Better Real-time Response for Time-share Scheduling

Scott A. Banachowski and Scott A. Brandt
Computer Science Department
University of California, Santa Cruz
{sbanacho,sbrandt}@cse.ucsc.edu

Abstract

Time-share CPU schedulers permeate general-purpose computer systems, yet provide little support for real-time constraints. We demonstrate that by making inferences from the behavior of applications, soft real-time scheduling performance is achievable using a best-effort policy in which the scheduler has no a priori knowledge of application resource needs. In a typical time-share scheduler, recent CPU usage is accounted for in a dynamic scheduling priority, so the behavior pattern of a process implicitly impacts the timeliness of future allocations. To improve the performance of latency-sensitive processes, we developed a CPU scheduler that explicitly uses past behavior to make short-term scheduling decisions, while still preserving the long-term goal of fairness. In this paper, we show that this scheduler, called BeRate, outperforms Linux when scheduling workloads that contain applications with periodic deadlines.

1 Introduction

Modern computer systems of all types are growing in complexity and it is desirable for such systems to concurrently manage combinations of non-real-time, soft real-time, and hard real-time processes. Many general-purpose operating systems use CPU schedulers adapted from time-share systems. Time-share schedulers implement a *best-effort* policy. As the term "best-effort" implies, the scheduler provides no facilities for meeting specific performance guarantees. As a result, processes with temporal constraints such as real-time and multimedia applications, may not receive timely allocation of CPU required to meet deadlines. This paper discusses a system that serves soft real-time and non-real-time processes using a best-effort policy.

Best-effort policies are attractive due to their simplicity and ease of use; applications do not require system interfaces for reserving CPU bandwidth, and the scheduler need not incorporate admission control or service guarantees. Although not suitable for hard real-time platforms where missed deadlines equate to system failure, best-effort policies may suit soft real-time applications that allow degraded performance. Recognizing that best-effort scheduling is a desirable policy for general-purpose systems and that soft real-time applications are becoming ubiquitous on these systems, our research aims to improve soft real-time performance using time-share schedulers.

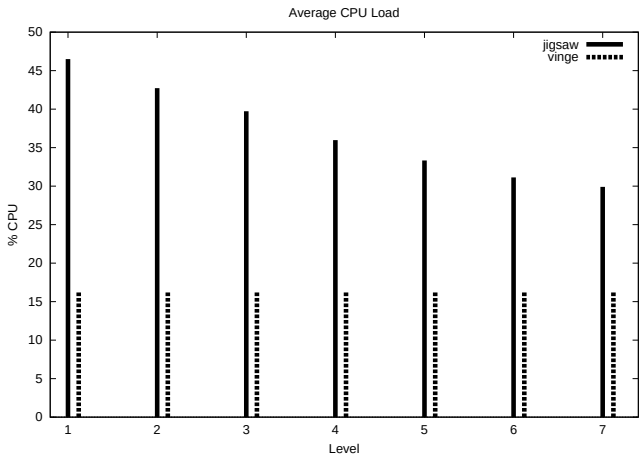
The Best-effort Rate CPU scheduler (**BeRate**) enhances performance for soft real-time applications while providing adequate progress and response to all applications. The scheduler provides periodic applications with better latency response, while preserving the behavior of traditional time-sharing schedulers for non-periodic processes.

2 Background and Motivation

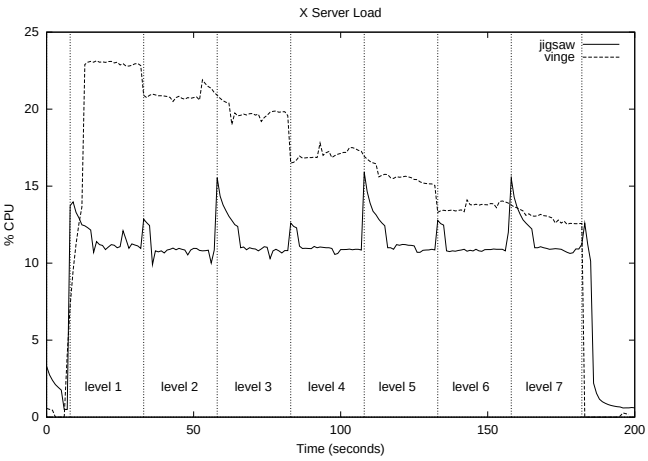
Time-share scheduling algorithms were not designed for periodic deadline processing, and without service guarantees the performance of real-time applications degrade in the presence of scheduling latency caused by concurrent execution of applications. We wish to improve the responsiveness of best-effort schedulers when serving workloads containing periodic deadlines. Previous research on the DQM system [7] demonstrates that it is possible to robustly execute soft real-time applications on best-effort systems. DQM allows applications to dynamically adjust their resource usage to available resources. By adjusting demand so that a set of applications uses less than 100% of resources, a best-effort scheduler is able to provide reasonable soft real-time performance.

2.1 Process resource characterization

DQM and other soft real-time systems support scheduling with deadline constraints, but lose a primary advantage of the best-effort model by requiring *a priori* specification of application resource needs. The interface to the scheduler is exposed; either programmers or users must negotiate



(a) CPU load as a function of QoS level.



(b) X server load during cycles through QoS level.

Figure 1. CPU load incurred by quality levels of a video playback, collected on two different systems.

with the scheduler to control scheduling policies. The programming or run-time model lacks generality, and restricts the portability of applications.

It is difficult to characterize resource needs, because applications perform inconsistently on different systems. To demonstrate this, we use a video player capable of changing its CPU load by adjusting image quality. We measured the load incurred by seven discrete levels of descending quality. Figure 1(a) shows the average load per level. On the system *jigsaw*, higher levels correspond to decreased load, but on *vinge* load remains constant across levels even though quality is diminished. This is explained by observing the X window server application: its CPU usage is plotted in Figure 1(b). On *jigsaw* the X server load remains relatively constant (with brief load peaks at level transitions). On *vinge*, changing the video quality effects X's load; on this system the player offloads its image processing to the X server's video driver. If we had predicted the CPU requirements of the application on one platform, it wouldn't scale in the same manner on another system, with the additional unexpected side-effect on another application.

A contribution of our research is to eliminate the difficulty associated with resource characterization by using online detection of timeliness requirements. If resource demands are determined online as a process executes, it is not essential for usage or deadline requirements to be communicated to the scheduler. Because most soft real-time processes use operating system primitives for synchronization, the rate at which they enter runnable state is observable by the kernel. BEST [2] is a best-effort Linux scheduler that improves performance for soft-real time processes by making the assumption that applications with periodic deadlines enter a *runnable state* when they begin a periodic computation. By observing when a process becomes runnable we may infer its period. By assuming a predicted period is a deadline, and allocating in order of earliest deadline, processes with periodic deadlines receive timely allocation of resources.

BEST is effective but violates fairness, so BeRate was developed to overcome this shortcoming. The assumption that resources be divided equally among processes is implicit in time-sharing schedulers. Because BEST deems periodic processes most important, user-assigned priorities are ignored. During overload the absence of fairness leads to instability or inability to provide adequate performance to other important processes [3]. Even worse, if a non-periodic process is more important than a periodic process, there is no way for BEST to enforce it. BeRate uses techniques similar to BEST, but instead of explicitly measuring deadline, it determines the rate a process recently consumed CPU, and predicts a deadline that allows the process to proceed within its allocated fair-share.

2.2 Time-share scheduling anomalies

Time-share schedulers allocate resources so that multiple tasks appear to execute simultaneously. The scheduler attempts to provide fast response time for latency-sensitive processes, while maintaining fairness of CPU allocation over long-term. To improve the responsiveness, schedulers estimate processes' recent CPU usage, and assign I/O-bound processes a higher, but short-lived, dynamic

priority. BSD uses multi-level feedback queues [19], and Linux mimics this behavior, although dynamic priority calculations differ [6].

Although a goal of time-share systems is low response latency, it does not mean that applications will meet deadlines. In the Linux or BSD scheduler, the performance of a process with periodic deadlines is impacted by the phasing of the period with the system clock (since the kernels do accounting during the system clock interrupt) and the phasing of quanta execution in relation to other processes. A process with deadlines will typically block between computations, appearing I/O-bound, and receiving a higher priority. Nevertheless, the scheduler does not always assign CPU in time for deadlines to be met. During a clock interrupt, charging a tick to a process may reduce its short-term priority, causing lower responsiveness in a subsequent period. The opposite effect, where a process is rarely billed for CPU use is also possible. Etsion [11] found a case where a periodic task in Linux is billed for only 2% of its CPU consumption. In experiments with BSD, we found situations where a task with short periodic deadlines consumed 3 times its share of CPU because its use was under-accounted.

Increasing the frequency the kernel gathers statistics helps alleviate sampling problems, but does not entirely solve scheduler latency, as soft real-time processes still miss deadlines when they should be able to make them; we observe this in our experiments of Section 5. Our solution is to change the scheduling algorithm: instead of using clock-driven samples to characterize a process, we explicitly measure behavior when it awakens. A BeRate scheduler measures the rates of processes, dynamic priority becomes a function of both rate and period. Using this technique allows soft real-time processes to meet deadlines, while preserving the default behavior of time-share schedulers.

3 Related Work

The goal of BeRate is to improve latency of soft real-time scheduling in time-share environments. Some time-share systems provide hard real-time capability [15, 30] by allowing real-time tasks to run with high static priority, and assigning other tasks remaining bandwidth. However, using static real-time priorities is inadequate for handling continuous sound or video [22]; it causes pathologies due to unfairness (lower priority processes may never make progress), and it requires workload-dependent tuning which is difficult, especially when the workload is as dynamic and unpredictable in desktop systems.

Researchers use several hierarchical scheduling techniques to adapting multi-level scheduling to soft real-time systems [8, 9, 12, 13, 16, 24, 26]. The architectural approach of dividing schedulers into levels creates flexibility when running a mix of applications of differing processing needs; with it comes the problem of choosing ideal configurations, which as research indicates is not trivial. We do not introduce the complexity of multiple levels in BeRate. However, using the BeRate scheduler does not preclude integration into multi-level schemes.

Proportional-share schedulers assign processing bandwidth so that processes receive CPU within bounded rates [4, 10, 17, 20, 23, 27, 28, 29]. To meet deadlines, a proportional scheduler must know the rate requirements of processes. This information is usually fed to the scheduler through system APIs. However, it may be difficult to determine rate if the performance of the target processor is unknown [14]. The BeRate scheduler does not need to be informed of processes' rates, making the development and use of soft real-time applications easier. It uses techniques similar to proportional schedulers by generating deadlines from processes' allocated shares, by observing past execution patterns and inferring deadlines.

Several projects aim to reduce the latency of context switching in the Linux kernel. The low-latency patch reduces the size of uninterpretable execution paths inside the kernel by adding opportunities for preemption [21]. The preemptable Linux patch allows multiple threads to execute in the kernel simultaneously, so that preemption need not be disabled inside the kernel [18]. Both developments reduce latency in the kernel, and are important for supporting real-time applications. However, neither approach fixes latency caused by inappropriate scheduling decisions. BeRate is complementary to these techniques, reducing latency caused by the scheduler's decisions.

Scheduling latency may be reduced by increasing the system clock frequency [1]. In BeRate, we increased the timer resolution of Linux by a factor of 8. The default Linux clock is 100 Hz; for a video stream of 33 frames/second the average period is 3 ticks, so a measurement error of 1 tick is a significant percentage of its period. By increasing the timer resolution to 800 Hz, on-line measurements are finer grained, and provide a better estimate of application periods. The processing power of modern systems is able to tolerate this slight increase in overhead due to interrupt processing [11].

4 Implementation

To develop the BeRate scheduler we had a number of specific design criteria. Neither users nor developers need to provide any *a priori* information about processes. When processes do not miss deadlines, they have the opportunity to wait for the next period, allowing the kernel to measure usage and increasing the likelihood of consistent and detectable patterns. The default behavior of the BeRate reasonably conforms to time-sharing scheduler policies: in the long-term all processes receive a fair-share of resources (ad-

justable with *nice*), but in short-term favor I/O-bound over CPU-bound processes.

4.1 Linux scheduler overview

We implemented the BeRate scheduling algorithm in the Linux 2.4.9 kernel. A brief description of the unmodified Linux scheduler follows.

The function `schedule()` allocates the CPU to a process. It selects the process with highest dynamic priority from the runnable queue. The execution of `schedule()` is triggered two ways: explicitly when a running process is put to sleep, or upon return from an interrupt or trap. The function `goodness()` calculates dynamic priorities. The dynamic priority is the process's remaining time quantum, and decreases as the process executes. When all runnable processes consume their quantum, `schedule()` recomputes their dynamic priority using $pri = pri/2 + nice$, where *nice* is a positively scaled user-settable scheduling priority. At this time, a blocked process with a non-zero time quantum receives a priority boost, increasing its responsiveness in when it awakens.

Linux maps *nice* values of processes to execution quanta. With a workload of n CPU-bound processes, Linux executes each for its quantum duration in round-robin fashion. We call an epoch of one round-robin period the load L; the epoch lasts $L = \sum_{i \in n} q_i$ ticks, where each process i has quantum q_i. Each process receives a CPU share of q_i/L.

4.2 BeRate scheduler details

The BeRate scheduling algorithm is simple. Every process has a periodic deadline, and the `schedule()` function selects the runnable process with the earliest deadline. Since actual deadlines are unknown, a heuristic estimates deadlines for periodic processes and pseudo-deadlines for other processes. There is no guarantee that assigned deadlines are met, rather deadline is used for ordering and preemption. Conceptually, deadlines are assigned so that processes receive the same CPU share and scheduling quanta as in unmodified Linux. However, processes with timeliness constraints need to be allocated shares in frequent, shorter periods instead of longer quanta, so have shorter deadlines.

When BeRate sets a deadline for a process, it assigns a *deadline expiration* which decrements as the process executes. Two events trigger a new deadline to be computed: either its previous deadline expires, or the process wakes from blocking. When a deadline expires, if the newly computed deadline is no longer earliest, another process becomes eligible to run and is scheduled. By setting the expiration timer to the same value Linux uses for quanta, a CPU-bound process resets its deadline after every quanta of execution preserving Linux's notion of long-term fairness.

Assuming that jobs executions are shorter than periods and that processes sleep between jobs, soft real-time (SRT) processes consume CPU in series of short CPU bursts instead of longer, single quanta. An SRT process i does a sequence of periodic job computations, each job having a deadline d_i. The average job length is $\bar{e}$ (individual jobs may vary in length). In order to meet deadlines, the CPU utilization of an SRT process must be less than its fair share: $\bar{e}/d_i \leq q_i/L$.

A key component of BeRate's algorithm is the prediction of deadline d_i. BeRate does not know deadlines, but for each process it knows q_i (and therefore load L) and may measure $\bar{e}$ from past behavior. The scheduler records the number of ticks e_i consumed in each process's period, and averages it with previous measurements using $\bar{e} = (e_i + w \times \bar{e})/(1+w)$ (w is a constant weight factor). Using these values, deadline is estimated as $d_i = \frac{\bar{e}_i \times L}{q_i}$.

For a process that meets periodic deadlines within its share, $\bar{e}$ reflects its average job time, and its deadline estimate is a lower bound of its actual deadline. For a CPU-bound process, $\bar{e} = q_i$, so deadline becomes L, meaning it should complete quanta in the epoch of a round-robin sequence; when running a workload entirely consisting of CPU-bound processes, BeRate chooses the same schedule as Linux. For I/O bound processes, $\bar{e}$ depends on the rate and duration of CPU bursts, and will be $\leq q_i$, generally leading to deadlines earlier than CPU-bound processes, for improved latency response.

5 Experimental Results

We conducted experiments comparing the performance of the BeRate scheduler with that of the Linux scheduler. Our soft real-time workload is statistically driven, so we repeated each experiment until the percent of missed deadlines per run was known to the nearest 10th of a percent, with confidence interval of 95%. We experimented using both simulations of the Linux and BeRate algorithms, and the actual Linux and BeRate implementations.

The figures in the following section represent a single run from our simulations, whereas in our discussion we refer to the aggregate results of many runs. Table 1 summarizes all these results. The table shows the increased performance when raising the Linux system clock from 100 Hz to 800 Hz, which alleviates some latency problems discussed in Section 2.2. In the simulations, the Linux clock is also set to 800 Hz, and although the simulation and real implementation performance slightly differ, the relative performance is similar.

Two synthetic workload applications were used. The process *CPU-bound* consumes CPU by crunching math operations, creating load in competition with SRT processes. The soft real-time application *srtsim* generates a periodic

Table 1. Summary of percentage of deadlines missed in all experiments. Experiments were repeated and missed deadlines averaged over several runs (percent missed deadlines are to the nearest tenth of percent with a 95% confidence interval).

		Simulated Scheduler		Actual Scheduler		
Experiment	**Process**	**Linux**	**BeRate**	**Linux 100 Hz**	**Linux 800 Hz**	**BeRate**
1	1 CPU-bound process					
	srtsim (25fps 50%)	7.7	0.3	19.7	10.7	0.0
2	2 CPU-bound processes					
	srtsim (25fps 25%)	7.7	0.2	21.5	5.1	0.0
	srtsim (25fps 25%)	7.2	0.2	21.5	5.1	0.0
3	2 CPU-bound processes			*figure not shown*		
	srtsim (25fps 25%)	10.0	0.1	27.6	7.0	0.0
	srtsim (33fps 25%)	7.6	0.1	31.2	5.2	0.0
4	3 CPU-bound processes					
	srtsim (25fps 25%)	5.6	0.0	17.8	0.9	0.0
5	1 CPU-bound process with nice +10					
	srtsim (33fps 67%)	11	0.2	32.4	0.8	0.0
6	1 CPU-bound process					
	srtsim (25fps 50%)	31.8	14.4	45.4	>15 †	>14 †
	srtsim (50fps 50%)	21.0	14.3	43.5		

† When overloaded, number of missed deadlines did not converge.

deadline workload that models frame-to-frame variability common with decoding MPEG video streams [3, 5].

We find in general, the Linux scheduler performs reasonably well when the total demand of soft real-time processes is less than 100% of the CPU and a process i requires no more than than $s_i = (q_i / \sum_{x \in n} q_x)$ of the CPU, where n is the set of running processes, and q_x is the share allocated to x. As the processing need of a soft real-time (SRT) process approach its load share, Linux is less effective at meetings deadlines, because it may service processes in arbitrary order. Time-share scheduling algorithms are unaware of resource requirements or deadlines, and well-intentioned scheduling decisions may result in some processes missing deadlines that could otherwise be met.

Our results show that BeRate alleviates the problem seen in Linux when an SRT process requires a CPU allocation close to its fair share. Figure 2 plots the progress when a CPU-bound and a SRT process requiring 50% CPU share the processor bandwidth. Because the Linux scheduler provides approximately equal CPU to each application, the SRT process should meet its deadlines. However, the SRT process misses 7.7% of its deadlines. Although the process receives enough CPU allocation, it does not always receive it in time. In the BeRate scheduler the SRT task misses 0.3% of deadlines while providing the equal resources to each application. We found that we must reduce the average usage of *srtsim* to below 40% before the Linux scheduler meets performance of the BeRate scheduler. In Figure 3, we doubled the number of both CPU-bound and SRT tasks, with each SRT process requiring a quarter of CPU bandwidth. Like in the previous experiment, in Linux each SRT process missed more than 7% of its respective deadlines, while in BeRate neither process missed more than 0.2%. We repeated the experiment, assigning the two SRT processes different frame rates, with similar results (figure not shown, but the results are summarized in Table 1).

Figure 4 shows finer-grain detail of the allocation by plotting on a shorter scale. Three CPU-bound processes compete with a single SRT process (requiring 25% CPU). The Linux scheduler provides a quarter of CPU to each process, but like the previous experiment, the SRT process is unable to meet deadlines, missing 5.6% of them, while in BeRate it misses almost none. In Linux, at several instances the SRT process misses a deadline because it is halted while a CPU-bound processes executes. This is due to phasing of dynamic priorities assigned by the Linux scheduler. The SRT process's dynamic priority decays at a slower rate than CPU-bound processes, and usually upon waking it preempts the currently executing process. However, when all quanta expire the dynamic priorities of processes are recomputed, and occasionally the SRT process is not greater upon waking, allowing a CPU-bound process to complete an entire quantum without interruption. The BeRate scheduler eliminates this problem, resulting in evenly spaced CPU allocations.

UNIX users may adjust the relative priorities of pro-

cesses using the *nice* utility, setting a priority in the range -20 to +19. In the Linux implementation *nice* scales a process's time quantum. A process with a *nice* of +10 receives $\frac{1}{2}$ of the default quantum, so when competing with another process of default *nice* of 0 its allocation is reduced to $\frac{1}{3}$ of the CPU. In Figure 5 the SRT process with frame rate of 33 frames/second requires an average of $\frac{2}{3}$ of the CPU to meet its deadlines, which we allocate by assigning the CPU-bound process a *nice* of +10. Even though the Linux scheduler provides the SRT process enough share to meet deadlines, it does not receive them in a timely manner and misses 11%. In the BeRate scheduler the SRT process misses few ($< 0.2\%$) deadlines.

In the last experiment, we compare BeRate to our BEST scheduler [2]. BEST attempts to meet any deadlines it can detect, while the goal of BeRate is only to meet those which may be met within the process's fair share. Because BeRate does not attempt to allocate more than a processes' fair share of resource, an SRT process needing more than its nominal share to meet deadlines may not perform well. Figure 6 shows the performance with three processes, one CPU-bound, and two SRTs that require 50% of CPU (but differ in frame rate). In this experiment, not all deadlines can be met. The BEST scheduler performs well, missing only 1.6% of each processes deadlines, but the CPU-bound process makes little progress. As expected, BeRate is not capable of meeting deadlines, but performed similarly to Linux, preserving the fair-share strategy during overload.

6 Conclusion

Best-effort schedulers make no resource guarantees, and are thought to perform poorly for soft real-time applications. Nevertheless, the best-effort model continues to be attractive for both application developers and users because it is simple to use. BeRate is a CPU scheduler that adheres to a best-effort scheduling policy while improving the responsiveness of periodic soft real-time processes.

Like the Linux scheduler, BeRate schedules without *a priori* knowledge of resource needs, and with fairness specified by user-assigned priorities. The BeRate scheduler uses the best-effort model, so no process is ever refused admission or provided a service guarantee. Like other best-effort systems, if the user overburdens the system, the user will experience degraded system performance [25]. However, in the presence of other applications or heavy (but not overburdened) use, the BeRate scheduler effectively meets soft real-time deadlines. Our experiments show that BeRate is effective at allocating CPU with less scheduling latency to processes that exhibit periodic behavior, and exceeds the performance of Linux in situations where deadlines can be met.

Acknowledgments We gratefully acknowledge Lonnie Welch and Hermann Härtig for technical discussions of this research. This research was funded by a DOE High-Performance Computer Science Fellowship.

References

[1] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of the linux kernel. In *Real-Time Technology and Applications Symposium (RTAS02)*, Sept. 2002.

[2] S. Banachowski and S. Brandt. The BEST scheduler for integrated processing of best-effort and soft real-time processes. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, pages 46–60, Jan. 2002.

[3] S. A. Banachowski. Using the best-effort scheduling model to support soft real-time processing. Master's thesis, University of California, Santa Cruz, Aug. 2002.

[4] A. Bavier and L. L. Peterson. BERT: A scheduler for best effort and real-time tasks. Technical Report TR-587-98, Princeton University, Aug. 1998.

[5] A. C. Bavier, A. B. Montz, and L. L. Peterson. Predicting MPEG execution times. In *Proceedings of the 1998 SIGMETRICS Conference*, pages 131–140, June 1998.

[6] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison–Wesley, 2nd edition, 1998.

[7] S. Brandt and G. Nutt. Flexible soft real-time processing in middleware. *Real-Time Systems*, pages 77–118, 2002.

[8] G. M. Candea and M. B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 157–166, Aug. 1998.

[9] H. Chu and K. Nahrstedt. A soft real time scheduling server in UNIX operating system. In *European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, Sept. 1997.

[10] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating System Principals*, Dec. 1999.

[11] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Effects of clock resolution on the scheduling of real-time and interactive processes. Technical Report 2001-14, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Nov. 2001.

[12] B. Ford and S. Susarla. CPU inheritance scheduling. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, Oct. 1996.

[13] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Oct. 1996.

[14] K. Jeffay and D. Bennett. A rate-based execution abstraction for multimedia computing. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, Apr. 1995.

[15] S. Khanna, M. Sebrée, and J. Zolnowsky. Realtime scheduling in SunOS 5.0. In *USENIX Winter 1992 Technical Conference*, pages 375–390, Jan. 1992.

[16] C. Lin, H. Chu, and K. Nahrstedt. A soft real-time scheduling server on the Windows NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*, Aug. 1998.

[17] G. Lipari and S. K. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS00)*, pages 166–175, May 2000.

[18] R. M. Love. Linux preemptable kernel patch. http://www.tech9.net/rml/linux, Oct. 2002.

[19] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison–Wesley, 1996.

[20] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

[21] A. Morton. Linux scheduling low-latency patch. http://www.zip.com.au/ akpm/linux/schedlat.html, Jan. 2001.

[22] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4UNIX scheduler unacceptable for multimedia applications. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, 1993.

[23] J. Nieh and M. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, Oct. 1997.

[24] M. A. Rau and E. Smirni. Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads. In *Proceedings of the 7th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '99)*, Mar. 1999.

[25] J. Regehr, M. B. Jones, and J. A. Stankovic. Operating system support for multimedia: The programming model matters. Technical Report MSR-TR-2000-98, Microsoft Research, Sept. 2000.

[26] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, Dec. 2001. IEEE.

[27] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Buruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the Real-Time Systems Symposium*, pages 288–299, Dec. 1996.

[28] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Sept. 1995.

[29] D. K. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. In *ACM Multimedia Conference*, Nov. 1996.

[30] V. Yodaiken and M. Barabanov. Real-time Linux. In *Proceedings of Linux Applications Development and Deployment Conference (USELINUX)*, Jan. 1997.

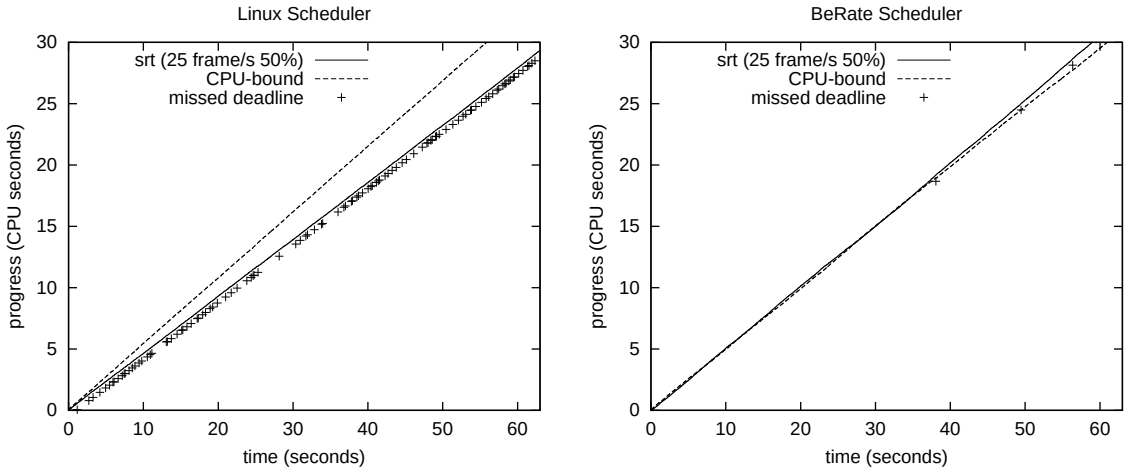


Figure 2. The progress of applications, with Linux and BeRate running (1) CPU-bound and (2) srtsim 25fps 50%. The crosses below the progress line indicate missed deadlines.

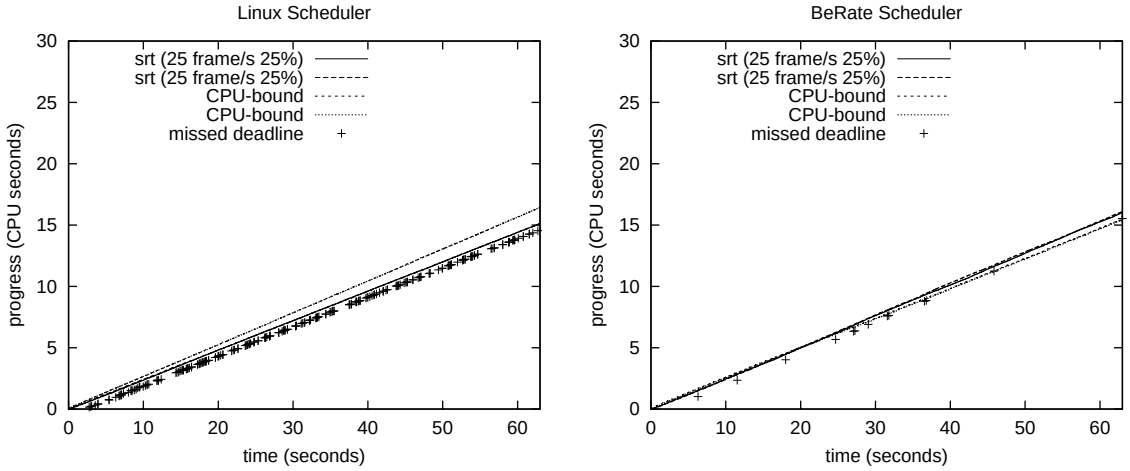


Figure 3. The progress of applications, with Linux and BeRate running (1-2) CPU-bound and (3-4) srtsim 25fps 25%. The crosses below the progress line indicate missed deadlines.

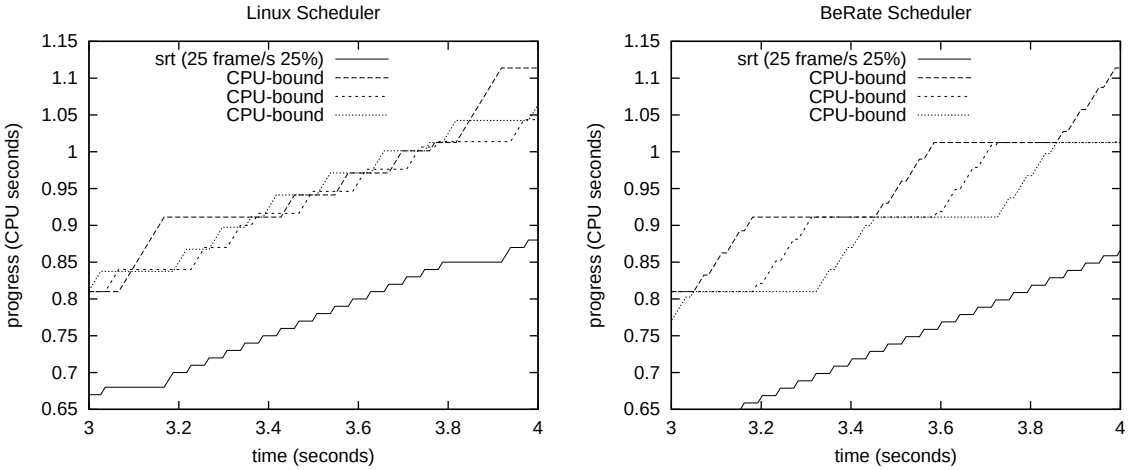


Figure 4. Progress of applications, with Linux and BeRate schedulers running (1-3) 3 CPU-bound processes and (4) srtsim 25fps 25%.

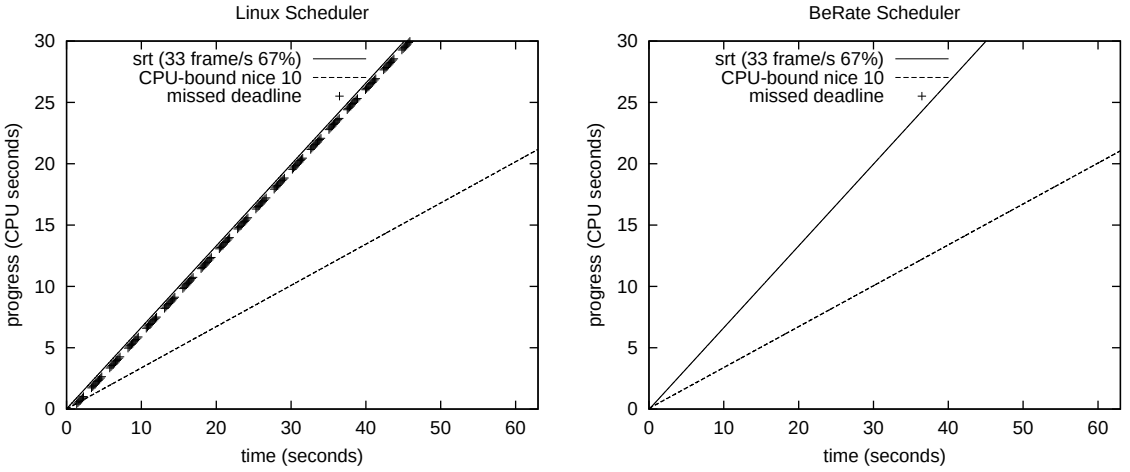


Figure 5. The progress of applications, with Linux and BeRate running (1) CPU-bound (w/ nice 10) and (2) srtsim 33fps 67%. Linux misses many deadlines, indicated by the bunches of crosses below the progress line.

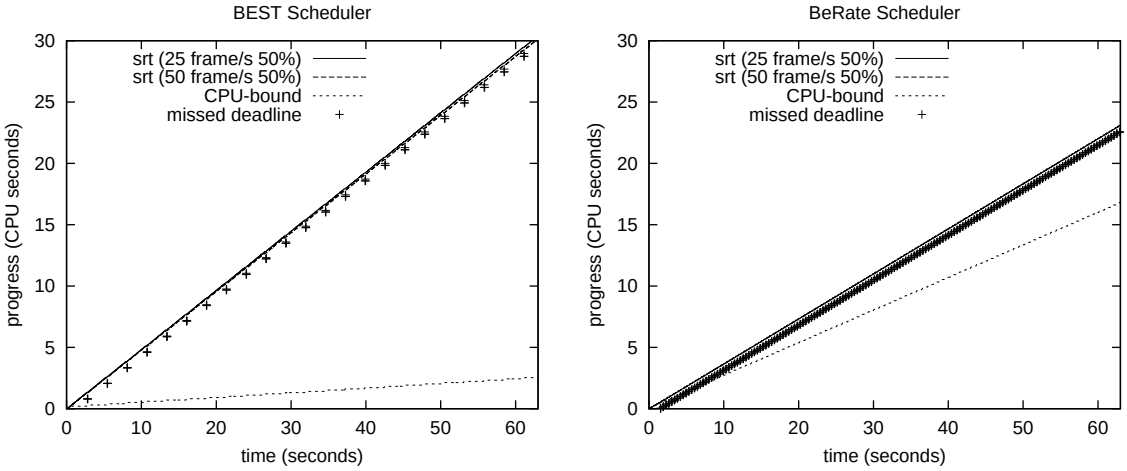


Figure 6. The progress of applications, with the BEST and BeRate schedulers with (1) CPU-bound, (2) srtsim 25fps 50% and (3) srtsim 50fps 50%