The BEST scheduler for integrated processing of best-effort and soft real-time processes

Scott A. Banachowski and Scott A. Brandt

Computer Science Department, University of California, Santa Cruz

ABSTRACT

Algorithms for allocating CPU bandwidth to soft real-time processes exist, yet best-effort scheduling remains an attractive model for both application developers and users. Best-effort scheduling is easy to use, provides a reasonable trade-off between fairness and responsiveness, and imposes no extra overhead for specifying resource demands. However, best-effort schedulers provide no resource guarantees, limiting their ability to support processes with timeliness constraints. Reacting to the need for better support of soft real-time multimedia applications while recognizing that the best-effort model permeates desktop computing for very good reasons, we have developed BEST, an enhanced best-effort scheduler that combines desirable aspects of both types of computing. BEST provides the well-behaved default characteristics of best-effort schedulers while significantly improving support for periodic soft real-time processes. BEST schedules using estimated deadlines based on the dynamically detected periods of processes exhibiting periodic behavior, and assigns pseudo-periods to non-periodic processes to allow for good response time. This paper discusses the BEST scheduling model and our implementation in Linux and presents results demonstrating that BEST outperforms the Linux scheduler in handling soft real-time processes, outperforms real-time schedulers in handling best-effort processes, and sometimes outperforms both, especially in situations of processor overload.

Keywords: Soft real-time, scheduling, Linux

1. INTRODUCTION

Although many people wish to use their desktop workstations as multimedia platforms, conventional desktop operating systems do not directly support the scheduling needs of soft real-time multimedia applications. Schedulers in conventional desktop operating systems use *best-effort* time-sharing policies designed to reduce the latency of interactive processes and provide adequate progress for all processes. Because they provide no guarantees of processing bandwidth, multimedia applications may or may not receive the timely scheduling they require in order to play continuous sound or video.[1]

Our previous work examined dynamic desktop soft real-time using Dynamic QoS Level Resource Management (DQM).[2, 3] In that work we showed that it is possible to robustly execute soft real-time applications on best-effort systems. In particular, we developed a middleware framework that allowed applications to dynamically adjust their resource usage based on the available resources. By adjusting resource usage such that the set of running applications use less than 100% of the available resources, a best-effort scheduler is able to provide reasonable soft real-time performance. In that work we also demonstrated dynamic estimate refinement, a technique that enables the system to dynamically adapt to incorrect or unspecified resource usage estimates.

Like most soft real-time systems, the DQM system has several issues that limit its ultimate utility in generic desktop environments. First, because it is a middleware solution, the performance of soft real-time applications varies significantly in the presence of best-effort or other applications that do not cooperate with the middleware resource manager. Second, like most soft real-time systems[4–10] it requires applications to interface with special-purpose soft real-time interface routines. Third, like other soft real-time systems, the DQM requires that applications provide *a priori* estimates of resource usage and period. Although the DQM

Further author information:
Email: {sbanacho,sbrandt@cse.ucsc.edu} Address: Computer Science Department, School of Engineering, Applied Sciences Bldg., U.C. Santa Cruz, 1156 High Street, Santa Cruz, CA 95064.

system can dynamically adjust to incorrect or unspecified resource usage estimates, it cannot adapt to incorrect or unspecified application periods.

This paper presents a solution that addresses those issues—BEST, a time-sharing scheduler that directly supports multimedia applications while providing adequate progress and response time for best-effort applications. BEST dynamically measures process behavior and uses the information to aid in scheduling decisions. By detecting the rate at which waiting processes enter the run queue, the scheduler boosts the performance of "well-behaved" periodic processes by increasing their priority, while preserving the behavior of traditional time-sharing schedulers for non-periodic processes; we use a **B**est-effort scheduler that is **E**nhanced for **S**oft real-time **T**ime-sharing, so we call it the BEST scheduler.

Current approaches to soft real-time scheduling require special interfaces to the scheduler—BEST differs by removing software authors' and users' awareness of the scheduler. Most specialized systems place a burden of specifying scheduling needs of an application either with the developer, who must use system calls to communicate and negotiate with a scheduler, or the user, who explicitly chooses priorities or executes external software to control scheduling policies. The BEST scheduler uses the best-effort model, so no process is refused admission or provided a service guarantee. This is an attractive model because it incurs no overhead for programmers or users. Like other best-effort systems, if the user overburdens the system, the user will experience degraded system performance.[11] However in the presence of other applications or heavy (but not overburdened) use, the BEST scheduler effectively meets soft real-time deadlines for applications that are well-behaved. And when overburdened, BEST continues to provide satisfactory progress to all applications.

This paper describes an implementation of the BEST scheduler in the Linux kernel. Section 2 summarizes related research that supports scheduling for multimedia. Section 3 describes the scheduler implementation, and Section 4 presents quantitative performance data. Finally, Section 5 discusses our future plans for this research and Section 6 provides concluding remarks.

2. RELATED WORK

Continuous real-time applications require enough processor bandwidth to meet their periodic deadlines.[12] We classify multimedia applications as *soft real-time* because, like real-time processes, they must meet periodic deadlines, but missing an occasional deadline results in diminished performance rather than outright failure.[13]

2.1. Real-time Scheduling

Real-time systems, such as RT-Mach,[10] are designed to meet hard deadline constraints. Some versions of UNIX support real-time scheduling classes,[14] and many systems adapt the POSIX standard for real-time extensions.[15] In order to ensure predictable behavior, these system use strict scheduling policies such as Rate Monotonic (RM) or Earliest Deadline First (EDF).[16] These scheduling algorithms require that the worst-case workload is known when configuring a system. For industrial applications, where systems are typically dedicated to specific purposes and deadlines are hard, real-time systems are attractive because they may be tuned to perform predictably. However real-time operating systems are not well-suited for desktop use because workload cannot in general be predicted. Most multimedia scheduling research focuses on integrating the desirable features of hard real-time scheduling into general-purpose systems that have inconsistent workloads; an example is the Nemesis Atropos scheduler, which uses EDF based on deadlines derived from a process's specified share of CPU bandwidth.[17] Multics also provides an EDF scheduler, using desired response time to determine virtual deadlines of non-real-time processes.[18] Like these systems, BEST schedules by earliest deadline, but unlike previous systems it automatically detects the periods of processes and assigns appropriate deadlines based on this information, and assigns pseudo-deadlines to non-periodic processes and schedules accordingly.

2.2. Multi-level Scheduling

One approach for handling a mix of applications divides processes according to type, and assigns each type to different schedulers; each scheduler uses the policy best suited for its type. In hierarchical schemes, a lower-level scheduler receives bandwidth allocated by the higher-level scheduling policy. For example, in Real-Time Linux,[19] the Linux kernel executes as the lowest priority task in a real-time scheduler alongside the other higher priority real-time tasks. POSIX extensions also implement hierarchical scheduling—the real-time classes defer to the time-sharing class when no real-time process is ready to execute.

Researchers use several techniques of adapting multi-level scheduling to the needs of soft real-time systems. Taking advantage of the POSIX multi-level scheduling classes, user processes may schedule soft real-time processes by dynamically altering their priorities,[20, 21] thereby removing soft real-time scheduling decisions from the kernel. Some more sophisticated approaches to hierarchical scheduling include the SFQ algorithm, which proportionally shares bandwidth among the levels so that time-critical applications receive adequate resources,[22] and CPU Inheritance Scheduling,[23] which allows scheduling threads to donate processing to other scheduler threads in flexible arbitrary arrangements of hierarchies. The Vassal project[24] adds a system interface for users to install their own schedulers. Another method applied to soft real-time is middleware resource management[*]. Middleware managers monitor a system's resources usage, and provide recommendations to adaptive soft real-time processes. DQM[2] uses this approach to maximize benefits for scalable soft real-time processes, independent of the underlying kernel scheduler.

The architectural approach of dividing scheduling into levels creates flexibility for systems 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. System architects, and in some cases users, must make informed decisions for the layout of scheduling hierarchy. For the BEST scheduler, we do not introduce the complexity of multiple levels of scheduling, and instead rely on a single algorithm for all processes. The algorithm is designed to minimize latency for periodic and interactive processes. However, using the scheduler does not preclude integration into multi-level schemes.

2.3. Proportional-share Scheduling

Recognizing the low predictability of general-purpose system workloads and the relaxed deadline requirements for multimedia applications, a large body of research focuses on creating new schedulers better suited to a mix of application types. Most systems allocate each process a share of processing bandwidth, and use an algorithm to assign allotted CPU guarantees within minimal error bounds. For periodic applications, share is allocated to meet the execution rate required to meet deadlines. Fair-sharing is enforced so no process inhibits another's ability to meet deadlines.

Proportional scheduling systems share similar concepts yet differ in strategy. Here we briefly mention some systems; this list is not comprehensive. EEVDF[25] calculates a virtual deadline for each process as a function of measured and allotted share, and schedules according to EDF; Stride Scheduling[26] uses a similar notion of virtual time. Systems such as BVT[5] and BERT[4] provide enhanced fair-sharing algorithms aimed at increasing the throughput of deadline-sensitive processes by dynamically reallocating shares on a short-term basis. Some systems utilize admission control: processes reserve shares, and the scheduler denies admission when requested reservations are not available.[8] The CM[27] and SMART[9] schedulers provide feedback to applications so they may adapt to dynamically changing loads, allowing the scheduler to adjust to higher workloads without resorting to restricting admission.

To meet deadlines, the proportional scheduler must determine the proper share for each process; a process must somehow specify its rate requirement. In many cases, this information is built into the process, and upon start-up it notifies the scheduler through a system API. It may be difficult to determine a desired rate if the speed of the target processor is unknown; abstractions for specifying rate address this problem[6] (however because the abstracted rates are typically not expressed in units of system clock ticks, clock skew is inevitably introduced). For systems that include feedback from the scheduler to the process, greater flexibility comes at the expense of even more demand on application developers. Additionally, some systems provide mechanisms for users to specify the quality of service they desire from a process, and allow run-time modification of share assignments through GUIs, placing the burden of scheduling specification on both the developers and the software users. The BEST scheduler does not need to be informed of processes' rates, making the development and use of SRT applications easier.

[*]While not strictly hierarchical, middleware can be considered a meta-scheduler for participating processes.

2.4. The State of SRT Scheduling

The scheduling algorithms and systems proposed by researchers support service guarantees that are not possible with best-effort scheduling. However, fully utilizing them involves difficult decisions provided by system builders, applications developers, and users. System builders must set appropriate architecture for hierarchies of schedulers. Developers must conform to new system APIs, reducing the portability of applications. Users must hassle with tuning the scheduling parameters for desired performance; the average desktop user may not be interested in or capable of accomplishing this task. Our experience suggests that most multimedia applications only suffer occasional glitches which may be adequately addressed with better best-effort scheduling. The ease and simplicity of best-effort scheduling makes it the most attractive model for many platforms—by enhancing the performance of soft real-time processes, the users may never notice the absence of service guarantees.

3. IMPLEMENTATION

The goal of BEST is to enhance the performance of soft real-time tasks by detecting periodic processes and boosting their priorities to improve their chances of meeting their deadlines. Like most UNIX schedulers,[28, 29] it dynamically calculates process priorities. It is aimed at desktop users who desire better performance from multimedia applications without the complexity of a system with service guarantees.

The scheduler must decide which programs have periodic deadline requirements by making an assumption: applications with periodic deadlines enter a *runnable state* when they begin a periodic computation, and upon completion use synchronization primitives (such as timers) to wait for the beginning of the next period. We predict that by observing the times that a process enters the queue of runnable processes we can make reasonable guesses about its period. A periodic process is "well-behaved" if it enters the runnable queue in a predictable pattern. It is possible that some processes that repeatedly enter the runnable state may be misidentified as having a periodic deadline even though they do not; in this case, they may benefit from the mistake. This is not a concern as long as the CPU resource isn't significantly overburdened, and the scheduler can be tuned to minimize the likelihood of such occurrences.

In order to test the assumption that multimedia processes enter the run queue with predictable period, we instrumented the Linux kernel to record the entry time of a process to the nearest $\frac{1}{800}$ of a second, and then ran some sample single-threaded multimedia processes. We examined mpeg_play, a desktop MPEG video player, and mpg123, a desktop mp3 audio player. We found that these processes did exhibit measurable periodic behavior. Table 1 shows the average period and standard deviation in microseconds for the sample runs. These experiments were executed on a 650 MHz AMD i686 system.

Both multimedia programs enter the run queue with detectable periods, however their behavior differs: the video player must keep a nominal framerate, whereas the audio player periodically feeds a buffer so its timing requirement is not as strict; it only needs to keep the buffer from completely draining. When mpeg_play displays frames, it enters the run queue twice per frame, once for frame synchronization and once waiting for a video buffer, then sleeps until it is time to display the next frame. Because it enters the run queue twice during the same tick, the detected average period is half the actual frame period, and the standard deviation is close

Table 1: Average period, standard deviation, and CPU usage for sample multimedia processes.

Process	**Average period (ms)**	**Standard deviation**	**CPU usage**
mpeg_play (24 frame/s)	21.7	21.9	19.3%
mpeg_play (24 frame/s) (no display)	42.3	9.2	13.0%
mpeg_play (30 frame/s)	17.0	17.4	19.5%
mpeg_play (30 frame/s) (no display)	34.6	13.1	15.9%
mpg123 (128 kbit/s)	160.2	31.7	2.2%
mpg123 (128 kbit/s) (different mp3)	160.2	31.8	2.2%

to the average period (Section 3.3.2 explains how BEST deals with this anomaly). When display is disabled, mpeg_play processes the file without rendering it. In this case it does not block waiting for the video frame buffer, and its period it equivalent to its frame rate. The audio player is not driven by a framerate or clock; it repeatedly fills a buffer with audio data, sleeping for a fixed period between each fill. As expected, it exhibits periodic behavior, but since the period is not scheduled at a specific rate it has higher variance than the video player.

3.1. Design Goals

In developing the BEST scheduler we had a number of specific design criteria. The criteria and rationale for the scheduler design are:

1. The same scheduling policy should apply to every application, regardless of its scheduling needs—a uniform algorithm simplifies scheduling decisions.
2. Neither users or developers should have to provide any *a priori* information about the process to be executed.
3. The scheduler should enhance the performance of soft real-time applications.
    - Processes that enter the runnable queue in predictable patterns should receive a priority boost; it should be based on classical real-time scheduling results, i.e. the priority boost should be based on measured rate or deadline.
    - This algorithm should create a positive feedback loop for well-behaved soft real-time processes; when processes do not miss deadlines, they have the opportunity to wait for the next period, increasing the likelihood of consistent patterns.
    - The scheduler should be preemptive. Since periodic processes have higher priority, this prevents missed deadlines.
4. The default behavior of the scheduler should be reasonable and consistent with general purpose time-sharing schedulers.
    - The scheduler should favor interactive processes over CPU-bound processes.
    - No process should starve. The presence of compute intensive periodic processes cannot completely hinder the progress of other processes. Processes will receive time slices that prevent them from monopolizing the CPU and improve overall responsiveness.
    - When the system is not fully loaded, changes to workload should not effect the performance of already executing processes. For fully loaded systems, performance should degrade gracefully.

3.2. Linux Scheduler Details

We implemented the BEST scheduling algorithm in the Linux 2.2.5 kernel. We selected Linux as a development platform because it is a popular desktop environment and source code is readily available. A brief description of the unmodified Linux scheduler is instructive for understanding the differences.

A function called `schedule()` allocates the CPU to a process. It loops through all processes in the runnable queue, and selects the one with highest dynamic priority. 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 if the running process's `need_resched` flag is set. For example, when a process's time quantum expires, the timer interrupt handler sets its `need_resched` flag.

A function called `goodness()` calculates dynamic priorities. The dynamic priority is also interpreted as a time quantum, and decreases for every clock tick the process executes. When all runnable processes consume their quantum, `schedule()` loops through all processes (including those not in the run queue) and recomputes their dynamic priority using $pri = pri/2 + nice$, where $nice$ is a positively scaled user-settable scheduling priority. When this calculation occurs, a suspended process with a non-zero time quantum receives a priority boost; the purpose is to increase the responsiveness of interactive processes over CPU-bound processes. For a program that is always suspended, the priority as a function of the number of calculations n, is $pri(n) = [(2^{n+1} - 1)/2^n] \times nice$. Priority quickly increases, but as $n \to \infty$, $pri = 2 \times nice$, limiting the priority (and time quantum) from growing too large.

The Linux scheduler implementation is designed to mimic the behavior of a multi-level feedback queue, and although dynamic priority calculations differ from 4.4BSD (well-documented by McKusick *et al.*[29]), the goal of the scheduling policy remains the same: favor I/O-bound over CPU-intensive processes while allowing no process to starve. The 4.4BSD scheduler differs from Linux by including a time-decaying estimate of the CPU usage in the priority calculation. The Windows NT scheduler employs a similar technique.[24]

3.3. BEST Scheduler Details

The BEST scheduler uses an even simpler algorithm than the Linux scheduler. Every process has a deadline that is computed when the process enters the runnable queue. The `schedule()` function selects the runnable process with the earliest deadline. Since we do not know a process's deadline, a simple heuristic is used to estimate its period, and the deadline is set to the expiration of its next period.

3.3.1. Period detection

BEST estimates period when a process enters the runnable queue (queue entry is through a single function called `wake_up_process()`). The estimated period P_{est} is the time that elapsed since the process previously entered the runnable queue. The new effective period P_n is calculated by taking a weighted average with the previous period P_{n-1}, stated as $P_n = (P_{est} + w \times P_{n-1})/(1 + w)$. Adjusting the weight factor w controls how fast the scheduler forgets previous behavior. If the period exceeds a maximum value it is truncated, therefore periods longer than this maximum are not detected. The `wake_up_process()` function determines a process's next deadline by adding its effective period to the current time. The scheduler uses the deadline as a priority when selecting runnable processes. This function also sets an additional value called the *deadline timer*. This timer indicates how much CPU time a process may receive before its deadline is reset. Like the quantum timer, the deadline timer value is decremented for every tick the process executes. The deadline and deadline timer values are stored in the process's state structure.

Every time a process is scheduled for the CPU it executes until either its quantum expires, it blocks, or it is preempted, at which time `schedule()` is triggered. Before selecting the next process to execute, if the current process's deadline timer expired, `schedule()` sets its next deadline to a time beyond the maximum detectable period—in effect it lowers the priority of any process that doesn't leave the runnable queue before its deadline timer expires. Only `wake_up_process()` resets the deadline timer, so for a process that never blocks `schedule()` will postpone its deadline whenever it executes. Postponing a deadline to greater than the maximum period ensures that processes with detected periods will have earlier deadlines. Because a postponed deadline is not recomputed until after the process is allocated the CPU, starvation is prevented. Once a deadline is set, eventually the process will be scheduled as time advances.

3.3.2. Confidence

Not every process that repeatedly wakes up is periodic, so an additional step evaluates the *confidence* that the estimated period is indeed due to periodic behavior. Confidence is a measure of the difference between the current measured period, and the nearest multiple of the average period. (The expression $|P_{est} \bmod P_n - P_n/2|$ calculates a confidence value between 0 and $P_n/2$, but in practice we use bit shifts and fixed-point math, yielding a value normalized between 0 and 16). A process is "well-behaved" if its confidence exceeds a threshold. A process that is not well-behaved will receive a deadline timer of 0, meaning that it is eligible to have its deadline reset following its next scheduled quantum. Although scheduling priority is set according to its estimated period, once it uses its current quantum the process will be rescheduled with a CPU-bound pseudo-period unless its confidence increases above threshold.

By calculating confidence using a multiple of the period (implicit in the modulo operation), we elude the effect of detecting an average period that is half the actual period (as observed in Table 1, where we saw that a video player entered the run queue twice per frame). This effect also helps processes that miss an occasional deadline. When a periodic process misses a deadline, it may not sleep and wake up again until a later period, when it successfully completes a computation on time. This process will still receive a high confidence rating when it does, allowing it another opportunity to meet its next deadline. However, the weighting of its time-decaying average period will impact its next deadline assignment and subsequent confidence rating. The coupled effect of this weighting factor and the confidence threshold impact the performance of the scheduler.

3.3.3. Other changes

In order to detect periods of processes with high rates, we increased the timer resolution of Linux by a factor of 8. Linux processes 100 clock ticks per second; for a video player showing 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 ticks per second, measurements are more finely grained and provide a better estimate of the application periods. Interestingly, we found that speeding up the timer *increased* the throughput of processes by about 5%, not the intuitive result expected from increasing the frequency of timer interrupt processing. We do not at present have a satisfactory explanation for this result.

BEST uses a time quantum to limit the time a process may hold the CPU. In Linux, the default processing time quantum is 0.2 seconds, and may be modified between 0.01 and 0.40 seconds using the UNIX *nice* facility. In BEST the default processing quantum is set to 0.1 second, which is historically the quantum used in BSD as it provides an ideal responsiveness for interactive processes.[29] Using the *nice* facility scales this range between 0.005 and 0.20 seconds. In BEST, a running process may be preempted by one with an earlier deadline; however, similar to Linux, context switches are reduced by disabling preemption when a current quantum expires in less than 10 milliseconds. Note that when an executing process still has a positive deadline timer, an expired quantum will not affect its scheduling priority and it is likely to be selected again. This makes the deadline timer behave like an interval timer, as used in Nemesis.[17] In the future, we plan to replace all quanta with interval timers to reduce unnecessary scheduling overhead.

3.3.4. Tuning the scheduler

Several parameters affect the behavior of the BEST scheduler: the maximum period, the deadline timer, the weight used for averaging period measurements, and the confidence threshold. The maximum detectable period controls the responsiveness of CPU-bound processes in a heavily loaded system, since their pseudo-deadlines are delayed beyond this period. The deadline timer dictates the maximum amount of time a process receives before resetting its deadline; if too large, a process may monopolize the CPU, and if too short, it may not meet its deadline. Finally, the averaging weight and confidence threshold impact the effectiveness of the period detection algorithm.

For the BEST prototype described in Section 4, we use a maximum period of 5.12 seconds (with an extra offset of 0.1 second added to the deadline of CPU-bound processes), a weighting average of $\frac{1}{3}$, and a generous threshold that allows any confidence level to pass. The deadline timer is the same as the period, so that a process's deadline is not reset until it uses all the processing time of a period. In practice, a process should use less time than its period, otherwise it will miss deadlines in the presence of any other work and is generally not schedulable on a shared system. These defaults work well in our experiments where all processes were "well-behaved," although we expect them to require tuning for more real workloads.

4. EXPERIMENTAL RESULTS

To examine how well the BEST scheduler meets the design criteria set forth in Section 3.1 we conducted a set of experiments comparing the performance of the BEST scheduler with that of the Linux scheduler and a Rate Monotonic (RM) scheduler. We chose RM as a representative real-time scheduler because the POSIX standard specifies a scheduling class with static priorities capable of supporting RM scheduling. Note, however, that the calculation of priorities for RM scheduling requires a specification of application periods while the default Linux scheduler and BEST do not.

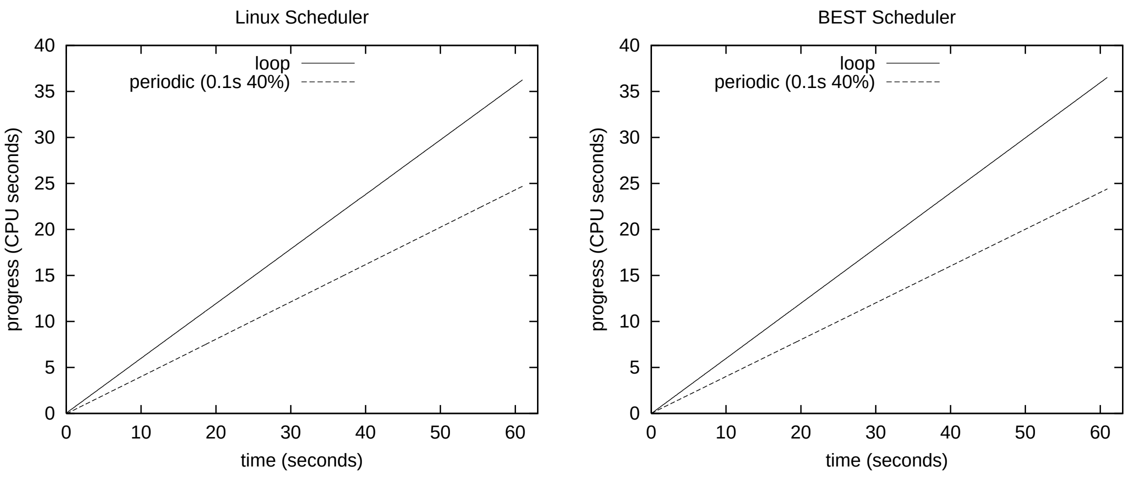


Figure 1: Linux and BEST schedulers running (1) loop and (2) periodic 0.1s 40%.

While running combinations of greedy (CPU-intensive) and periodic (soft real-time) processes, we measured the throughput of all processes and the number of missed deadlines for periodic processes. Two synthetic applications were used in the experiments. *Loop* endlessly consumes CPU bandwidth by crunching math operations. *Periodic* takes two arguments, a period and a percentage; it attempts to consume the desired percentage of CPU bandwidth during each period. If it completes before a deadline it pauses until the beginning of the next period, and if not it records a missed deadline and starts the next period's computation. All experiments were executed on a 200 MHz Pentium Pro system with 256K cache and 256M RAM.

Our results show that 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 each process requires no more than than $\frac{1}{n}$ of the CPU, where n is the number of running processes. This is consistent with our previous work in executing soft real-time processes on best-effort systems. Figure 1 shows the performance of the Linux scheduler and the BEST scheduler with one best-effort process (loop) and one SRT process (periodic, with $\frac{1}{10}$ second period and CPU requirements of 40% of the CPU). Because the Linux scheduler provides approximately equal amounts of CPU cycles to each application and because the SRT process requires less than 50% of the CPU (it's nominal fair share) the Linux scheduler met all application deadlines. Similarly, the BEST scheduler met all deadlines and provided the same amount of resources to each application as the Linux scheduler: 40% of the CPU was granted to the SRT process and 60% was granted to the best-effort process. A similar experiment was performed (Figure 2) replacing loop with a second SRT application (periodic, with the same parameters). The results were similar except that the idle loop consumed the remaining 20% of unused CPU cycles. With reasonable priorities, RM would perform the same.

When a soft real-time process requires more than its nominal share of the available CPU cycles, the Linux scheduler is unable to satisfy it in the presence of CPU-bound best-effort processes. Specifically, when two processes of equal priority compete, the Linux scheduler gives them each about 50% of the available CPU cycles (with slightly more given to processes that block occasionally). Figure 3 shows the performance of the Linux, BEST, and RM schedulers with one best-effort process (loop) and one SRT process (periodic, with $\frac{1}{10}$ second period and 70% CPU usage). Here we see that the Linux scheduler provides approximately 50% of the available CPU cycles to each process, causing the SRT process to miss 29% of its deadlines. By contrast, the RM scheduler (with appropriate priorities) provides the SRT process with 70% of the available cycles, enabling it to meet all of its deadlines while still allowing the best-effort process to progress at a reasonable rate. In this case, the BEST scheduler provided exactly the same performance as the RM scheduler, enabling the SRT process to meet all of its deadlines while still allowing the best-effort process to progress using the remaining CPU cycles. Recall, however, that BEST dynamically determines the application periods whereas Rate Monotonic requires that periods be specified in order to determine appropriate priorities.

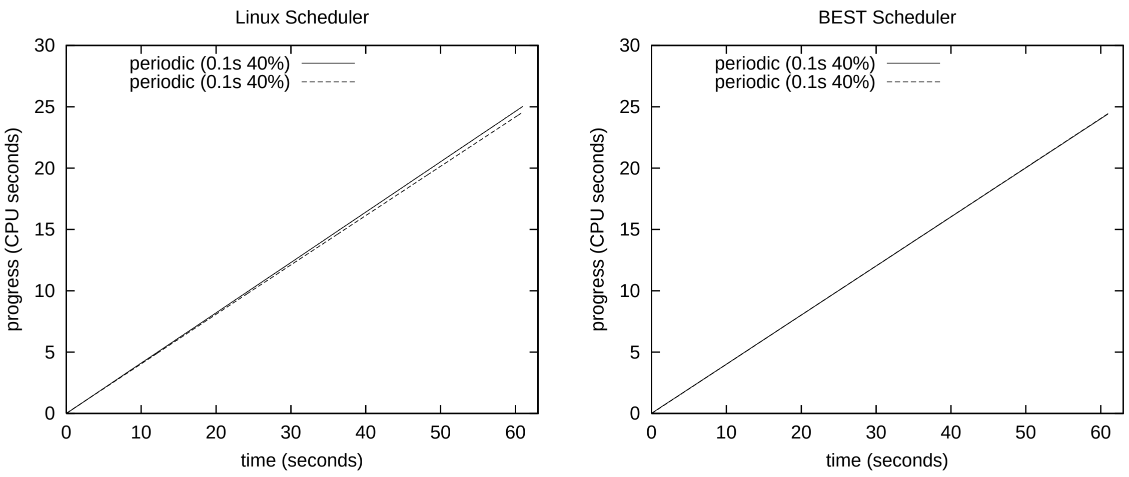


Figure 2: Linux and BEST schedulers running (1) periodic 0.1s 40% and (2) periodic 0.1s 40%.

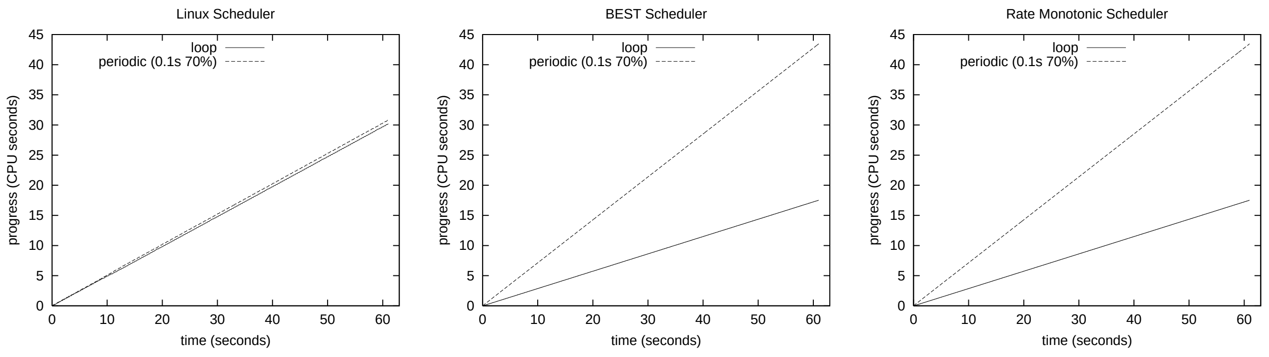
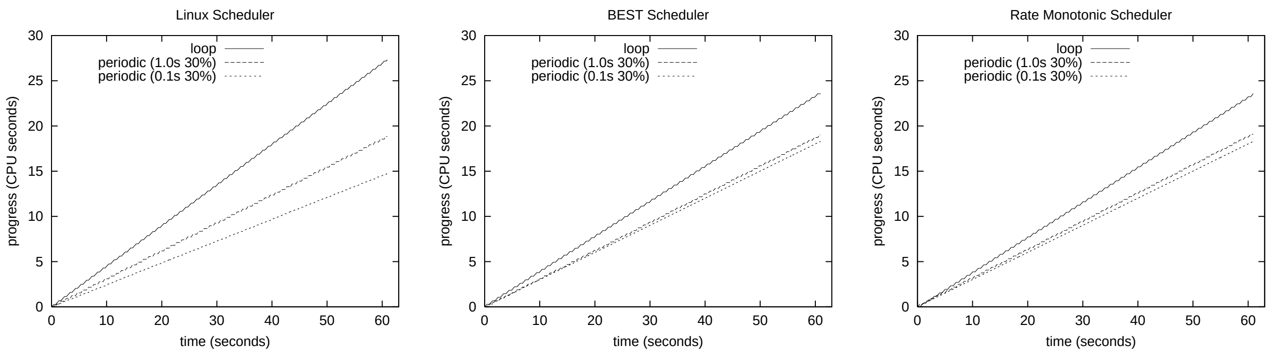


Figure 3: Linux, BEST, and RM schedulers with (1) loop and (2) periodic 0.1s 70%.

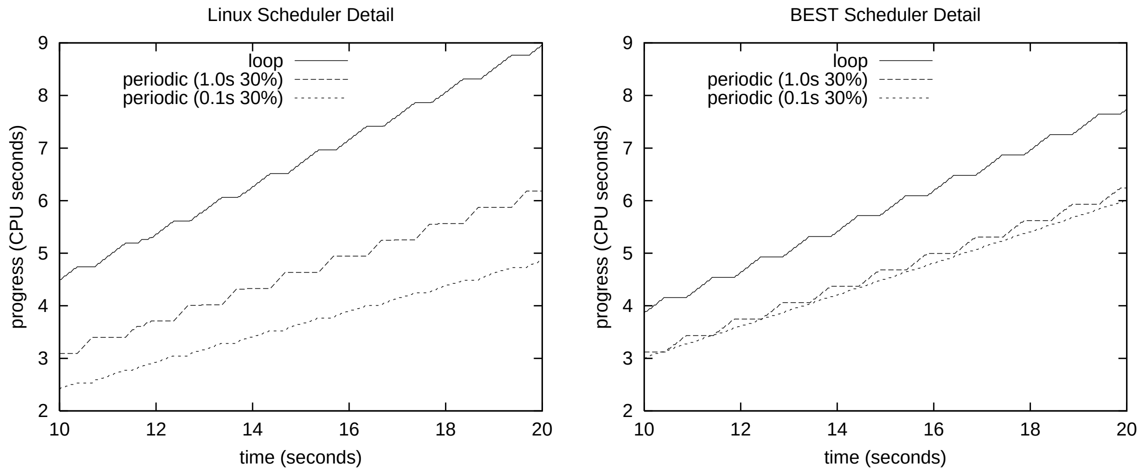
Even in cases where the Linux scheduler should theoretically allow SRT processes to meet all of their deadlines, such as when each requires less than its nominal share of the CPU, it is still possible that each process will fail to meet its deadline. Because the Linux scheduler is unaware of resource requirements or deadlines, its well-intentioned scheduling decisions can result in some processes missing deadlines that could otherwise be met. Figure 4(a) shows the Linux, BEST, and RM schedulers running three processes; one best-effort process and two SRT processes, each of which require 30% of the CPU, one with a period of $\frac{1}{10}$ second and the other with a period of 1 second. Because each SRT process needs less than its nominal share of the CPU, all deadlines should theoretically be met. However, due to the details of the Linux scheduler, we see that in fact the process with the shorter period received less than 30% of the CPU and missed 6% of its deadlines. By contrast, the same processes running with the BEST scheduler missed no deadlines. As can be seen from the graph, the two SRT process received about the same amount of CPU time and approximately 30% of the available cycles. Similarly, these processes made all deadlines under RM scheduling.

Figure 4(b) shows a magnified view of a portion of the data from Figure 4(a), providing greater detail about when each scheduling decision is made and how much CPU is scheduled at each decision. In particular, it shows that under the Linux scheduler, the short period SRT process receives less CPU and at greater intervals than under the BEST scheduler. Under the Linux scheduler, this process experiences a short gap every second where it is not scheduled. This causes the process to miss some deadlines. Under the BEST scheduler, this does not occur and the processes miss no deadlines at all.

While any set of soft-real time processes with a CPU requirement less than 100% is theoretically schedulable



(a) Linux, BEST, and RM schedulers.



(b) Detail view of application progress in Linux and BEST.

Figure 4: Linux, BEST, and RM schedulers with (1) loop (2) periodic 1s 30% and (3) periodic 0.1s 30%.

without missed deadlines, in many cases the Linux scheduler cannot allocate enough share to SRT processes in competition with a greedy process. Figure 5 shows the Linux, BEST and RM schedulers with four processes, one best-effort and three SRT, one with a period of 1 second requiring 30% of the CPU, one with a period of $\frac{1}{2}$ second requiring 30% of the CPU, and one with a period of $\frac{1}{10}$ second requiring 30% of the CPU. The Linux scheduler cannot meet all deadlines because it allocates roughly $\frac{1}{4}$ of the resources to each process, and it performs very poorly, missing 83%, 80%, and 10% of the deadlines of the periodic processes, respectively. Again, like the RM scheduler, the BEST scheduler met all deadlines while still allowing the best effort process to make progress.

One shortcoming of rate-monotonic scheduling is its inability to find a feasible schedule that fully utilizes the CPU when running periodic applications that exhibit non-harmonic periods.[12] By using EDF to make the actual scheduling decisions, BEST does not exhibit this property. Figure 6 shows the performance of the three schedulers with four processes, one best-effort and three SRT, one with a period of 0.61 seconds requiring 31% of the CPU, one with a period of 0.43 seconds requiring 30% of the CPU and one with a period of 0.13 seconds requiring 31% of the CPU. In this case, the SRT processes need 92% of the CPU and have non-harmonic periods. The Linux scheduler provides approximately equal CPU to each process with the result that the best-

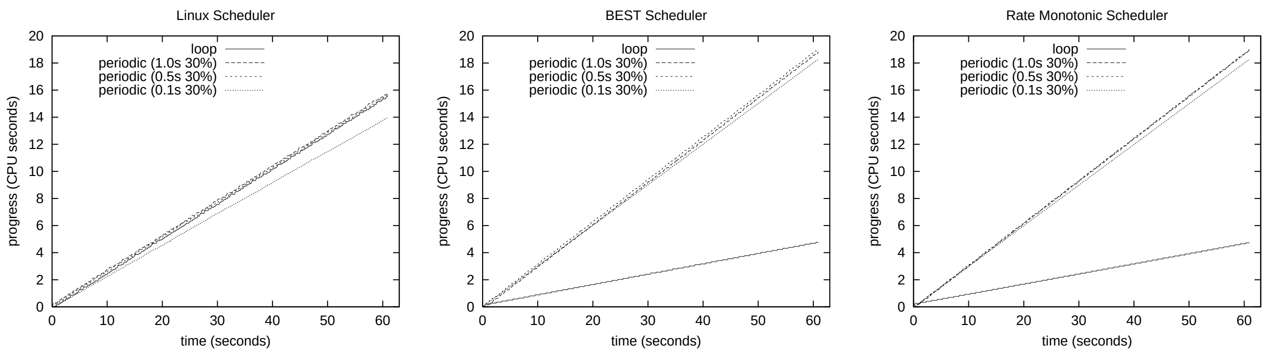


Figure 5. Linux, BEST and RM schedulers with (1) loop (2) periodic 1s 30% (3) periodic 0.5s 30% and (4) periodic 0.1s 30%

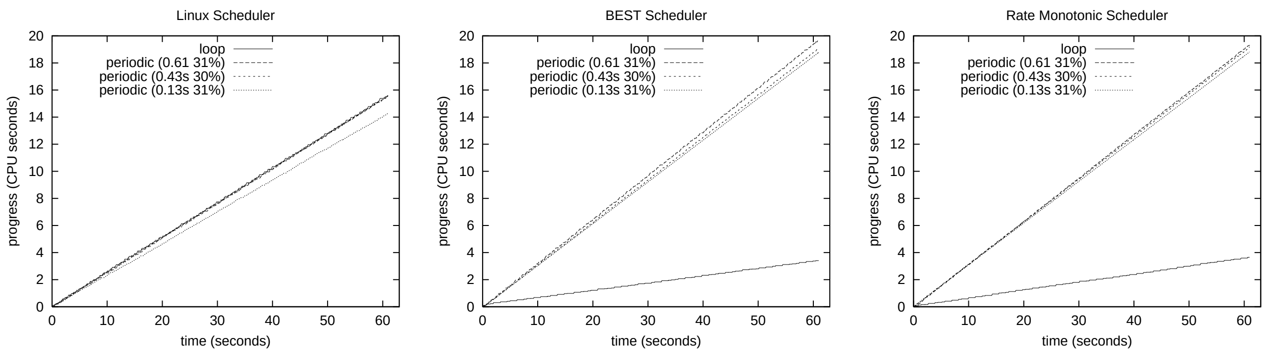


Figure 6. Linux, BEST, and RM schedulers with (1) loop (2) periodic 0.61s 31% (3) periodic 0.43s 30% and (4) periodic 0.13 31%.

effort process makes too much progress and prevents the SRT processes from meeting enough deadlines—they miss 77%, 66%, and 16% respectively. In theory, RM scheduling may not meet all deadlines and that is exactly what we observe. RM causes the process with the longest period (and thus the lowest Rate Monotonic priority) to miss 27% of its deadlines. Outperforming both other schedulers, BEST executes the processes and meets all SRT deadlines.

An important question about any SRT scheduler is how it performs in situations of system overload. Best-effort schedulers will in general allocate a proportional share of the CPU to each process, Rate Monotonic will meet the deadlines of processes with highest priority (generally those with the lowest period),[12] and EDF will miss all deadlines by roughly the same amount.[30] While the overload of either RT scheduler might be considered optimal in some strictly SRT environments, they suffer from the fact that they will starve best-effort processes entirely. Figure 7 shows the performance of the Linux, BEST, and RM schedulers with four processes, one best-effort, one SRT with period 1 second requiring 40% of the CPU, one SRT with period $\frac{1}{2}$ seconds requiring 40% of the CPU, and one SRT with period $\frac{1}{10}$ seconds requiring 40% of the CPU. Because the resource requirements of the SRT process sum to greater than 100% of the CPU, no scheduler can meet all of the deadlines. The Linux scheduler gives each process roughly $\frac{1}{4}$ of the CPU, causing the SRT processes to miss 98%, 93%, and 19% of their deadlines, respectively, and allows the best-effort process to make very good progress, getting a full $\frac{1}{4}$ of the available CPU cycles. The RM scheduler meets all of the deadlines for the two SRT processes with shorter deadlines, but misses 98% of the deadlines for the SRT process with the longest deadline and doesn't allow the best-effort process to run at all. The BEST scheduler embodies the best characteristics of both schedulers, distributing (somewhat) and minimizing the missed deadlines (71%, 5%, and 2% respectively) while still allowing the best-effort process to make reasonable progress.

A primary goal of the BEST scheduler is to minimize the number of missed deadlines for soft real-time

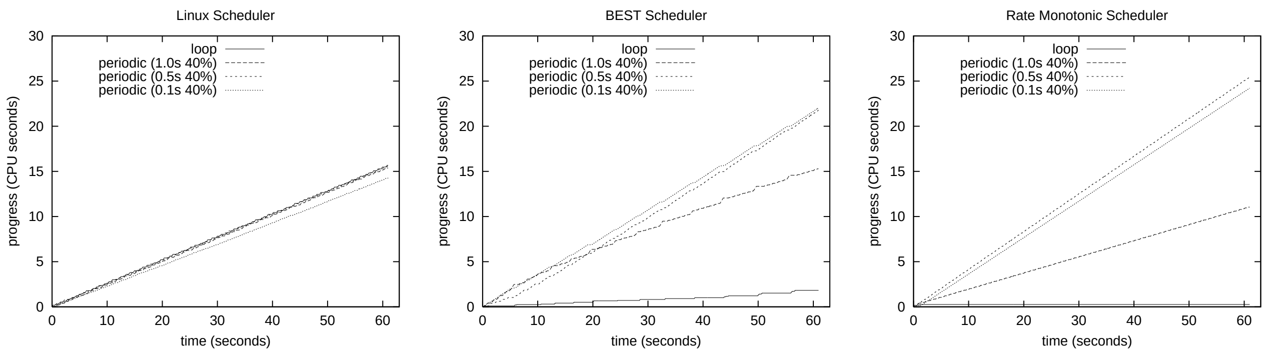


Figure 7. Linux, BEST, and RM schedulers with (1) loop (2) periodic 1s 40% (3) periodic 0.5s 40% and (4) periodic 0.1 40%.

Table 2: Summary of percentage of deadlines missed for all experiments.

		Scheduler		
Experiment	**Process**	**Linux**	**BEST**	**RM**
1	loop	-	-	-
	periodic(0.1s 40%)	0	0	0
2	periodic(0.1s 40%)	0	0	0
	periodic(0.1s 40%)	0	0	0
3	loop	-	-	-
	periodic(0.1s 70%)	29	0	0
4	loop	-	-	-
	periodic(1.0s 30%)	0	0	0
	periodic(0.1s 30%)	6	0	0
5	loop	-	-	-
	periodic(1.0s 30%)	83	0	0
	periodic(0.5s 30%)	80	0	0
	periodic(0.1s 30%)	10	0	0
6	loop	-	-	-
	periodic(0.61s 31%)	77	0	27
	periodic(0.43s 30%)	66	0	0
	periodic(0.13s 31%)	16	0	0
7	loop	-	-	-
	periodic(1.0s 40%)	98	71	98
	periodic(0.5s 40%)	93	5	0
	periodic(0.1s 40%)	19	2	0

processes while providing good best-effort performance. The previous figures show that BEST approximates the performance of both Linux and RM as appropriate. Table 2 summarizes the percentage of deadlines missed by each scheduler in all of the experiments. It shows that BEST meets or exceeds the performance of both the Linux and Rate Monotonic schedulers in each of the experiments shown.

A final question that is difficult to answer with data is the qualitative one—how does the scheduler perform in general use? To attempt to answer this question, we have been running the BEST scheduler (in Linux) on a desktop machine for the past few months. We have experienced no anomalous behavior, response time has been satisfactory and "normal," SRT processes definitely appear to run better, and BE processes do not starve while SRT processes are running. While this is not conclusive, it is quite encouraging.

5. FUTURE WORK

Experiments show that the BEST prototype meets its intended purpose: enhancing the performance of periodic processes while capturing the benefits of a best-effort model. It also preserves the design goals of general-purpose operating systems by favoring I/O-intensive over CPU-bound jobs; as users we successfully employ the scheduler on a development and general purpose platform with no adverse effect on responsiveness. However, many of the scheduler parameters are not tuned for executing real multimedia applications and workloads. For example, processes entering the system are currently initialized with scheduling parameters that make them appear periodic. Running a batch process (such as a compile) will generate several jobs that might interfere with already running processes; by determining proper initial values for process structures we can limit this effect. In the future we will tune BEST scheduler using the behavior of popular multimedia packages as examples (such as MpegTV and xmms). Many multimedia applications are multi-threaded and depend on the responsiveness of other processes such as the X server. In light of the dependencies of these applications, we must reexamine the naive assumptions of our test processes. For example, a more sophisticated version of the scheduler may detect dependencies and allow a process to inherit the deadline of related processes.

In addition, we believe that BEST has better jitter performance than the default Linux scheduler. We intend to characterize the jitter performance of BEST and compare it with the other schedulers examined in this paper. Another aspect of BEST that we have yet to examine is the effect of changing priorities. We believe that by automatically adjusting the priorities of the SRT processes we can adapt the system to provide any missed-deadline behavior desired. In particular, we should be able to spread out the missed deadlines across the SRT processes or minimize the missed deadlines of more important processes.

6. CONCLUSION

Standard best-effort schedulers make no resource guarantees, but soft real-time applications require some assurance of resource allocation in order to meet deadlines. Best-effort scheduling is thought to perform poorly for multimedia; but because it is simple to use, the best-effort model continues to be attractive for both application developers and users of general purpose systems. BEST is a CPU scheduler that adheres to a best-effort scheduling policy while automatically detecting and boosting the performance of periodic soft real-time processes. BEST dynamically determines application periods and schedules processes according to earliest deadline first, a well-known scheduler for real-time systems. However, unlike real-time schedulers, it uses simple heuristics to determine deadlines for both periodic and non-periodic processes.

This paper presents the design and implementation of the prototype BEST scheduler in the Linux kernel. It includes the results of a set of experiments demonstrating the scheduler's effectiveness at boosting the performance of processes with soft deadlines, while preserving desired characteristics of general purpose time-sharing schedulers. In particular, our results show that BEST performs as well as or better than the Linux scheduler and RM scheduling in handling best-effort, soft real-time, and a combination of the two types of processes. This holds true in situations of both processor underload and processor overload and is done with no *a priori* knowledge of the applications, their resource usage, or their periods. By continuing these experiments with more realistic workloads and by fine-tuning the scheduler parameters, we expect to develop the prototype into a full-fledged and robust system appropriate for widespread and general use.

ACKNOWLEDGMENTS

The work in this paper was motivated in part by a suggestion from Lonnie Welch that it would be interesting to see if we could dynamically detect application periods at run-time; the authors gratefully acknowledge that suggestion. We thank the reviewers for offering additional insight and ideas for future work. We also thank Zachary Peterson for reviewing drafts of this paper. This research was funded in part by a DOE High-Performance Computer Science Fellowship and a USENIX Student Research Grant.

REFERENCES

1. 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 Operationg System Support for Digital Audio and Video*, 1993.
2. S. Brandt, G. Nutt, T. Berk, and J. Mankovich, "A dynamic quality of service middleware agent for mediating application resource usage," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 307–317, Dec. 1998.
3. S. Brandt and G. Nutt, "Flexible soft real-time processing in middleware." *Real-Time Systems*, 2001. to appear.
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. 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.
6. 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.
7. M. Jones, J. B. III, and A. Forin, "An overview of the Rialto real-time architecture," in *Proceedings of the 7th ACM SIGOPS European Workshop*, pp. 249–256, Sept. 1996.
8. C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proceedings of the IEEE Internation Conference on Multimedia Computing and Systems*, pp. 90–99, May 1994.
9. J. Nieh and M. Lam, "The design, implementation and evaluation of SMART: A scheduler for multimedia applications," in *Proceedings of the Sixteenth Symposium on Operating System Principals*, Oct. 1997.
10. H. Tokuda, T. Nakajimi, and P. Rao, "Real-time Mach: Towards a predictable real-time system," in *Proceedings of USENIX Mach Workshop*, Oct. 1990.
11. 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.
12. A. Burns, "Scheduling hard real-time systems: A review," *Software Engineering Journal* **6**, pp. 116–128, May 1991.
13. E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time operating systems," in *Proceedings of the 6th IEEE Real-Time Systems Symposium*, Dec. 1985.
14. S. Khanna, M. Sebrée, and J. Zolnowsky, "Realtime scheduling in SunOS 5.0," in *USENIX Winter 1992 Technical Conference*, pp. 375–390, Jan. 1992.
15. IEEE, *IEEE Standard for Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application Programming Interface (API)-Amendment 1: Realtime Extension [C Language]*, Std1003.1b-1993 ed., 1994.
16. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprograming in a hard-real-time environment," *Journal of the Association for Computing Machinery* **20**, pp. 46–61, Jan. 1973.
17. I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," in *IEEE Journal on Selected Areas in Communications*, pp. 1280–1297, Sept. 1996.
18. B. Mullen, "The Multics scheduler." http://www.multicians.org/mult-sched.html, Aug. 1995.
19. V. Yodaiken and M. Barabanov, "Real-time Linux," in *Proceedings of Linux Applications Development and Deployment Conference (USELINUX)*, Jan. 1997.
20. H. hua 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.
21. C. han Lin, H. hua 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.

22. P. Goyal, X. Guo, and H. M. Vin, “A hierarchical CPU scheduler for multimeida operating systems,” in *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Oct. 1996.
23. B. Ford and S. Susarla, “CPU inheritance scheduling,” in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 91–105, Oct. 1996.
24. G. M. Candea and M. B. Jones, “Vassal: Loadable scheduler support for multi-policy scheduling,” in *Proceedings of the 2nd USENIX Windows NT Symposium*, pp. 157–166, Aug. 1998.
25. 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*, pp. 288–299, Dec. 1996.
26. C. A. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Massachusetts Institute of Technology, Sept. 1995.
27. D. K. Yau and S. S. Lam, “Adaptive rate-controlled scheduling for multimedia applications,” in *ACM Multimedia Conference*, Nov. 1996.
28. M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*, Addison Wesley Longman, 2nd ed., 1998.
29. M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Publishing, 1996.
30. J. A. Stankovic, M. Spuri, M. D. Natale, and G. Buttazo, “Implications of classical scheduling results for real-time systems,” *Computer* **28**, pp. 16–25, June 1995.