrew Morton's lock-breaking patch [13], and the TimeSys Linux/GPL kernel [23], which includes a number of scheduler enhancements and also makes most device driver activity preemptible by threads. Second, there are those that change the CPU scheduling algorithm, such as Linux-SRT [4], QLinux [22], RED-Linux [25], and Linux/RK [14].

It is difficult to measure, analyze, and understand the behavior of real-time applications that are scheduled by these different Linux variants and by other operating systems. To address this problem we have developed *Hourglass*, a heavily instrumented synthetic real-time application that operates entirely in user space and requires no modifications to the operating system. Our focus is on Linux, but Hourglass also runs on Windows 2000 and FreeBSD; it should be easy to port to other Win32- and Unix-based systems.

2 Monitoring Scheduling Behavior

Broadly speaking, there are two ways to learn about the run-time behavior of an application: by instrumenting the application and by instrumenting the kernel. Each approach has advantages and disadvantages.

2.1 Instrumented Kernels

Kernel-based instrumentation and measurement techniques can be divided into approaches that monitor the execution of a single application and those that monitor the system as a whole. An example that fits into the first category is `ptrace()`, a system facility that permits a parent process to monitor and intercept any kernel calls made by a child process. The second category is exemplified by the Linux Trace Toolkit (LTT) [26], a general-purpose event logging facility for the Linux kernel. The important properties of LTT are that it logs events into physical memory, doing no I/O until requested, and that it does not add a lot of overhead to normal operations.

Some of the factors favoring use of an instrumented kernel include:

- Since the kernel is not treated as a black box, it is possible to figure out exactly why particular timing effects were observed rather than relying on indirect inferences.
- Since applications do not need to be modified, real applications can be run without adding instrumentation code that may increase complexity or affect their timing behavior.

2.2 Instrumented Applications

Instrumented applications fall into two categories. First, it is possible to add instrumentation to real applications and second, synthetic applications can be used. Real applications can be instrumented in a variety of ways: by hand, by linking against instrumented library routines, by interposing on library calls using macro or linker tricks, or by rewriting a binary. For example, *gscope* [6] is a visualization tool that uses the metaphor of an oscilloscope; it provides an API that applications can use to send signals to *gscope*, where they are processed and displayed. *ATOM* [21] exemplifies a very different approach: it uses binary rewriting to add customized instrumentation to an application. Because *ATOM* provides very fine-grained instrumentation it could be used to add timing instrumentation to real applications, although care would have to be taken to avoid slowing a program down too much.

Synthetic applications such as Hourglass need not be instrumented further because their entire purpose is to provide instrumentation. By abstracting away from real applications it is possible to obtain very predictable and controllable behavior, to get fine-grained timing measurements, and to easily model a wide variety of application scenarios without changing complex application code. On the other hand some application characteristics, such as patterns of synchronization between threads, may be difficult or impossible to model using a synthetic application.

Instrumented applications have the following benefits:

- The timing information reported is guaranteed to be authentic in the sense that actual application scheduling behavior is measured. This is often better than measuring timing in the kernel, where it may be difficult to instrument all code of interest, e.g. interrupt handlers, bottom-half handlers, etc. Furthermore, cache effects can be measured at the application level.

```
thrd_func () {
    sleep_until (my_start_time);
    while (now < my_finish_time) {
        run_workload ()
    }
}

main () {
    parse_command_line_args ()
    foreach (0..num_threads-1) {
        pthread_create (thrd_func)
    }
    sleep_until (overall_finish_time)
    print_results ()
}
```

Figure 1: Hourglass in a nutshell

- Since the kernel is not modified, there is no risk of destabilizing the system, or breaking or slowing down non-real-time applications. Also, there are no kernel patches to conflict with the many modified Linux kernels listed in Section 1.

2.3 Summary

Kernel-based instrumentation is the best way to measure OS-specific metrics such as the time to execute a particular code path in the kernel. A combination of kernel-based instrumentation and hand-inserted application instrumentation is almost always the right choice for debugging a specific real-time application on a specific operating system. On the other hand, for performing a comparative evaluation of real-time operating systems or for exploring application scenarios other than those provided by available applications, a user-space synthetic application is almost always the right choice.

3 Hourglass Structure and Internals

Hourglass is structured as a collection of cooperating threads: one for each thread that the user requests, plus the thread that initially runs `main()`. Figure 1 shows pseudocode for Hourglass. Most of the complexity is in `run_workload()`, which is covered in this section.

3.1 Detecting Gaps

While starting up, Hourglass allocates a memory buffer (about 5 MB long, by default) for storing an *execution trace*. The trace is generated by threads as they run: each time a thread detects a gap in its execution, it allocates a record in the trace buffer and uses it to store the start and end time of a continuous block of CPU time that the thread received. Taken together, these records produce a

```

while (1) {
  now = rdtsc ()
  if (now - last_exec > GAP) {
    rec = allocate_record ()
    rec.id = my_pid
    rec.start = exec_start
    rec.end = last_exec
    exec_start = now
  }
  last_exec = now
}

```

Figure 2: Algorithm for recording intervals of continuous CPU time received by a thread

map of how the CPU was used during a particular Hourglass run.

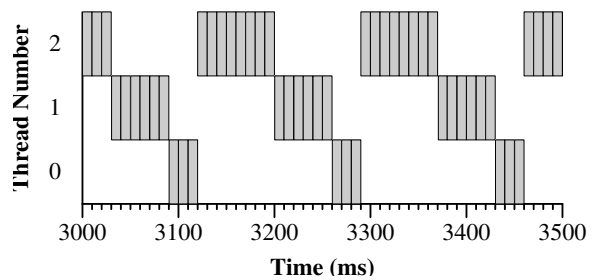
To learn when it is receiving CPU time, each thread continuously polls the Pentium timestamp counter by executing the `rdtsc` instruction, which returns a 64-bit quantity indicating the number of processor cycles since the machine was booted. If the difference between successive reads of the timestamp counter exceeds a `GAP` threshold, the thread considers its execution to have been interrupted and adds an element to the trace buffer recording the start and finish of the time interval during which it had uninterrupted access to the processor. By setting `GAP` appropriately, even very small interruptions, such as those caused by interrupt handlers that run for a few microseconds, can be detected. Figure 2 shows pseudocode for the gap-detection algorithm, and Figure 3 shows some execution traces produced by post-processing Hourglass output with `render_trace`, a script that is distributed with Hourglass. Gray rectangles represent continuous blocks of CPU time allocated to a thread. To produce these traces, Hourglass was run with this command line

```

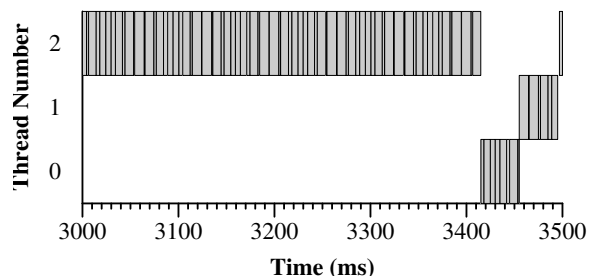
hourglass -n 3 -d 5s \
  -t 0 -p LOW \
  -t 1 -p NORMAL \
  -t 2 -p HIGH

```

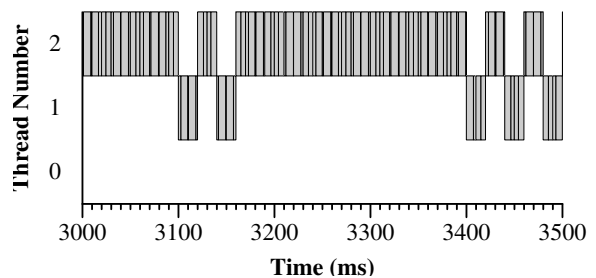
which creates three threads at different priorities and lets them record an execution trace for five seconds (Appendix A provides command line usage information for Hourglass). The top three traces in Figure 3 are 500 ms segments of the 5 s traces, and were respectively taken on an otherwise idle Linux machine, an otherwise idle Windows 2000 machine, and an otherwise idle FreeBSD machine. The bottom trace segment is 100 ms long in order to show more detail, and was taken on a Linux machine running `xmms`, an audio player application. The operating system versions used are described at the beginning of Section 5.



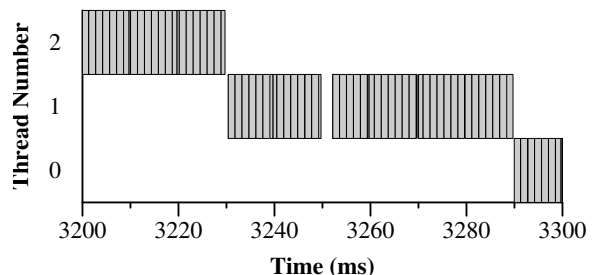
(a) Hourglass running by itself on plain Linux 2.4.17



(b) Hourglass running by itself on Windows 2000



(c) Hourglass running by itself on FreeBSD 4.5



(d) Detail of Hourglass sharing the processor with `xmms` on plain Linux 2.4.17

Figure 3: Example execution traces

The first trace clearly shows Linux's clock interrupts arriving every 10 ms, and it also shows that higher-priority threads run for longer than low-priority threads rather than running more often. Running for longer is the correct implementation for a server operating system since it minimizes context switch overhead. The second trace shows that the Windows 2000 scheduler is not nearly as fair to low-priority threads as the Linux scheduler is. In fact, the Windows 2000 scheduler is almost a static priority scheduler: in the presence of a high-priority CPU-bound thread, lower-priority threads only receive brief use of the CPU every few seconds [20, pp. 367–368]. The FreeBSD trace (third from the top) seems to show some sort of complex behavior caused by its multi-level feedback queue. FreeBSD lies between Linux and Windows 2000 with respect to scheduling fairness. Finally, the bottom trace is broken up by system activity: the C-Media sound card driver handles about 700 interrupts per second when `xmms` is active, and the act of reading mp3 data from disk causes about 20 IDE interrupts per second.

Keeping close track of when each thread runs provides a basis for implementing many useful capabilities. For example, by examining gaps in thread execution, we can determine how long various events of interest such as context switches and interrupts take to execute. By adding up the durations of intervals during which a thread had uninterrupted access to the CPU, a thread can determine if it has received enough processor time to perform some computation.

A number of details have to be taken care of for Hourglass to operate correctly. First, threads that require low-latency scheduling must run at a real-time priority, allowing them to preempt normal application threads. Second, the Hourglass process should be locked into physical memory in order to avoid reporting spurious delays that are caused by page faults. On systems that do not support `mlockall()`, Hourglass touches each page of dynamically allocated memory to ensure that it is mapped to a physical page before any timing operations are run. This is a hack, but it should work in situations where there is no memory pressure during an Hourglass run. Third, the value of the `GAP` constant needs to be chosen to be longer than the typical iteration time of the loop in Figure 2, but shorter than the length of the shortest actual gap in execution that a thread experiences. In practice, this is not difficult: on our test machine, an 850 MHz Pentium III, the execution trace loop usually takes about 225 ns to execute. Setting `GAP` to twice this value appears to avoid detecting spurious gaps while still detecting even fast interrupt handlers. Fourth, since `rdtsc` returns values in cycles, Hourglass needs to be aware of the clock speed of the processor it is run-

ning on. This is can be determined by comparing the rates at which `rdtsc` and `gettimeofday()` run.

3.2 Thread Execution Models

To facilitate the modeling of a variety of application scenarios, Hourglass supports a number of thread execution models. All thread models except the last, the latency test, record records of when they run in order to participate in the creation of an execution trace.

CPU bound: These threads simply run whenever they can.

CPU bound, scanning: These threads are CPU-bound, and they sequentially access elements in an array, modeling tasks with non-zero working sets in the data caches.

CPU bound, yielding: These threads are CPU-bound but voluntarily yield the processor each time they have received a configurable amount of CPU time.

Periodic: These threads have a characteristic execution time and period. During each period they run until they have received their execution time, at which point they block until the beginning of the next period. The timer that they block on can be selected at run time. Each time a periodic thread has not received its execution time by the end of its period, it registers a deadline miss. Periodic threads model real-time applications that have a characteristic rate, such as real-time audio processing, video decoding, or a software modem.

CPU bound, periodic: These threads have a characteristic execution time and period, but they never block. Rather, each time a thread receives its CPU time requirement, it simply begins another period. This models the class of applications such as games and other real-time simulations that must provide some minimum frame rate, but can opportunistically use extra CPU time to provide higher frame rates. For example, we might require that a game produce at least 20 frames per second. If it requires 10 ms to produce a single frame, then its execution time is 10 ms and its period is 50 ms. This thread will register a deadline miss any time a 50 ms period elapses where no frame is produced. If all CPU time is available to this thread, it will provide 100 frames per second.

Latency test: When these threads run they record the current time and then sleep until a period has passed. They are used to measure dispatch latency: the time between when a thread becomes ready and when it starts to run. Latency test threads are different from periodic threads in that they automatically synchronize their periods to the timer source rather than maintaining an independent period. In other words, each time they awaken

they compute their next wakeup time by adding their period to the current time instead of the time at which they should have awakened. This simplifies latency testing by eliminating phasing errors that can make things difficult for actual real-time applications.

It is easy to add new thread execution models to Hourglass. For example, it might be useful to add a new periodic task model where, instead of requiring the same amount of CPU time during each period, the execution time requirement follows a statistical distribution or uses a table-driven approach to simulate the deterministic variability shown by an MPEG-2 video decoder as it processes different types of frames.

3.3 Time and Timers

In most general-purpose operating systems it is hard to gain control of the CPU at a precise time, unless that time happens to be synchronized with a periodic clock interrupt. In many systems clock interrupts arrive every 10 ms. In Linux and Windows 2000, better timing can be obtained using the real-time clock (RTC), which can be programmed to interrupt the CPU at power-of-two frequencies; for example, at 1024 Hz it provides 976 μ s resolution — precise enough for most applications. Even better timers for Linux can be obtained through a kernel patch [1] that provides the POSIX timer interface to applications based on either the RTC, the programmable interrupt timer (PIT), or the ACPI power management timer. The last two can achieve small-microsecond accuracy. Windows 2000 provides *multimedia timers* that, on most platforms, provide a timer resolution of about 1 ms. Hourglass is capable of using all of these timers, and even of mixing them within a run.

3.4 Data Output and Analysis

Hourglass has three main design goals with respect to its output. First, all output happens after a run finishes, in order to avoid contaminating timing information with I/O effects. Second, Hourglass produces output that is as complete as possible, and is (in principle) human-readable. And finally, lines that contain data that is likely to be of interest to scripts are tagged with special strings that make them easy to identify.

4 Inferring Scheduling Behavior

Because Hourglass runs entirely in user space, the causes for gaps need to be inferred, and often this knowledge can only be statistical. The following list provides examples of some kinds of inferences that can be made using Hourglass, and it illustrates the distinction between statistical and non-statistical inference.

- If Thread 1 runs, a gap occurs, and then Thread 2 runs, one can make the non-statistical inference that a context switch occurred during the gap. Statistical inferences that can be made after looking at a number of such gaps include: (1) that the shortest gap indicates the fastest path through the context switch code, (2) that if there are many gaps of about the same length, and this length isn't much longer than the shortest gap, then the median member of this set of gaps is representative of the expected context switch time, and (3) that any very long outlier gaps have either been contaminated by other system activity such as interrupt handlers, or that the context switch code has an unpredictable execution time.
- If a context switch is observed between two threads that are known to never yield or block (voluntarily or involuntarily), one can make a non-statistical inference that the context switch was an involuntary preemption, and hence a clock interrupt must have occurred during the gap, in addition to a context switch.
- If the highest priority active thread is known to never block, gaps in its execution must be caused by high-priority non-thread activity such as interrupts and bottom-half device driver routines.
- If threads running on an idle machine have few gaps, and if threads running on the same machine under a particular workload (e.g. receiving network data) show many gaps, one can make the statistical inference that the workload is causing the gaps.
- If a thread running on an otherwise idle machine has occasional high dispatch latencies when using one kind of timer, but not another, then the statistical inference can be made that one of the timer implementations is inaccurate or broken. One can have higher confidence in this inference when running on a low-latency or preemptible kernel that almost never shows long “real” gaps.

If these kinds of inferences are insufficient, there are two potential solutions. First, it might be possible that a more clever use of Hourglass, or a modification to Hourglass, would permit stronger inferences to be made. Second, it might be necessary to stop viewing the kernel as a black-box, and to start using a package such as the Linux Trace Toolkit. Although Hourglass does not currently interact with LTT, it would be useful to be able to correlate results from the two sources. This would be straightforward because a single time line, from `rdtsc`, is available.

5 Usage Scenarios

The purpose of this section is to provide a number of examples showing how to use Hourglass to measure different kinds of scheduling behavior. We are *not* attempting a good comparative evaluation of the different operating systems; for this reason and because the effects being shown are easily repeatable from run to run, we usually omit confidence intervals for the data being presented.

The scenarios in this section are intended to tell a story. We begin with two ways to measure context switch costs — these provide useful bits of knowledge but often have little direct impact on application programmers. The next example shows how to measure dispatch latency, which is a useful metric for real-time application developers but is still in the operating system domain. The fourth example moves into the application domain by directly measuring the ability of a single application to meet its deadlines while the operating system is busy doing other things; the example after that focuses on using a response time analysis to make predictions about the interaction between several real-time applications. Finally, the last example is about *CPU reservations*, a scheduling abstraction for guaranteeing that the requirements of multiple real-time applications will be met even when there is no coordination or cooperation between applications.

All performance data for this paper were taken on a uniprocessor 850 MHz Pentium III with 512 MB of RAM. Linux experiments were run on kernel version 2.4.17 unless otherwise specified. The high-resolution timer patch [1] is version 2.4.17-2.0. The preemptible kernel uses Robert Love’s preempt-kernel-rml-2.4.17-3 and lock-break-rml-2.4.17-2 patches [12]. TimeSys Linux/GPL [23] is their version 3.0.108, based on Linux 2.4.7. FreeBSD measurements were taken on 4.5-RELEASE, and on that platform Hourglass uses the Linuxthreads package in order to get kernel threads. Windows 2000 measurements were taken on service pack 2.

5.1 Measuring the Direct Cost of Context Switches

One of the most straightforward uses of Hourglass’s ability to measure gaps in thread execution is to infer the cost of running the kernel context switch code. To do this, invoke Hourglass with:

```
hourglass -d 60s -n 10 -a -p NORMAL -w CPU
```

or:

```
hourglass -d 60s -n 10 -a -p NORMAL \  
-w CPU_YIELD 0.9ms
```

These commands cause Hourglass to run for 60 seconds after creating ten threads, all running at the default time-sharing priority, all of which are CPU-bound. The only difference between the two commands is that the second causes each thread to yield the processor each time it has received 900 μ s of processor time, permitting us to distinguish between voluntary context switches (caused by yielding) and involuntary context switches (quantum expirations). Involuntary context switches are more expensive because they always occur at the same time as a clock interrupt. The output from these commands contains many lines of the following form:

```
0 442.323192 443.222287 0.899095 0.002184  
1 443.224485 444.123644 0.899159 0.002198  
9 444.125827 445.024925 0.899098 0.002183
```

The first column is the thread id, the second and third columns show the start and finish times of a gap-free interval of CPU time that the thread received, the fourth column is the duration of the interval (the difference between the second and third columns), and the last column is the “gap” between the start of the record being reported and the end of the previous record. All times are in milliseconds.

Hourglass is distributed with a Perl script `process_ctx` that produces a histogram of context switch times from Hourglass output, and also performs some simple statistical analyses. Figure 4 shows some histograms of context switch times for thread expirations on several different OS configurations. These histograms reveal some interesting performance characteristics of the different systems. First, we can see that Linux with high-res timers appears to have a more efficient clock interrupt handler than standard Linux, while the TimeSys kernel appears to be a bit less efficient than the high-res kernel but is still more efficient than standard Linux. Second, we see that RML’s preemptible kernel and lock breaking patches don’t appear to affect context switch time. Third, the TimeSys Linux/GPL kernel, Windows 2000, and FreeBSD all cause about 50 context switches per second, rather than about 18 context switches per second for the Linux kernels based on 2.4.17. In other words, a modern stable Linux kernel provides CPU-bound threads with 60 ms quanta; TimeSys Linux (which is based on 2.4.7), Windows 2000, and FreeBSD give them 20 ms quanta; the kernel distributed with RedHat Linux 6.2 causes only about 7 context switches per second. The context switch cost for FreeBSD 4.5 kernel threads is over 30 μ s — considerably higher than the other OSs measured. This is because FreeBSD 4.5 Linuxthreads have not been optimized, and fail to take advantage of the opportunity to avoid page table operations when switching between

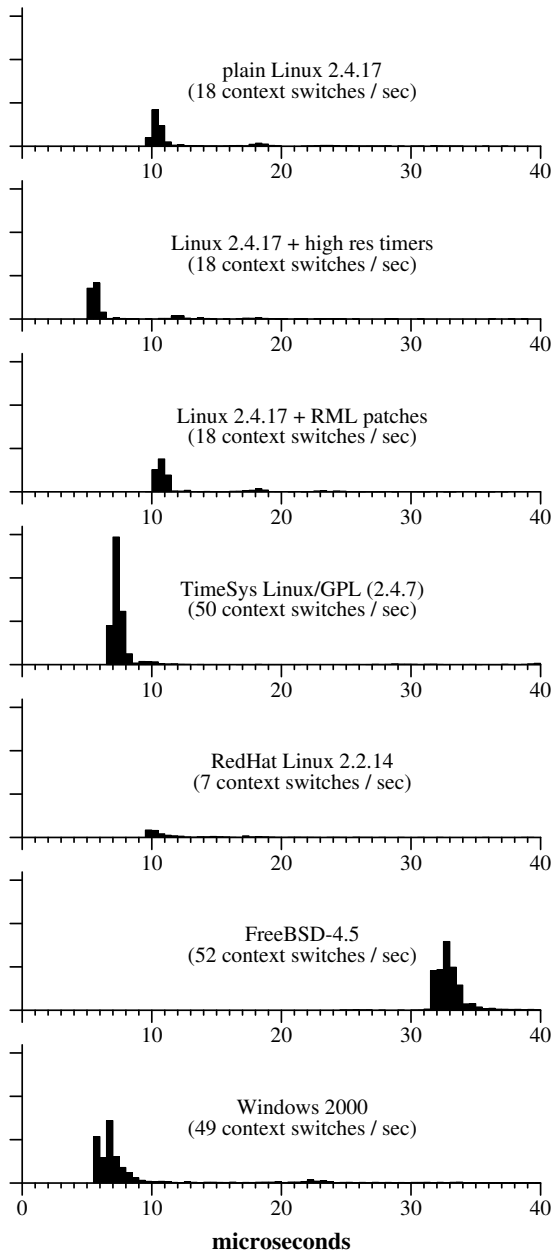


Figure 4: Histograms of times for involuntary context switches (e.g. caused by quantum expirations) between CPU-bound threads within a single process

threads in the same address space. We verified this by comparing the costs shown for FreeBSD in Figure 4 with the cost to context switch between processes. There was no statistically significant difference.

5.2 Measuring the Indirect Cost of Context Switches

Time spent in the kernel accounts for only one of the costs of context switches. Another cost, caused by lost cache locality, is potentially more severe. To see this, consider a thread whose working set in the L2 cache is 100 KB. If this thread is descheduled and, when scheduled again, starts running on a cold cache, it will spend time executing slowly until its working set again completely resides in the cache. The 850 MHz machine that we use has a memory read bandwidth of approximately 300 MB/s, so reading the 100 KB will take about $333 \mu\text{s}$ — far longer than the context switches that we saw in the previous section that took on the order of $10 \mu\text{s}$. This explains why scheduling quanta have not gotten much smaller over the past few decades, even though microprocessors have gotten thousands of times faster: for applications with non-trivial cache state, context switch times are tied to DRAM speeds, which grow much more slowly than processor speeds. For example, 1 ms scheduling quanta would lead to extremely good average application response time — this would be great for multimedia and interactive applications. However, instead of causing a 1% overhead, as the $10 \mu\text{s}$ context switch costs from the previous section might lead us to believe, this would create a 33% overhead if the average application has a 100 KB working set in the L2 cache.

Measuring application slowdown due to lost cache locality requires a more subtle technique than the one we used in the previous section: rather than measuring gaps, we must directly measure progress made by the application. The experimental methodology is as follows. To model a thread with, say, a 32 KB working set, Hourglass creates a thread that allocates 32 KB and repeatedly scans through it. For the control, Hourglass creates a single thread with a specified working set and measures how much “work” it can accomplish during a specific period of time. Each pass over the thread’s working set is considered to be a unit of work. We can then compare this amount of work with the total work performed by ten threads during the same time period. With all other factors being equal, any lost work for the ten-thread case must be caused by context switch overhead. The performance penalty C per context switch can be computed as

$$C = \left(\frac{W_1 - W_{10}}{W_1} \right) \left(\frac{T}{N_{10}} \right)$$

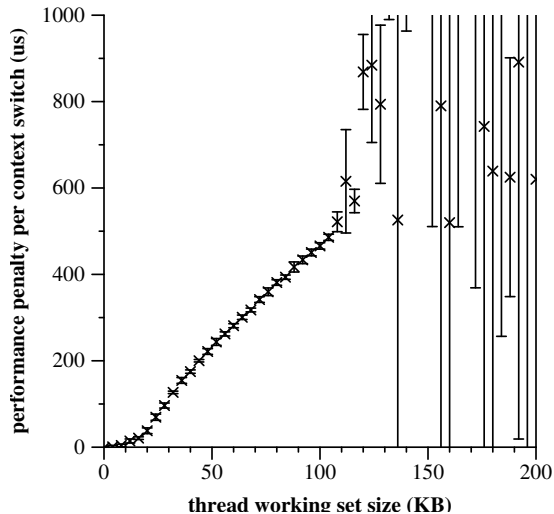


Figure 5: Thread working set size vs. performance lost to context switches on an 850 MHz Pentium III

where W_1 and W_{10} are respectively the total amount of work done by one and ten threads, T is the duration of an experimental run, and N_{10} is the number of context switches that occurred during the ten-thread run. In other words, the first term represents the fraction of work lost due to context switches; when multiplied by the duration of the experiment this gives the total performance penalty due to context switches; dividing by the number of context switches gives the penalty per switch.

Figure 5 shows the results of running this experiment for a variety of thread working set sizes on Linux 2.4.17. For each working set size, 15 trials were run; each trial consisted of a 45-second run of Hourglass with one thread, and a 45-second run with ten threads. Confidence intervals were calculated at 95% using the t -test [7, pp. 209–211]. The surprising thing about this graph is not how bad the results are (in fact, the results are roughly in line with the back-of-the-envelope calculation from the beginning of this section), but how unpredictable the data becomes for working set sizes over 128 KB (half the size of the L2 cache on a Pentium III Coppermine chip). We don’t have a good explanation for this, but it probably has to do with the lack of page coloring logic in the Linux memory allocator — this can lead to cache conflict misses within a single application that are very unpredictable from run to run. Similar experiments, when run on FreeBSD, produced much more predictable results (see [5] for some discussion of the FreeBSD VM system, including page coloring optimizations).

Linux version	Samples later than:			
	1 ms	5 ms	10 ms	50 ms
2.2.14 RedHat	179	112	112	0
2.4.17 plain	284	56	45	1
2.4.17 RML	202	8	2	0
2.4.7 TimeSys	0	0	0	0

Table 1: Dispatch latencies for various Linux kernels with contention from filesystem and network activity

5.3 Measuring Dispatch Latency

A real-time operating system should be able to start running a high-priority thread shortly after it becomes ready. The time interval between when a thread becomes ready and when it begins to run is called *dispatch latency*. Dispatch latency is a serious problem for applications such as real-time audio where failure to produce a sample on time can result in audible artifacts. The point of the various lock-breaking and preemption patches for Linux is to get a substantial reduction in dispatch latency without overly increasing system overhead or reducing system maintainability. Running Hourglass with

```
hourglass -d 6m -n 1 -t 0 -p RTHIGH \
-w LAT 5.3ms -i RTC
```

creates a high-priority thread that uses Hourglass’s built in real-time clock timers to wake itself up every 5.3 ms (the reason for the funny period is that it approximates a multiple of the power-of-two frequencies supported by the RTC). The Hourglass output then contains many lines like

```
latlate: 97.843206
latlate: 54.101933
latlate: 62.381242
```

indicating, in microseconds, the lateness of successive timer expirations.

Our experimental procedure for measuring dispatch latency was to run Hourglass in latency test mode for six minutes while running background work that creates scheduling contention. The background work that we chose was to concurrently (1) run a recursive `grep` over about 900 MB of data in about 60,000 files that is NFS3-mounted over a loopback connection, and (2) run a recursive `grep` over another copy of the same data that is NFS3-mounted with the server running on a separate machine. This is not necessarily a good choice for a real-world latency test, but it’s fine for a demonstration.

Table 1 shows the results of this experiment for several Linux kernel versions where the total number of samples taken per operating system version is about

Linux version	Missed deadlines	Throughput
control 1	0.0%	–
control 2	–	94 Mbps
2.4.17 plain	33.0%	68 Mbps
2.4.17 RML	0.4%	66 Mbps
2.4.7 TimeSys	0.0%	59 Mbps

Table 2: A real-time thread that requires 4 ms of CPU time during every 5 ms can miss deadlines due to Ethernet receive processing

65,000. The results are pretty much what we would expect: the standard Linux kernels are fine most of the time, but they display occasional high latencies. The kernel with RML patches (supporting preemption and breaking of some locks [12]) does better, and the more aggressively modified TimeSys Linux/GPL kernel [23] never shows a latency above about 600 μ s.

5.4 Meeting Application Deadlines during Receive Processing

Although average- and worst-case dispatch latency are useful metrics for characterizing real-time operating systems, they only tell part of the story. As far as applications are concerned, what is important is not when they *start* running, but when they *finish*. In this section we show how kernel activity, in particular network receive processing, can affect applications’ ability to get work done. Other kinds of receive processing, such as disk and USB, can also cause scheduling problems [17]. The experimental procedure was to hit the test machine with a full-speed TCP stream from another machine over 100 Mbps Ethernet using `netperf` at the same time that a real-time application attempts to meet its deadlines. If the receive processing interferes too much with application execution, deadlines will be missed.

The real-time application has very demanding requirements: it is a periodic thread that requires 4 ms of CPU time during each 5 ms period. This models a thread used for professional-quality real-time digital audio, where the end-to-end processing latency for sound data must be extremely low. The thread has the highest possible priority and uses real-time clock timers to schedule itself. The command line that does this is:

```
hourglass -d 20s -n 1 -t 0 -p RTHIGH \
-w PERIODIC 4ms 5ms -i RTC
```

We first ran two controls: one with only the real-time task, and one with only network receive processing. The results, shown in the first two lines of Table 2, were that all of the versions of Linux that we tested can meet

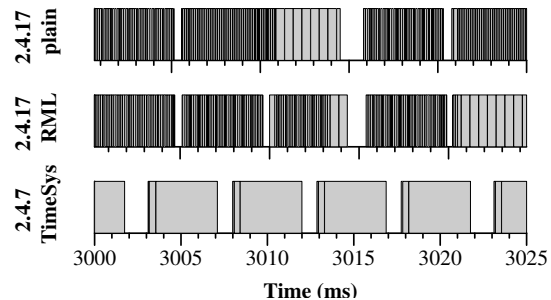


Figure 6: The effects of Ethernet receive processing on the execution of a periodic real-time thread

all deadlines on an otherwise quiet machine (control 1), and that a machine running only `netperf` can receive 94 Mbps of Ethernet data (control 2). The third line in Table 2 shows that plain Linux permits the application to miss about a third of its deadlines, but that it achieves the highest throughput. The preemptible Linux kernel with lock breaking patches misses less than one percent of its deadlines (53 out of about 12,000 deadlines, to be precise) and it gets only marginally lower throughput than plain Linux. Finally, the TimeSys Linux/GPL kernel misses no deadlines at all, but at the cost of reduced network throughput.

The tradeoff here is clear: by more strictly respecting the programmer’s choice to run a user-level thread in preference to kernel activity, the two real-time-enhanced kernels permit real-time workloads to miss fewer deadlines. However, this causes network processing to be delayed, resulting in lower throughput. Figure 6 shows 25 ms segments of the execution traces generated by Hourglass while running these experiments. The top trace, generated on plain Linux, shows receive processing — interrupt and bottom-half handlers — interfering with thread execution to a significant extent. In the middle execution trace the 5 ms periodic structure of the real-time thread’s execution is a little more apparent. However, there is still considerable interference: this is because RML’s preemptible kernel patches only apply to the top half of the kernel; interrupts and bottom-half handlers can still freely run during the execution of a high-priority thread. Finally, the execution of the TimeSys Linux/GPL kernel shows that the real-time thread almost entirely locks out Ethernet processing: in this kernel both bottom-half handlers and most of each interrupt handler are run in thread context, and are therefore preemptible by user-mode threads.

The results of this experiment are sensitive to a number of parameters. For example, although plain Linux 2.4.17 did not permit a thread requiring 4 ms of CPU time during every 5 ms to meet most of its dead-

lines while the host receives full-bandwidth data over 100 Mbps Ethernet, under the same conditions a thread that requires 8 ms of CPU time during every 10 ms can meet essentially all of its deadlines. However, in this case the incoming bandwidth is only 52 Mbps. There are several factors at work here. First, at small-millisecond granularities operating system performance can be complex and unpredictable: even a small number of cache misses can affect performance at this level. Second, the window-based nature of TCP makes it somewhat undesirable for use in performance studies like this because its behavior may be highly nonlinear with respect to other factors in the experiment. The reduced bandwidth for the 10 ms periodic thread is most likely a result of the longer blocks of CPU time allocated to the real-time thread, which starves the `netperf` application for long enough that the sender must back off. Also, it is important to realize that 100 Mbps Ethernet puts a fairly modest load on a modern CPU: gigabit Ethernet, FireWire, and USB 2.0 can all potentially cause much larger receive processing loads than the ones measured here.

In summary, doing this kind of performance analysis “right” is very difficult because (1) the results are often highly sensitive to details of the how the experiment is performed, and (2) tuning the real-time performance of an OS kernel is inherently a matter of tradeoffs, and different people have different metrics for success. For example, it is possible to find otherwise reasonable people who are almost solely concerned with any one of the following metrics: average-case application throughput, real-time response, kernel code cleanliness and clarity, scalability to large multiprocessors, and scalability to small embedded devices.

5.5 Determining whether Concurrent Applications can meet Deadlines

The previous section addressed the ability of a single task to meet its deadlines under adverse conditions. In this section we consider multiple tasks. The motivation for this includes scenarios where, for example, a user is concurrently using a software modem, playing a game or listening to music, and encoding live video in the background. Modern PC hardware is capable of performing all of these tasks concurrently, provided that the operating system schedules each application at the proper times.

The workload is comprised of two tasks. The first task models either audio or software modem processing, and requires that a thread receive 3 ms of processor time during every 8 ms period. Software modems [10] and real-time audio can have tight latency requirements that are in this range. The second task represents a video

application that must maintain about 30 frames per second: it requires 17 ms of CPU time during each 33 ms period. We assign priorities to the two tasks in rate-monotonic fashion: the application with the shorter period gets a higher priority. Then, we use a standard real-time schedulability analysis to see if each application can be expected to meet its deadlines. The analysis returns answers as *worst-case response times* — the longest possible time between a task becoming ready and finishing its computation after taking into account all possible interleavings between the tasks [24]. If the response time of a task is less than or equal to its period, the task should work; otherwise it probably will not. In this example the response time of Task 0 is 3 ms, and Task 1 is 29 ms. In other words, there is no possible interleaving of the executions of the two tasks that can cause Task 0 to finish more than 3 ms after it becomes ready to run (this should be obvious — since Task 0 has higher priority, Task 1 cannot interfere with its execution), and Task 1 will always finish its work by 29 ms after it becomes ready, leaving 4 ms of slack before the end of its period.

To model this scenario in Hourglass, we run

```
hourglass -n 2 \
-t 0 -p RTMED -w PERIODIC 3ms 8ms -i HR \
-t 1 -p RTLOW -w PERIODIC 17ms 33ms -i NATIVE
```

Here, we give Task 0 a high-resolution timer to model the fact that we are considering it to be interrupt-driven, where the interrupts come from either the soft modem or audio hardware. Task 1, on the other hand, uses `usleep()` to effect wakeups. The result of this experiment on a Linux kernel with the high-resolution timer patch contains lines that look something like

```
thread 0: missed 0 deadlines, hit 1001
thread 1: missed 142 deadlines, hit 143
```

meaning that Task 1 misses about as many deadlines as it hits. There are two reasons for the disagreement between the response time analysis and Hourglass output. First, the 10 ms granularity of `usleep()`-based timers results in *release jitter*. Release jitter occurs when a thread is conceptually ready to run earlier than the operating system becomes ready to run it. To see how this happens here, consider the case where Task 1, the video application, finishes a frame just four or five milliseconds before it is due to start processing the next frame. It then puts itself to sleep, but may have to wait 10 ms for the next clock interrupt to arrive and wake it up. The added delay can cause it to miss deadlines. The second reason for disagreement is that if a thread in the real-time priority class calls the Linux implementation

of `usleep()` with an argument of 2 ms or smaller, the kernel will actually busy-wait until the expiration time. This has the effect of increasing the worst-case run time of Task 1 from 17 ms to 19 ms.

Fortunately, the response time analysis is able to take release jitter into account. Re-running the analysis for Task 1 with 8 ms of release jitter and a 19 ms worst-case compute time shows that its worst-case response time is 39 ms. This is longer than its period, and therefore Task 1 is not guaranteed to meet all deadlines.

Since the task set is *infeasible*, or not guaranteed to allow all tasks to meet all deadlines, we now consider ways to make it feasible. One way is to schedule Task 1's wakeups using a timer that causes less release jitter and avoids the harmful busy-waiting; another way is to decrease the run-time of one of the tasks enough that the tasks become feasible in spite of these factors. To test the first method, we invoke Hourglass with

```
hourglass -n 2 \  
-t 0 -p RTMED -w PERIODIC 3ms 8ms -i HR \  
-t 1 -p RTLOW -w PERIODIC 17ms 33ms -i HR
```

which produces these results

```
thread 0: missed 0 deadlines, hit 1003  
thread 1: missed 0 deadlines, hit 304
```

on a Linux kernel with the high-res timers patch. This corroborates the initial response time analysis in this section that predicted that the task set is feasible.

To apply the second method, assume that we have reduced the execution time of Task 1 to 12 ms by decreasing the resolution of the video display or by making use of a hardware accelerator. The response time analysis indicates that in this case all deadlines can be met. We test this by running

```
hourglass -n 2 \  
-t 0 -p RTMED -w PERIODIC 3ms 8ms -i HR \  
-t 1 -p RTLOW -w PERIODIC 12ms 33ms -i NATIVE
```

which gives these results:

```
thread 0: missed 0 deadlines, hit 1001  
thread 1: missed 0 deadlines, hit 303
```

Papers by Audsley et al. [3] and Tindell et al. [24] are good sources of information about response time analysis for real-time task sets. SPAK [15] is a static priority analysis kit that implements response time analyses for a variety of task set assumptions, as well as a task simulator that can serve as a middle ground between response-time analysis and Hourglass for testing the ability of task sets to meet their deadlines.

5.6 Using CPU Reservations

The response time analysis used in the previous section requires a *closed system*, where all real-time applications and their requirements must be known before the analysis can be performed. However, when real-time applications are run on general-purpose operating systems like Linux the *open system* model is usually more appropriate: real-time applications can begin or end at any time, and applications can change their requirements freely. The standard Linux kernel can be used as an open system, but there is a problem: without knowing the requirements of the other applications, it is impossible to know, for a given application, what priority should be assigned to it in order to allow it to operate correctly, and to not interfere with existing applications. This lack of coordination between applications can lead to *priority inflation* where developers overestimate the priority at which their application should run, since a badly performing application reflects negatively on its developer. In the extreme case, this results in excessive migration of code into interrupt handlers [10]. Also, priority-based scheduling suffers from the problem that a misbehaving application can stop the progress of all applications running at lower priorities. Furthermore, rate-monotonic scheduling assigns priorities solely based on the periods of applications; this can result in anomalies where, for example, a critical application is given lower priority than an unimportant application.

In recent years scheduling solutions have been developed that address all of these problems. They go by different names, but the core abstraction is always some form of *CPU reservation*. For example, a video application could reserve 5 ms of CPU time during every 30 ms period, in order to ensure that it can always display 33 frames per second. If the operating system agrees to provide the reservation, it is essentially guaranteeing that the CPU time will always be available. This sort of guarantee relieves developers of the burden of finding a good priority for their application. However, it adds the non-trivial difficulty of specifying application requirements — this is probably the main reason that reservation schedulers have not yet made it into mainstream operating systems.

Hourglass provides direct support for making CPU reservations. Currently it supports only Linux/RK from CMU [11] and TimeSys Linux/CPU [23] (a proprietary extension to TimeSys Linux/GPL). However, reservation functionality is modular and support for new schedulers can be added with minimal effort.

To illustrate the use of CPU reservations, we return to the task set from the previous section. To run the

tasks in CPU reservations rather than using priority-based scheduling, we invoke Hourglass like this:

```
hourglass -n 2 \  
-t 0 -w PERIODIC 3ms 8ms -rh 3ms 8ms \  
-t 1 -w PERIODIC 17ms 33ms -rh 17ms 33ms
```

The resulting output contains the lines:

```
thread 0: missed 985 deadlines, hit 12  
thread 1: missed 106 deadlines, hit 197
```

Obviously things did not go well during this run. The mistake was to reserve exactly as much CPU time as each thread needed — this makes them vulnerable to minute perturbations in the schedule. Rather, we should reserve just a little extra CPU time like this:

```
hourglass -n 2 \  
-t 0 -w PERIODIC 3ms 8ms -rh 3.1ms 8ms \  
-t 1 -w PERIODIC 17ms 33ms -rh 17.1ms 33ms
```

The resulting output contains the lines:

```
thread 0: missed 0 deadlines, hit 1000  
thread 1: missed 0 deadlines, hit 303
```

For real applications, we would have done two things differently. First, we would have given them *soft* CPU reservations instead of *hard* CPU reservations — hard reservations enforce a strict upper limit on the amount of time a thread will receive, while soft reservations guarantee that applications will receive a minimum amount of CPU time, but permit them to receive extra time if it is available. Hourglass accepts the `-rs` command-line option to provide a soft reservation. Second, rather than just reserving 0.1 ms of extra CPU time per thread per period, we would tune the amount of over-reservation to match the application’s importance (e.g. a critical application gets a bigger reservation, all other things being equal) and its expected variability (e.g. an application with high variance in run-time, such as an MPEG video decoder, would get a larger reservation than a more predictable application).

6 Related Work

Hourglass fits into the general methodology that has been called *gray-box systems* [2]. The name refers to the fact that these techniques make inferences about system behavior by combining the results of observations with general knowledge about internal structure — the operating system is treated as a mixture between a black-box and a white-box.

Section 2 compared Hourglass with other ways to measure or infer scheduling behavior. As far as we know, Hourglass has no direct competition — there are no publicly available tools that occupy the same niche. However, Hourglass was influenced by `txofy`, a tool developed by Mike Jones and others for internal use at Microsoft Research; it was instrumental in producing data for a number of papers [8, 9, 16, 17, 18]. The gap detection algorithm from Section 3.1 was first developed for `txofy`. However, the two tools have diverged considerably: `txofy` was a one-off tool specifically designed to monitor and debug CPU reservation schedulers; it supports only a single source of timers, a single thread model, and a static set of threads (it has, however, been modified to support multiprocessors). Hourglass, on the other hand, is a general framework for making inferences about scheduling behavior; it is portable and extensible, and provides direct support for all techniques described in Section 5.

7 Availability and Status

Hourglass was written from scratch and is released under a BSD-style license. It runs on Linux, FreeBSD, and Windows 2000. The Hourglass home page is:

<http://www.cs.utah.edu/~regehr/hourglass>

Since web links tend to die, in the long run we suggest the alternate strategies of looking for Hourglass at the Flux Group web site: <http://www.cs.utah.edu/flux> or using the dominant web search engine to look for “regehr” and “hourglass.”

Hourglass currently supports uniprocessor x86-based systems. Both of these limitations could be addressed in straightforward ways. For example, Alphas support the `rpsc` instruction and UltraSPARCs have a `%tick` register; these are basically equivalent to the `rdtsc` cycle counter on x86 machines. To support multiprocessors Hourglass would require that the cycle counters on all processors are synchronized. Although Hourglass could not tell which processor each thread runs on without help from the kernel, it is possible to infer a consistent timeline from a multiprocessor execution trace. For example, if it is observed that Thread 1 receives a continuous segment of CPU time while Thread 2 stops running and Thread 3 starts, we can infer that Thread 1 was on one processor, and on a different processor Thread 2 was preempted in order to run Thread 3.

A more brute-force approach to generating accurate execution traces on multiprocessors would involve writing a minimal kernel extension that overloads the high-order bits of the cycle counters in such a way that any timestamp value could be identified as coming from a

particular processor — kernel support is required because writing to the model-specific x86 registers is a privileged operation. This approach would give accurate, unambiguous multiprocessor execution traces but would cause problems if the kernel or some other application relies on accurate or synchronized timestamp counters.

Machines such as laptops and Pentium4s can vary their clock speeds dynamically to deal with heat and power issues, making `rdtsc` less useful or useless. We have not attempted to address this problem since at present it's easy to avoid these processors. Dynamic speed scaling is not only a problem for programs like Hourglass, but also for actual real-time applications that depend on getting a certain amount of work done during a given time period.

Finally, an inconvenience of Hourglass on Unix-like systems is that users need root privileges to create threads in the real-time priority class and to pin pages into physical memory.

8 Conclusion

The contributions of this paper have been (1) to show that a lot of detail about operating system scheduling behavior can be inferred using an instrumented application and a little knowledge about internals, (2) to present some new gray-box inference techniques for learning about scheduling behavior, and (3) detailed descriptions of how to use a freely available tool to perform these measurements.

Although there are tools for measuring scheduling latency [19], Hourglass is a more comprehensive user-level real-time test application than any other software that we know of. It has been invaluable for discovering what is really happening to applications at the granularity of microseconds and milliseconds. We believe that this kind of application will become increasingly important as multimedia and other soft real-time applications become a more important part of the mix of programs that users run.

Acknowledgments

The author would like to thank Jason Baker, Paul Davis, Toon Moene, Mac Newbold, and Pat Tullmann for providing constructive feedback on drafts of this paper. Eric Eide provided graph and LaTeX expertise. Terry Lambert and others on the FreeBSD-hackers mailing list helped with analysis of performance on that OS. The Utah Emulab folks provided a great platform for running the experiments reported here.

This work was supported, in part, by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, under agreements F30602-99-1-0503 and F33615-00-C-1696.

References

- [1] George Anzinger. The Linux High Resolution Timers Project. <http://high-res-timers.sourceforge.net/>.
- [2] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, November 2001.
- [3] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
- [4] Stephen Childs and David Ingram. The Linux-SRT integrated multimedia system: Bringing QoS to the desktop. In *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, Taipei, Taiwan, May 2001.
- [5] Matthew Dillon. Design elements of the FreeBSD VM system. *Daemon News*, January 2000. http://www.daemonnews.org/200001/freebsd_vm.html.
- [6] Ashvin Goel and Jonathan Walpole. Gscope: A visualization tool for time-sensitive software. In *Proc. of the 2002 USENIX Annual Technical Conf. FREENIX Track*, Monterey, CA, June 2002.
- [7] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.
- [8] Michael B. Jones and John Regehr. CPU Reservations and Time Constraints: Implementation experience on Windows NT. In *Proc. of the 3rd USENIX Windows NT Symposium*, pages 93–102, Seattle, WA, July 1999.
- [9] Michael B. Jones, John Regehr, and Stefan Saroiu. Two case studies in predictable application scheduling using Rialto/NT. In *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 157–164, Taipei, Taiwan, May 2001.
- [10] Michael B. Jones and Stefan Saroiu. Predictability requirements of a soft modem. In *Proc. of the ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Cambridge, MA, June 2001.
- [11] Linux/RK: A Resource Kernel for Linux. <http://www-2.cs.cmu.edu/~rajkumar/linux-rk.html>.
- [12] Robert Love. Preemptible kernel and lock-breaking patches for Linux. <http://www.tech9.net/rml/linux>.
- [13] Andrew Morton. Low Latency Linux. <http://www.zip.com.au/~akpm/linux/schedlat.html>.

- [14] Rangunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource Kernels: A resource-centric approach to real-time and multimedia systems. In *Proc. of the SPIE/ACM Conf. on Multimedia Computing and Networking*, pages 150–164, San Jose, CA, January 1998.
- [15] John Regehr. SPAK: a static priority analysis kit. <http://www.cs.utah.edu/~regehr/spak>.
- [16] John Regehr. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, May 2001. <http://www.cs.utah.edu/~regehr/papers/diss>.
- [17] John Regehr and John A. Stankovic. Augmented CPU reservations: Towards predictable execution on general-purpose operating systems. In *Proc. of the 7th Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 141–148, Taipei, Taiwan, May 2001.
- [18] John Regehr and John A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, London, UK, December 2001.
- [19] Benno Senoner. Latencytest. <http://www.gardena.net/benno/linux/audio>.
- [20] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.
- [21] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, Orlando, FL, June 1994.
- [22] Vijay Sundaram, Abhishek Chandra, Pawan Goyal, Prashant Shenoy, Jasleen Sahni, and Harrick Vin. Application performance in the QLinux multimedia operating system. In *Proc. of the 8th ACM Conf. on Multimedia*, Los Angeles, CA, November 2000.
- [23] TimeSys Linux/GPL. <http://www.timesys.com/products>.
- [24] Ken Tindell, Alan Burns, and Andy J. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Real-Time Systems Journal*, 6(2):133–151, March 1994.
- [25] Yu-Chung Wang and Kwei-Jay Lin. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, pages 246–255, Phoenix, AZ, December 1999.
- [26] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proc. of the USENIX 2000 Annual Technical Conf.*, pages 1–14, San Diego, CA, June 2000.

A Command Line Arguments

This is Hourglass 0.5

usage: hourglass <options>

global options:

```
-n <number of threads to create> -- this is
the only mandatory option
-d <duration of experiment> (default is 10s)
-c -- print raw execution trace (in cycles and
not based at zero) in addition to standard
trace (which starts at zero and is in ms)
-t <thread number for subsequent options to
affect in the range 0..n-1>
-a -- subsequent per-thread options apply to
all threads; this remains in effect until
the next '-t' option is encountered
-e -- number of records in execution trace;
default is 300000 (about 5 MB)
```

per-thread options:

```
-p <priority> -- priority is one of:
IDLE, LOW, NORMAL, HIGH, HIGHEST
(timesharing -- default is NORMAL)
RTLOW, RTMED, RTHIGH (real-time)
-rh <amount> <period> -- request a hard
CPU reservation
-rs <amount> <period> -- request a soft
CPU reservation
-w <workload>
CPU -- cpu-bound execution trace (default)
CPU_SCAN <size> -- cpu-bound and also scans
an array that is size KB long
CPU_YIELD <amount> -- yield the processor
after getting <amount> of CPU time
CPU_SCAN_YIELD <size> <amount> -- combine
scanning and yielding behaviors
PERIODIC <amount> <period> -- periodic
task, make execution trace
CPU_PERIODIC <amount> <period> -- cpu-bound
periodic (see documentation)
LAT <period> -- dispatch latency test
-i <timer>
NATIVE -- native Unix or Windows timers
(default)
HR -- high resolution timers (if available:
requires a patched kernel)
RTC -- timers based on Linux real-time
clock (if available)
MM -- multimedia timers (Win32 only)
-s <time> -- start running this thread <time>
units into the Hourglass run
-f <time> -- terminate this thread <time>
units into the Hourglass run
```

Times are self-describing; valid examples are:
3m, 1.5s, 1020ms, 87.0us.