Appeared in *Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 141–148, Taipei, Taiwan, May 30–June 1 2001.

Augmented CPU Reservations: Towards Predictable Execution on General-Purpose Operating Systems

John Regehr John A. Stankovic
Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, VA 22903-2242, USA
john@regehr.org stankovic@cs.virginia.edu

Abstract

One problem with performing soft real-time computations on general-purpose operating systems is that these OSs may spend significant amounts of time in the kernel instead of performing work on behalf of the application that is nominally scheduled: the OS effectively steals time from the running application. Stolen time can be a significant obstacle to predictable program execution on real-time versions of Linux and Windows 2000, where it can cause applications to miss essentially all of their deadlines. We propose augmented CPU reservations*, a novel mechanism for using fine-grained accounting information about the amount of stolen time to help the scheduler allow applications to meet their deadlines. We have designed and implemented Rez-C and Rez-FB, two schedulers that provide augmented reservations, and we have tested them in Windows 2000, showing that they can increase the predictability of CPU reservations. We also experimentally quantify the severity of stolen time caused by a variety of devices such as hard disk controllers, a network interface, and a software modem under real-time versions of Windows 2000 and Linux.*

1. Introduction

Soft real-time applications with periodic CPU requirements are becoming an important part of the mix of programs that users run on general-purpose operating systems such as Linux and Windows 2000. These applications include simulation-based games, streaming audio and video, voice recognition, and hardware requiring real-time response from the OS. They are typically large and complex, they cannot be easily decomposed into real-time and non-real-time components, and they usually require the full services of a general-purpose operating system. Therefore, our work focuses on systems that predictably schedule user-level applications by modifying the OS scheduler [3, 7, 10], as opposed to systems that run a small hard real-time kernel "underneath" a general-purpose OS [2, 13].

The problem our work addresses is that low-level system activity in a general-purpose operating system can adversely affect the predictable scheduling of real-time applications. In effect, the OS *steals* time from the application that is currently scheduled.

Figures 1 and 2 illustrate the effects of stolen time. Each figure shows an actual execution trace of a CPU reservation that was granted by Rez, a new real-time scheduler we have developed for Windows 2000. In both figures Thread 2 has been granted a CPU reservation of 4 ms / 20 ms, meaning that it is guaranteed to be scheduled for 4 ms during every 20 ms interval. In the center of each figure is a single 4 ms block of time that Rez allocated to Thread 2. Each millisecond appears to be divided into roughly four pieces because the clock interrupt handler periodically steals time from the running application—we ran the clock at 4096 Hz to approximate precisely scheduled interrupts, which are not available in Windows 2000.

In Figure 1 our test application has the machine to itself, and Thread 1 uses up all CPU time not reserved by Thread 2. In Figure 2, the test application is running concurrently with an instance of Netperf, a network bandwidth measurement application that is receiving data from another machine over 100 Mbps Ethernet. Thread 1, which is running at low priority, gets less CPU time than it did in Figure 1 because it is sharing the processor with Netperf. However, Netperf does not get to run during the block of time reserved for Thread 2. There are gaps in the CPU time received by Thread 2 because the machine continues to receive data even when the Netperf application is not being scheduled, and the kernel steals time from Thread 2 to process this data. This stolen time causes Thread 2 to receive only about 82% of the CPU time that it reserved. A real-time application running under these circumstances will have difficulty meeting its deadlines.

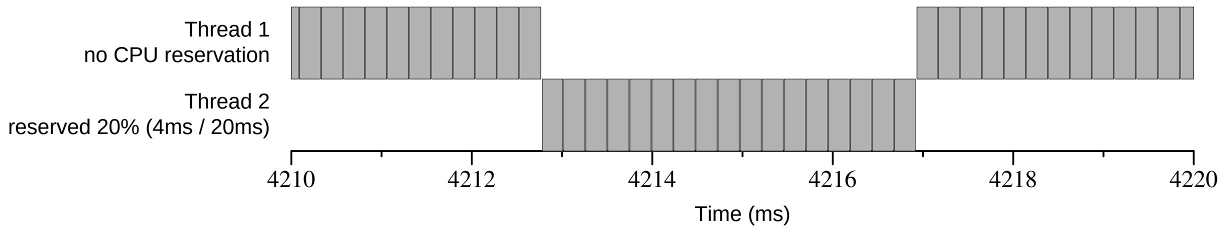

Figure 1. Execution trace of a CPU reservation functioning correctly on an otherwise idle machine.

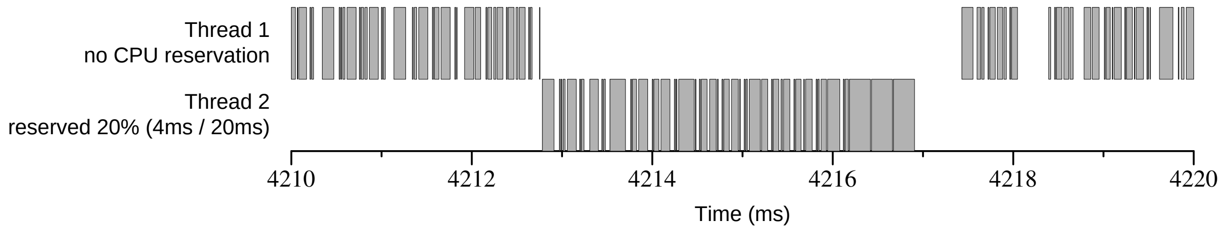

Figure 2. Gaps in Thread 2's execution indicate time being stolen from a CPU reservation.

The root of the problem is that although Rez ensures that Thread 2 will be scheduled for a 4 ms interval (or for several intervals adding up to 4 ms) during each 20 ms period, it is not necessarily the case that Thread 2 will get to execute for the entire 4 ms—stolen time is invisible to the thread scheduler. To address this problem we have designed and implemented two novel schedulers, Rez-C and Rez-FB, that provide *augmented CPU reservations* by actively measuring stolen time and counteracting its effects, permitting deadlines to be met even when the OS steals a significant amount of CPU time from real-time applications.

In Section 2 we present background material. Sections 3 and 4 describe the design, implementation, and evaluation of Rez-C and Rez-FB.

Section 5 strengthens the case for augmented CPU reservations by presenting the results of a study that quantifies the amount of time that can be stolen by a number of different device drivers in real-time versions of Linux and Windows 2000. We learned, for example, that the default configuration of the IDE disk driver in Linux can steal close to 50% of a CPU reservation.

2. Background

2.1. Characterization of Stolen Time

We define *stolen time* to be CPU time spent doing something other than running the currently scheduled application or performing services on its behalf. Time is stolen because *bottom-half device driver processing* in general-purpose operating systems is assigned a statically higher priority than any application processing, and because this time is not accounted for properly: it is "charged" to the application that happens to be running when a device needs service.

Bottom-half processing occurs in the context of interrupts and *deferred procedure calls*. Interrupts are hardware-supported high-priority routines invoked when an external device requests service. Deferred procedure calls (DPCs), which are analogous to *bottom-half handlers* in Linux and other Unix variants, were designed to give device drivers and other parts of the kernel access to high-priority, lightweight asynchronous processing outside of interrupt context [11, pp. 107–111].

In effect, Windows 2000 and similarly structured operating systems contain not one but three schedulers. Interrupts take precedence over all other processing and are scheduled by a fixed-priority preemptive scheduler that is implemented in hardware. DPCs take precedence over all thread processing and are scheduled by a simple FIFO scheduler. Finally, applications are scheduled at the lowest priority by the OS thread scheduler.

2.2. Rez: A Reservation Scheduler

Rez is a new scheduler that we have implemented; it provides CPU reservations to Windows 2000 threads. That is, it allows threads to run at an application-specified rate and granularity. For example, a thread that was granted a reservation of 0.75 ms / 8 ms would be guaranteed to run for 0.75 ms during every 8 ms period. Rez supports a wide range of reservation periods, between 1 ms and 1 s.

The Rez scheduling algorithm is similar to a number of other reservation schedulers that have been described in the literature, such as the *constant utilization server* developed by Deng et al. [3], the *constant bandwidth server* that Abeni and Buttazzo developed [1], and the Atropos scheduler developed for the Nemesis OS [9]. Rez assigns a budget to each thread that has a CPU reservation. Budgets are decremented in proportion to the CPU time allocated to the associated thread, and are replenished at the beginning of each period. Rez always schedules the thread that has the earliest deadline among all threads that are runnable and have a positive budget. The deadline for each thread is always taken to be the end of its current period.

Windows 2000 + Rez refers to the system composed of Windows 2000 and our modified scheduler. This system suffers from reduced predictability because bottom-half mechanisms in Windows 2000 can steal time from real-time applications. Our approach to improving predictability is called *augmented reservations*. The basis of this approach is to give the reservation scheduler access to fine-grained accounting information about how long the kernel spends running DPCs, allowing it to react accordingly. To this end we implemented two additional versions of the Rez scheduler called Rez-C and Rez-FB.

3. Coping with Stolen Time

Once a real-time scheduling abstraction such as CPU reservations has been implemented within a general-purpose operating system, the ways to increase predictability with respect to stolen time form a continuum:

1. To avoid further modifications to the core OS, but to manually move device driver code out of time-stealing bottom-half contexts.

2. To instrument stolen time and feed the resulting information into the real-time scheduler to allow it to compensate.

3. To modify bottom-half mechanisms to put them under control of the scheduler, eliminating this source of stolen time.

The first option has been recently explored by Jones and Saroiu [8] in the context of a software modem driver. Jeffay et al. [5] chose the third option: they modified FreeBSD to perform proportional-share scheduling of network protocol processing. Our work is based on the second option.

3.1. Measuring Stolen Time

Of the two sources of stolen time in Windows 2000, interrupts and DPCs, we have instrumented only DPCs. Although it would be straightforward to instrument hardware interrupt handlers as well, the time spent in DPCs serves as a useful approximation of the true amount of stolen time because interrupt handlers in Windows 2000 were designed to run very quickly: they perform critical processing and then enqueue a DPC to perform any additional work.

To instrument DPCs we added code to the beginning and end of the dispatch interrupt handler (a software interrupt handler that dequeues and runs DPCs) to query the Pentium timestamp counter (using the `rdtsc` instruction) which returns the number of cycles since the machine was turned on. By taking the difference between these values the system can accurately keep track of the amount of time spent running DPCs.

The interface to stolen time accounting is the `GetStolen()` function, which is available to code running in the Windows 2000 kernel; it returns the amount of stolen time since it was last called.

3.2. Rez-C: Increasing Predictability by Catching Up

Rez-C gives threads the opportunity to catch up when they have had time stolen from them. It does this by deducting from budgets only the actual amount of CPU time that threads have received, rather than deducting the length of the time interval that they were nominally scheduled for, which may include stolen time. For example, if a thread with a reservation of 4 ms / 20 ms is runnable and will have the earliest deadline during the next 4 ms, Rez-C schedules the thread and arranges to regain control of the CPU in 4 ms using a timer. When the timer expires, Rez-C checks how much time was stolen during the 4 ms interval using the `GetStolen()` function. If 1.2 ms were stolen, then Rez-C deducts 2.8 ms from the thread's budget. If the thread still has the earliest deadline, Rez-C arranges to wake itself up in 1.2 ms and allows the thread to keep running.

Rez-C uses accounting information about stolen time at a low level, bypassing the admission controller. When there is not sufficient slack in the schedule, allowing a thread to catch up may cause other threads with reservations to miss their deadlines or applications in the timesharing class to be starved. These deficiencies motivated us to design Rez-FB.

3.3. Rez-FB: Increasing Predictability using Feedback Control

Our second strategy for coping with stolen time assumes that the amount of stolen time in the near future will be similar to what it was in the recent past. It uses a feedback loop to minimize the difference between the amount of CPU time that each application attempted to reserve and the amount of CPU time that it actually receives. There is an instance of the feedback controller for each thread that has a CPU reservation. The parameters and variables used by the feedback controller are:

- A set point R, the amount of CPU time that an application requested.
- A control variable C, the amount of CPU time reserved by Rez-FB on behalf of an application.
- A process variable P, the amount of CPU time that an application actually receives; this is calculated by summing the lengths of the time intervals during which the application was scheduled and subtracting the amount of time stolen during those intervals.
- A constant gain $G \leq 1$.

The feedback equation, which is evaluated for each reservation each time its period starts, is:

$$C_t = C_{t-1} + G(R - P_{t-1})$$

In other words, at the beginning of a reservation's period the amount of CPU time requested by Rez-FB on behalf of the application is adjusted to compensate for the difference between the desired and actual amounts of CPU time during the previous period. The gain helps prevent overshooting and can be used to change how aggressively the controller reacts to changing amounts of stolen time.

Because Rez-FB applies accounting information to reservation amounts rather than budgets, it does not bypass the admission controller. Therefore, Rez-FB will not allow threads with CPU reservations to interfere with each other, or with time-sharing applications. On the other hand, Rez-FB reacts more slowly to stolen time than Rez-C, potentially causing applications to miss more deadlines when the amount of stolen time varies on a short time scale. The feedback controller currently used by Rez-FB is a special case of a PID (Proportional-Integral-Derivative) controller that contains only the integral term. In the future we may attempt to improve Rez-FB's response to changes in the amount of stolen time using more sophisticated controllers that have proportional and derivative terms as well.

4. Experimental Evaluation of Rez-C and Rez-FB

We now evaluate and compare Rez-C and Rez-FB according to the following criteria: how well do augmented reservations help applications meet their deadlines under adverse conditions, and how efficient are they in terms of run-time overhead?

The application scenarios that we will consider include the following elements: a general-purpose operating system, Windows 2000, that has been extended with Rez (as a control), Rez-C, or Rez-FB; a soft real-time application that uses a reservation to meet its periodic CPU requirements; and a background workload that causes the OS to steal time from the real-time application.

Unless otherwise stated, all data points in this section and in Section 5 are averages of ten 10-second runs. Confidence intervals are calculated at 95%.

4.1. Test Application

The soft real-time application used in our experiments is a synthetic test application. The important qualities of this application are: the ability to create multiple threads at different priorities with optional CPU reservations; the ability to detect stolen time by measuring exactly when these threads are scheduled; and for threads with reservations, the ability to determine if and when deadlines are missed.

To this end we started with a test application that was originally developed for the Rialto operating system at Microsoft Research and later ported to Rialto/NT [6]. For the work reported here we ported it to TimeSys Linux/RT and Windows 2000 + Rez, and gave it the capacity to detect deadline misses for threads with ongoing CPU reservations.

Rather than using the CPU accounting provided by the operating system[1] our test application repeatedly polls the Pentium timestamp counter. When the difference between two successive reads is longer than 2.2 μs, time is assumed to have been stolen from the application between the two reads. This number was experimentally determined to be significantly longer than the usual time between successive reads of the timestamp counter and significantly shorter than common stolen time intervals. The duration of the stolen time interval is taken to be the difference between the two timer values minus the average time spent on the code path between timer reads (550 ns). We believe that this provides a highly accurate measure of the actual CPU time received by the application.

Threads with reservations repeatedly check if the amount of wall clock time that has elapsed since the reservation was granted has passed a multiple of the reservation period. Each time this happens (that is, each time a period ends) they register a deadline hit if at least the reserved amount of CPU time has been received during that period, or a deadline miss otherwise.

4.2. Test Workload and Platform

The workload used is an incoming TCP stream over 100 Mbps Ethernet (we characterize other sources of stolen time in Section 5). We chose this workload because it is

[1] The statistical accounting performed by Windows 2000 and Linux is particularly inappropriate for monitoring the usage of threads that have CPU reservations: since accounting is periodic, the periods of reservations and accounting can resonate, causing threads to be drastically over- or under-charged for their CPU usage.

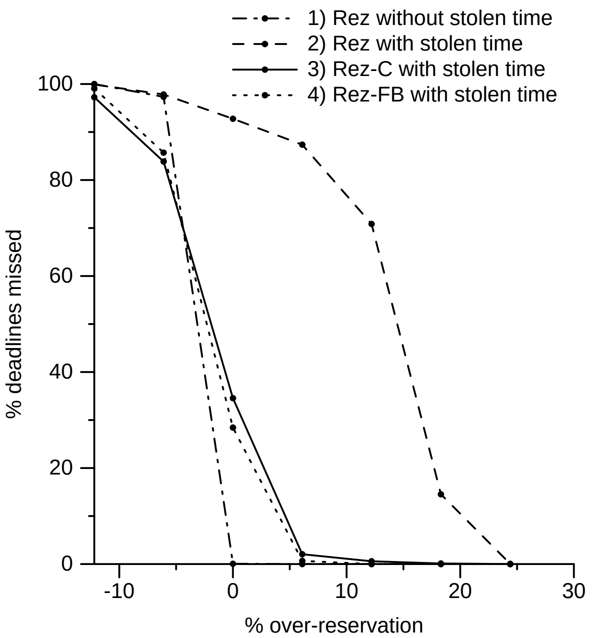

Figure 3. Predictability of Rez, Rez-C, and Rez-FB.

entirely plausible that a desktop computer may be in the middle of a real-time task such as playing a game or performing voice recognition when high-bandwidth data (such as a file transfer) arrives over a home or office network.

To actually transfer data we used the default mode of Netperf [4] version 2.1, which establishes a TCP connection between two machines and transfers data over it as rapidly as possible.

Our test machine was a dual 500 MHz Pentium III with 256 MB of RAM. It ran in uniprocessor mode for all experiments. It was connected to the network using an Intel EtherExpress Pro/100B PCI Ethernet adapter. For all experiments in this section the machine ran one of our modified Windows 2000 kernels, and had a timer interrupt period (and therefore a minimum enforceable scheduling granularity) of 244 μs.

4.3. Reducing Deadline Misses using Rez-C and Rez-FB

Figure 3 shows the number of deadline misses detected by our test application with a reservation of 4 ms / 20 ms under four different conditions:

1. Scheduled by Rez, on an otherwise idle machine.
2. Scheduled by Rez, while the test machine received a TCP stream over 100 Mbps Ethernet.
3. Scheduled by Rez-C, while the test machine received a TCP stream over 100 Mbps Ethernet.
4. Scheduled by Rez-FB, while the test machine received a TCP stream over 100 Mbps Ethernet.

To meet each deadline, the test application needed to receive 4 ms of CPU time during a 20 ms period. In order to demonstrate the effect of statically over-reserving as a hedge against stolen time, for each of the four conditions we had Rez actually reserve more or less than the 4 ms that was requested. So, if Rez were set to over-reserve by 50%, it would actually reserve 6 ms / 20 ms when requested to reserve 4 ms / 20 ms.

Line 1 shows that on an idle machine, any amount of under-reservation will cause most deadlines to be missed, and that no deadlines are missed by the test application when it reserves at least the required amount of CPU time. This control shows that Rez is implementing CPU reservations correctly.

Line 2 (the rightmost line on the graph) illustrates the effect of time stolen by network receive processing. To avoid missing deadlines, Rez must over-reserve by about 24%. This is quite a large amount, and would not prevent the application from missing deadlines in the case that several drivers steal time simultaneously.

Lines 3 and 4 are very similar, and show that both Rez-C and Rez-FB increase the predictability of CPU reservations when the OS is stealing time from applications. Notice that a small amount of over-reservation (about 6%) is required before the percentage of missed deadlines goes to nearly zero. Some deadlines are missed in the 0% over-reservation case because we only instrumented DPCs, and not hardware interrupt handlers.

We calculated confidence intervals for the data points in Figure 3 but omitted them from the graph because they were visually distracting. The 95% confidence interval was always within 4% of the means reported here, and was much closer than that for most data points.

4.4. Response of Rez-FB to Variations in Load

Rez-C is very simple and retains no information about stolen time between periods of a reservation. However, as we described in Section 3.3, Rez-FB uses a feedback loop that compares its current performance to performance in the previous period. This raises questions about stability, overshoot, and reaction time. The intuition behind the feedback loop is straightforward and we believe that Rez-FB is stable and predictable, and that it will quickly converge on the correct amount of CPU time to allocate. However, we have performed an experiment under changing load in order to test this hypothesis. Figure 4 shows the response of Rez-FB to a 1-second burst of network traffic, which arrives between times 1000 and 2000. The test application has created a single thread with a CPU reservation of 4 ms / 20 ms. So, $R = 4.0$ ms. This graph contains no confidence intervals since

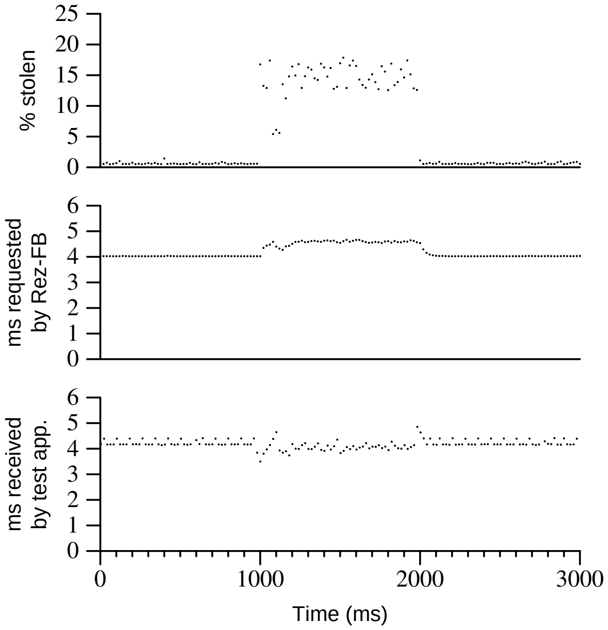

Figure 4. Performance of Rez-FB under changing load.

the data come from a single execution. Each point on the graph represents a single 20 ms period.

The top graph in Figure 4 shows the amount of stolen time reported to Rez-FB by each call to `GetStolen()`, expressed as a percentage of 4 ms. It shows that there is a small base amount of time stolen by background system activity, and that stolen time increases dramatically during network activity. This is exactly what we would expect. The middle graph shows C_t, the amount of CPU time requested by Rez-FB during each period. As expected, Rez-FB requests more CPU time when more time is being stolen by the kernel. Although the amount of stolen time at the start of Netperf's run is noisy (probably due to TCP slow start and other factors), its finish is abrupt and around time 2000 C_t drops from its elevated level back to about 4.0, cushioned by the gain G. Although we tried several different values for G, all experiments reported here used a value of 0.5. We did not look into the sensitivity of this parameter in detail, but values between 0.5 and 1 appeared to produce about the same number of missed deadlines.

The bottom graph shows P_t, the actual amount of CPU time received by our test application during each period. While the machine is quiet (in ranges 0-1000 and 2000-3000) the values of P_t are quantized because the scheduling enforcement granularity is limited to 244 μs. A deadline is missed whenever P_t is below 4.0; this happened 22 times during the test. This is due to unaccounted stolen time from the network interrupt handler and also to lag in the feedback loop. While missing 22 deadlines may be a problem for some applications, this is significantly better than missing most of the 500 deadlines between times 1000 and 2000, as would have happened with Rez under the same conditions. To support applications that cannot tolerate a few missed deadlines, the system would need to instrument stolen time more comprehensively or statically over-reserve by a small amount.

4.5. Run-Time Overhead of Rez-C and Rez-FB

Both Rez-C and Rez-FB add very little overhead to the scheduler. The overhead of instrumenting DPCs (incurred each time the kernel drains the DPC queue) is twice the time taken to read the Pentium timestamp counter and write its result to memory, plus the time taken by a few arithmetic operations. Similarly, the overhead of the `GetStolen()` call is the time taken to run a handful of instructions.

To verify that the augmented reservation schedulers add little overhead we measured how much CPU time was lost to a reservation running under Rez-C and Rez-FB as compared to the basic Windows 2000 + Rez. This revealed that Rez-C adds 0.012%±0.0028 overhead and Rez-FB adds 0.017%±0.0024 overhead. We would have been tempted to assume that these differences were noise but the confidence intervals indicate that they are robust.

5. Stolen Time in Other Systems and by Other Devices

In this section we provide additional motivation for augmented CPU reservations by presenting the results of a study of the amount of time that can be stolen by the Linux and Windows 2000 kernels when they receive asynchronous data from a number of different external devices. For the Linux tests we used TimeSys Linux/RT [12], which adds resource kernel functionality [10] such as CPU reservations and precise timer interrupts to Linux.

Our goal was not to compare the real-time performance of Linux/RT and Windows 2000 + Rez, but rather to shed light on the phenomenon of stolen time and to find out how much time can be stolen by the drivers for various devices on two completely different operating systems. Indeed, these results will generalize to any operating system that processes asynchronous data in high-priority, bottom-half contexts without proper accounting. As far as we know, these are the first published results of this type.

For Linux tests we used TimeSys Linux/RT version 1.1A, which is based on version 2.2.14 of the Linux kernel. All Linux tests ran on the same machine that ran all of our Windows 2000 tests (a dual 500 MHz Pentium III booted in uniprocessor mode).

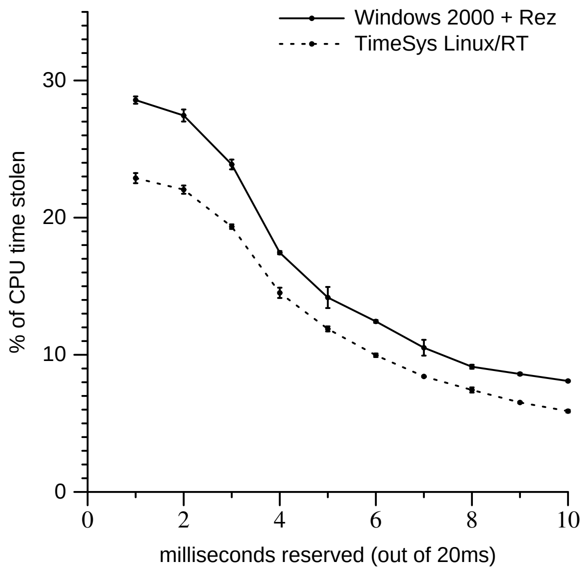

Figure 5. Time stolen by the kernel to process an incoming TCP stream.

5.1. Effect of Reservation Amount on Time-Stealing by Network Processing

Figure 5 shows how the amount of time stolen from a CPU reservation by network receive processing changes with the amount of the reservation. The test application reserved between 1 ms and 10 ms out of 20 ms. The reason that the proportion of stolen time decreases as the size of the block of reserved time increases can be seen by looking closely at Figure 2: towards the end of the reserved block of time (after time 4216) there is little stolen time. This is because the Netperf application does not get to run during time reserved by the real-time application; therefore, kernel network buffers are not drained and packets are not acked, causing the sender to stop sending after a few milliseconds.

5.2. Hard Disk Controllers

Table 1 shows the amount of time that was stolen from CPU reservations of 4 ms / 20 ms by OS kernels as they processed data coming from hard disks. We show measurements for both Linux/RT and Windows 2000 + Rez, for a SCSI and an IDE disk, and using both direct memory access (DMA) and programmed I/O (PIO) to move data to the host, when possible.

The SCSI disk used in this test is a Seagate Barracuda 36, connected to the host over an Ultra2 SCSI bus; it can sustain a bandwidth of 18.5 MB/s while reading large files. The IDE disk is an older model (a Seagate 1270SL) that can sustain a bandwidth of only about 2.35 MB/s.

From this table we conclude that disk transfers that use DMA cause the OS to steal only a small amount of time

OS	disk / driver	% time stolen
Windows 2000 + Rez	IDE / DMA	0.78±0.052
	IDE / PIO	n/a
	SCSI / DMA	0.55±0.026
	SCSI / PIO	n/a
Linux/RT	IDE / DMA	1.1 ±0.28
	IDE / PIO	49.0 ±3.5
	SCSI / DMA	0.74±0.20
	SCSI / PIO	n/a

Table 1. Amount of time stolen from a CPU reservation by disk device drivers.

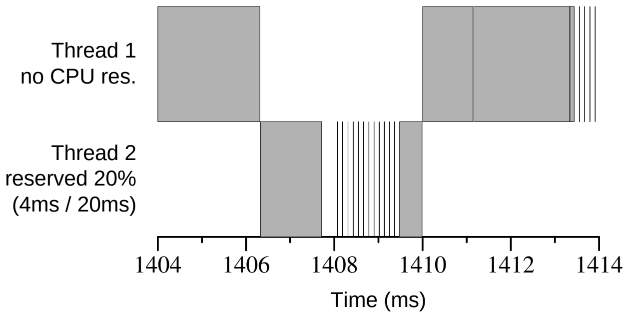

Figure 6. Time stolen from a Linux/RT CPU reservation by the IDE disk driver.

from real-time applications (the confidence intervals indicate that the differences, though small, are real). However, even a slow disk can severely hurt real-time performance if its driver uses PIO: in our test it caused the Linux kernel to steal nearly half of the CPU time that a real-time application reserved. Therefore, it is imperative that real-time systems avoid using PIO-based drivers for medium- and high-speed devices. The large gap in Thread 2's block of reserved time in Figure 6 illustrates the problem.

Unfortunately, Linux distributions continue to ship with PIO as the default data transfer mode for IDE disks. For example, we recently installed Redhat Linux 7.0 on a new machine equipped with a high-performance IDE disk. Due to overhead of PIO transfers, the machine was barely usable while large files were being read from the hard disk. Interactive performance improved dramatically when we turned on DMA for the hard disk using the `hdparm` command. Windows 2000 uses DMA by default for both SCSI and IDE disks.

5.3. Software-Based Modems

Software-based modems contain minimal hardware support: they perform all signal processing in software on the main CPU. Connecting a software modem at 45333 bps (the highest our local lines would support) caused Windows 2000 + Rez to steal 9.87%±0.15 from a CPU reservation of 4 ms / 20 ms. We did not test the soft modem under Linux because driver support was not available.

As Jones and Saroiu [8] mention, software-based implementations of Digital Subscriber Line (DSL) will require large amounts of CPU time: 25% or more of a 600 MHz Pentium III. Obviously a soft DSL driver will steal significant amounts of CPU time from applications if its signal processing is performed in a bottom-half context.

5.4. USB Ports

While retrieving a large file over USB (the file was stored on a CompactFlash memory card), a reservation of 4 ms / 20 ms under Windows 2000 + Rez had 5.7%±0.032 of its reservation stolen. USB could become a much more serious source of stolen time in the future as USB 2.0 becomes popular—it is 40 times faster than USB 1.1 (480 Mbps instead of 12 Mbps). Finally, while we did not test Firewire devices, at 400 Mbps it is a potentially serious source of stolen time.

6. Conclusion

Our contributions have been:

- To design, implement, and evaluate two novel schedulers that provide *augmented CPU reservations*, increasing predictability when time is stolen from applications that they schedule. We believe that Rez-C and Rez-FB provide large gains in predictability for small additional cost.
- To quantify the amount of time that can be stolen by the drivers for a number of common devices on Linux and Windows 2000. We believe that the results of this study will be useful in a variety of situations: it should serve as a warning to people developing real-time applications that may run on busy machines, it could help developers who are moving code from bottom-half contexts into threads decide which drivers should be targeted first, and it should aid the intuitions of system designers who are deciding what kinds of system services to put in schedulable contexts.

Acknowledgments

The authors would like to thank Marty Humphrey, Mike Jones, Chenyang Lu, and Stefan Saroiu for their helpful comments on drafts of this paper. This work was funded, in part, by Microsoft Research and by NSF grant number NSF CCR-9901706.

References

[1] Luca Abeni and Giorgio Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, December 1998.

[2] Gregory Bollella and Kevin Jeffay. Support For Real-Time Computing Within General Purpose Operating Systems: Supporting Co-Resident Operating Systems. In *Proc. of the 1st IEEE Real-Time Technology and Applications Symposium*, pages 4–14, Chicago, IL, May 1995.

[3] Zhong Deng, Jane W.-S. Liu, Lynn Zhang, Seri Mouna, and Alban Frei. An Open Environment for Real-Time Applications. *Real-Time Systems Journal*, 16(2/3):165–185, May 1999.

[4] Hewlett-Packard Company Information Networks Division. Netperf: A Network Performance Benchmark, 1995. `http://www.netperf.org/netperf/training/Netperf.html`.

[5] Kevin Jeffay, F. Donelson Smith, Arun Moorthy, and James Anderson. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, pages 480–491, Madrid, Spain, December 1998.

[6] 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.

[7] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211, Saint-Malô, France, October 1997.

[8] 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.

[9] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[10] Shuichi Oikawa and Ragunathan Rajkumar. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proc. of the 5th IEEE Real-Time Technology and Applications Symposium*, pages 111–120, Vancouver, BC, Canada, June 1999.

[11] David A. Solomon and Mark E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.

[12] TimeSys. The Concise Handbook of Linux for Embedded Real-Time Systems, 2000. `ftp://ftp.timesys.com/pub/docs/LinuxRTHandbook.pdf`.

[13] Victor Yodaiken. The RTLinux Manifesto. In *Proc. of The 5th Linux Expo*, Raleigh, NC, March 1999.