UNIVERSITY OF CALIFORNIA
SANTA CRUZ

USING THE BEST-EFFORT SCHEDULING MODEL TO SUPPORT SOFT REAL-TIME PROCESSING
OR
BEST DOES BETTER

A report submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Scott A. Banachowski

August 2002

Using the Best-effort Scheduling Model to Support Soft Real-time Processing

Scott A. Banachowski

Abstract

Algorithms for making timely CPU allocations to soft real-time processes exist, yet best-effort scheduling algorithms remain an attractive model for developers and users. Best-effort scheduling is easy to use, provides a reasonable trade-off between fairness and responsiveness, and imposes no overhead for specifying resource demands. However, best-effort schedulers provide no resource guarantees, limiting their ability to support processes with timeliness constraints. Reacting to a need for better support of soft real-time multimedia applications while recognizing that the best-effort model permeates desktop computing for good reason, we have developed a technique for inferring soft-real time behavior from executing processes. We have created two schedulers based on this technique, the Best-Effort scheduler enhanced for Soft Real-Time (BEST) and a rate-controlling scheduler (BEST-RATE), that combine desirable aspects of both best-effort and soft real-time scheduling. BEST provides the well-behaved default characteristics of best-effort schedulers while significantly improving performance of periodic soft real-time processes. BEST schedules using estimated deadlines based on the dynamically detected periods of processes, and assigns pseudo-periods to non-periodic processes to allow for good response time. BEST-RATE, developed to overcome lack of fairness in BEST, provides timely allocation to periodic processes while providing equal resource allocation among all processes. This paper discusses a Linux implementation of the BEST and BEST-RATE schedulers and presents results demonstrating that they outperform the Linux scheduler in handling soft real-time processes, and outperform real-time schedulers in handling mixed workloads of soft real-time and best-effort processes.

Acknowledgments

I thank Scott Brandt for his advice, mentoring, friendship, and his displays of “Midwest heroism.” I also thank Darrell Long, Ethan Miller, and the UCSC Computer Systems Lab for their support over the last two years, especially Zachary Peterson for reviewing drafts of this paper and co-conspiring to make the lab fun. I am especially grateful for the technical discussions of this research with Hermann Härtig at the Dresden University of Technology, Lonnie Welch at the University of Ohio, and their respective research groups, who contributed valuable feedback. Randal Burns provided me with insight into the world of research. On a personal note Denise Lum and my parents contributed to this work, if not in the technical details, then in spirit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Schedulers for conventional desktop operating systems use a *best-effort* policy. The goal of a best-effort scheduler is to provide adequate progress and fairness to applications by distributing CPU bandwidth and scheduling latency equally among competing processes. As the name "best-effort" implies, the scheduler provides no facilities for making or meeting CPU allocation reservations for processes. Because best-effort schedulers provide no guarantees of CPU bandwidth, processes with timeliness constraints may or may not receive the timely allocation of CPU required to meet deadlines.

In the best-effort model, both latency and progress of an application depend on the processor load. A scheduler that allocates CPU time-slices to processes in a round-robin fashion is an example of a best-effort policy; once a process executes for a time-slice, it must wait the duration of the combined time-slices of all other processes before proceeding. Multi-level feedback queues, commonly used in both UNIX-based and Windows systems [12, 26], improve on round-robin by first servicing processes that incurred I/O delay, effectively trading their scheduling latency for I/O latency. Although this strategy helps balance the fairness of overall latency, it does not ensure that

a process will meet any particular timeliness goals.

Conventional desktop operating systems do not directly support the scheduling needs of real-time applications because they use the best-effort scheduling model. The model is attractive because it is simple and easy to use; applications do not require special interfaces to the operating system for reserving CPU bandwidth, and the system need not incorporate admission control or service guarantees. Clearly a desktop operating system is not a desirable platform for hard or firm real-time applications, in which a failure to meet a deadline is considered a failure of the system. However, a soft real-time application does not require all deadlines to be met; a missed deadline leads to undesirable degradation of performance, but not catastrophic failure.

The popularity of multimedia applications such as audio and video players for desktop workstations is a testament to the capability of best-effort systems to satisfactorily execute soft real-time applications. However without service guarantees, the performance of soft real-time applications degrades in the presence of scheduling latency. In order to eliminate performance problems, the user is left adjusting process priorities (via commands like *nice* that scale time-slices), or setting static real-time priorities. Scaling time-slices reduces the impact of scheduling latency but does not eliminate it, and is intrusive because it requires intervention from the user. Using static real-time priorities is inadequate for handling continuous sound or video [29]; it violates the best-effort model causing 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 as on a desktop system.

The result of research in desktop soft real-time schedulers is several systems that provide support for mixed workloads consisting of multimedia applications and applications without timeliness requirements (best-effort processes), discussed further in Section 1.1. These systems require

applications to interface with special-purpose soft real-time routines to provide runtime parameters such as resource usage and periodic deadline. Although they address lack of timeliness, they lose one primary advantage of the best-effort model by requiring *a priori* specifications of application resource and timeliness needs. The interface to the scheduler is exposed, meaning that either the application programmers or the users must interact and negotiate with the scheduler to control the scheduling policies. By doing so, the programming or run-time model loses generality, restricting the portability of applications. In addition, it is often difficult to characterize the resource needs of applications in advance because performance may vary on different systems.

In this report, we consider using the best-effort model to support applications with periodic deadlines. Previous research on the Dynamic QoS Level Resource Management (DQM) system demonstrates that it is possible to robustly execute soft real-time applications on best-effort systems [6, 7, 8]. This system uses a middleware framework that allows 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.

Like other soft real-time systems, the DQM system has several issues that limit its ultimate utility in generic desktop environments. Although the DQM system dynamically adjusts to incorrect or unspecified resource usage estimates, it cannot adapt to incorrect or unspecified application periods. 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.

This report presents a solution that addresses those issues—BEST, the **B**est-effort scheduler **E**nhanced for **S**oft real-time **T**ime-sharing. BEST is a time-sharing scheduler that directly

supports multimedia applications while providing adequate progress and response time for best-effort applications. 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. BEST dynamically measures process behavior and uses this 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.

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 [32]. 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 report describes an implementation of the BEST scheduler in the Linux kernel and a variation of the technique, called BEST-RATE, that monitors and controls the rates of running processes. Chapter 2 describes some properties of soft real-time multimedia applications that we model to create synthetic workloads for testing the scheduler. Chapter 3 describes the BEST scheduler implementation, and presents quantitative performance data. Chapter 4 presents the BEST-RATE scheduler implementation and its performance. Finally, Chapter 5 gives conclusions with comments on open areas for future research. The remainder of this chapter discusses previous research on systems that support scheduling for soft real-time applications.

1.1 Related Work

Continuous real-time applications require enough processor bandwidth to meet their periodic deadlines [9]. 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 [20].

1.1.1 Traditional Real-time Scheduling

Real-time systems, such as RT-Mach [35], are designed to meet hard deadline constraints. Some versions of UNIX support real-time scheduling classes [21], and many systems adapt the POSIX standard for real-time extensions [18]. In order to ensure predictable behavior, these systems use strict scheduling policies such as Rate Monotonic (RM) or Earliest Deadline First (EDF) [24]. 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, that uses EDF based on deadlines derived from a process's specified share of CPU bandwidth [22]. Multics also provides an EDF scheduler, using desired response time to determine virtual deadlines of non-real-time processes [28]. 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, while assigning pseudo-deadlines to non-periodic processes, and schedules accordingly.

1.1.2 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 [39], 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 [13, 23], 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 [16], and CPU Inheritance Scheduling [15], which allows scheduling threads to donate processing to other scheduler threads in flexible arbitrary arrangements of hierarchies. The Vassal project [12] adds a system interface that allows users to install their own schedulers. Another method applied to soft real-time is middleware resource management[1]. Middleware managers monitor a system's resources usage, and provide recommendations to adaptive soft real-time processes. DQM [6]

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

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.

1.1.3 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 [34] calculates a virtual deadline for each process as a function of measured and allotted share, and schedules according to EDF; Stride Scheduling [36] uses a similar notion of virtual time. Systems such as BVT [14]

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 [27]. The CM [38] and SMART [30] 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 [19] (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. BEST-RATE uses techniques similar to proportional schedulers by generating deadlines from process rates. Like BEST, BEST-RATE does not need information about rates; it looks at the past behavior of processes to infer their execution pattern, and assigns future allocations that reduce relative scheduling latency for those with periodic deadlines.

1.1.4 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.

Chapter 2

Multimedia Properties

This chapter describes an application, called *srtsim*, that simulates properties of processes with periodic deadlines. The simulator was developed as part of the BEST research project to provide test workloads for multimedia, soft real-time or real-time schedulers. Like many synthetic workloads used for experimentation, the application uses statistical information gathered from actual workloads to generate realistic behavior. We examined the source code of several multimedia applications, and measured some statistical information about MPEG video streams to determine feasible simulator behavior and workload generation. The goal of this chapter is to present properties of multimedia formats and applications that may be useful for others needing to model systems that run multimedia workloads.

In order to test the performance of a system, it is often desirable to drive it with inputs generated from synthetic workloads. When testing schedulers, typical metrics for evaluating performance include throughput and fairness. However for real-time schedulers, timeliness of resource allocation is most important. In order to create test scenarios for a desktop-based soft real-time scheduler, we need applications with periodic deadlines to simulate workloads. The

purpose of this simulator, to mimic the variability of frame-to-frame processing time inherent in multimedia, differs from other real-time workload generators such as Hourglass [31], which is intended for making fine-grain timing measurements of scheduler performance.

2.1 Multimedia Properties

To test a scheduler we found it easier to write an application that simulates a process with periodic deadlines rather than use real multimedia applications. This way, the simulator may create many different workloads without needing to rely on multimedia streams to drive it; simulation affords the flexibility to vary the periodic deadline and the amount of processing for each instance of the simulator. Using a simulator also allows easy instrumentation to measure aspects of performance, such as which deadlines were missed.

To make the simulator behave like actual multimedia applications, we observed the behavior of several Linux applications to determine how they synchronized frames and what happens when a deadline is missed. We also measured statistics of many MPEG movies to extract properties of the video media useful for generating realistic workloads.

2.1.1 Soft real-time models

We examined two single-threaded public-domain multimedia applications developed for UNIX systems: a sound player called *mpg123* and a video player called *mpeg_play*. We examined the source code and executed these applications to determine their behavior. We do not include other software models, such as those that synchronize video and audio during playback, and multi-threaded applications that provide other services (such as a graphical equalizer displays) in real-

Table 2.1: Average period, standard deviation, and CPU usage for sample multimedia processes.

Process	Average period (ms)	Standard deviation	CPU usage
mpeg_play (24 frame/s) (no display)	42.3	9.2	13.0%
mpeg_play (24 frame/s)	21.7	21.9	19.3%
mpeg_play (30 frame/s) (no display)	34.6	13.1	15.9%
mpeg_play (30 frame/s)	17.0	17.4	19.5%
mpg123 (128 kbit/s) (song 1)	160.2	31.7	2.2%
mpg123 (128 kbit/s) (song 2)	160.2	31.8	2.2%

time while playing media, in this study.

The sound player has periodic deadlines, although the software does not adjust its run-time behavior to adapt to the media; it works by simply keeping the memory used as audio buffers supplied with data. In the main loop of the program, the program synchronizes by making blocking *write* calls to the sound driver in order to play audio samples. When the driver accepts the new frame of data, the next frame is processed.

The video player software is aware of its periodic deadline, which it determines from the frame rate of a video stream. It uses functions provided by the operating system to synchronize the display of frames to its deadline. Before displaying a frame, the player pauses by using the *select* call if there exists spare time before the next frame is due. This way the video player displays frames at the desired frame rate. When there is no spare time, i.e. the frame deadline is missed, the player resets its next deadline to begin after the late frame is displayed. This way, a late frame does not reduce the available processing time of future frames. The delay will degrade the overall frame rate, but as deadlines are soft for desktop media players, this behavior is not considered failure as it is with hard deadlines.

The BEST scheduler works under the assumption that periodic deadlines are detectable

by the operating system kernel. In order to test this assumption we instrumented the Linux kernel to record entry times to the run queue to the nearest $\frac{1}{800}$ of a second, and then ran the multimedia processes described above. These experiments were executed on a 650 MHz AMD i686 system. We measured a period of approximately 0.16 seconds when playing sound files (Table 2.1), and for the video files a period corresponding to the frame rate. Table 2.1 shows the measured periods.

When *mpeg_play* displays frames, it actually 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, usually during the same clock tick, the detected average period is half the actual frame period, and the standard deviation is close to the average period. 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 is equivalent to its frame rate. The audio player is not driven by a frame-rate 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.

2.1.2 Video-stream properties

The sound player model is simple: the audio device is supplied with a constant stream of data which it buffers. Deadlines are not critical for the sound application as long as it supplies data at a near-constant rate. The video properties are more interesting, because a frame must be processed and displayed at a constant rate, and the application requires synchronization from the operating system.

Multimedia is usually transfered and stored in a compressed format, such as MPEG. Due to compression, the amount of processing for each frame varies. The compression may be

spacial (within a frame) or temporal (using information across frames). MPEG video data includes three types of frames: I-frames (intra-frames) contain information to decode a single stand-alone frame, while P-frames (predicted frames) and B-frames (bi-directional frames) need to reference information from previous frames, or from previous and future frames, respectively, in order to decode an image. The presence of different frame types makes it difficult to predict the amount of processing needed by a video stream from frame-to-frame.

We examined several MPEG files using *mpeg_play* to capture statistics about the video streams and to ascertain useful video properties for our own modeling. To generate a simulated video stream, one should only need to specify a periodic deadline and average rate (or processing time) of CPU consumption. Using these parameters, the simulator may then generate frames exhibiting the same frame-to-frame statistical variability observed in actual video streams. This allows easy scaling of frame rate and average CPU usage while maintaining realistic video properties. We measured the variance of each frame processing time relative to the overall average. We also determined the breakdown of frame-types in a stream, and the average frame processing time of each frame relative to the overall average. We find the statistical properties of the media frames to be consistent with other studies [17]

Three MPEG files we observed consisted only of I-Frames; Table 2.2 summarizes the statistics we measured. We looked at seven MPEG files that consist of all frame types; Table 2.3 summarizes their statistics. It includes the breakdown of the number and percentage of each frame-type. Over all frames and each frame type, the table shows the standard deviation and variance of processing times. It also shows the relative scale of each frame-type, which is the average processing time for the type divided by the average processing time of all frames. Figure 2.1 shows the frame processing time of one of the videos (file G, a scene from the Hollywood blockbuster

Table 2.2: Summary of I-Frame only MPEG statistics.

File	Number frames	Average time consumed (ms)	Standard deviation	Percent variance
A	253	1.03	0.107	10.3%
B	18	1.29	0.039	3.1%
C	557	0.91	0.115	12.6%

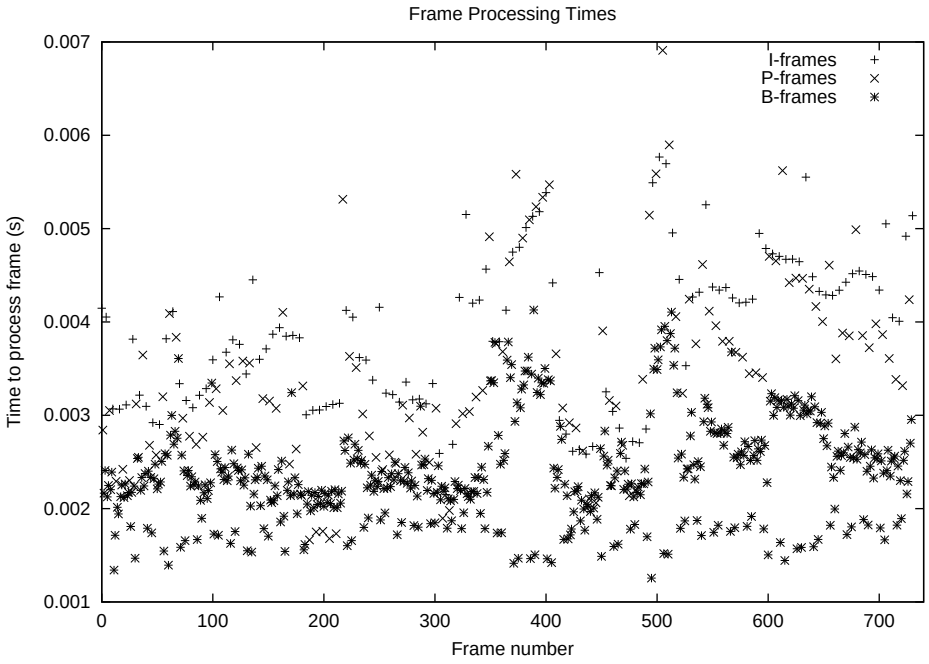


Figure 2.1: For each MPEG frame of a video (action movie excerpt), this graph shows the processing time of the frame.

(*Under Siege*) on a frame-by-frame basis, and Figure 2.2 is a histogram of the frame-processing time distribution.

2.2 Workload Simulator Implementation

In this section we describe *srtsim*, the UNIX soft real-time application simulator based on the patterns characterized in Section 2.1.2. The simulator is an application with periodic dead-

Table 2.3: Summary of MPEG with I,P and B statistics.

File	Frame Type	Number frames	Percent of total	Average time consumed (ms)	Standard deviation	Percent variance	Relative scale
D		1758	–	0.71	1.27	178%	–
	I	294	16.7%	1.33	2.36	176%	1.87
	P	293	16.7%	1.16	1.79	153%	1.64
	B	1171	66.7%	0.44	0.08	18.1%	0.62
E		720	–	4.06	1.26	31.0 %	–
	I	121	16.8%	6.46	0.46	7.2%	1.59
	P	120	16.7%	4.84	0.40	8.2%	1.19
	B	479	66.5%	3.25	0.12	3.8%	0.92
F		1210	–	2.84	1.09	83.2%	–
	I	41	3.4%	5.78	0.75	13.0%	2.03
	P	81	6.7%	4.25	1.60	37.7%	1.50
	B	1088	89.9%	2.62	0.76	29.0%	0.92
G		731	–	2.83	0.92	32.7%	–
	I	123	16.8%	3.90	0.81	20.7%	1.38
	P	122	16.7%	3.49	1.00	28.8%	1.23
	B	486	66.5%	2.39	0.53	22.1%	0.85
H		6299	–	2.08	0.58	27.7%	–
	I	420	6.7%	3.19	0.27	8.6%	1.53
	P	1681	26.7%	2.70	0.23	8.4%	1.30
	B	4198	66.7%	1.72	0.26	15.2%	0.83
I		625	–	2.59	0.66	25.4%	–
	I	36	5.8%	4.06	0.27	6.8%	1.56
	P	174	27.8%	3.31	0.23	7.0%	1.28
	B	415	66.4%	2.17	0.20	9.4%	0.83
J		901	–	2.19	0.50	23.0%	–
	I	51	5.7%	3.17	0.35	11.1%	1.45
	P	250	27.7%	2.76	0.21	7.7%	1.26
	B	600	66.6%	1.87	0.19	9.9%	0.85

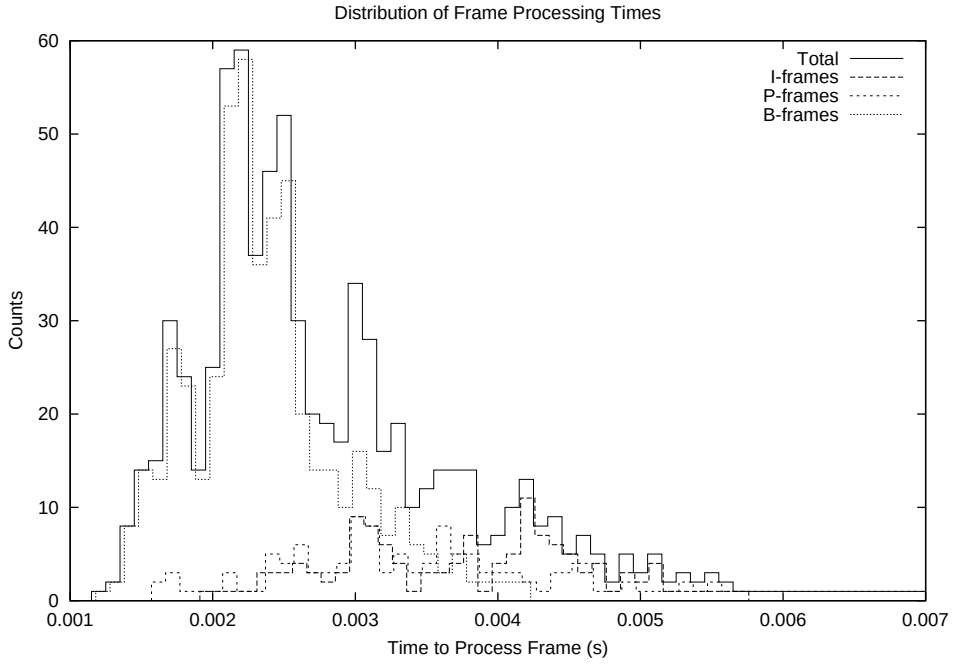


Figure 2.2: This histogram shows the distribution of processing times for each frame type of the data shown in Figure 2.1.

lines used to generate workloads for evaluating operating system schedulers. Using command line arguments, it may run in several different modes and allows the user to specify the period of deadlines and rate of CPU consumption. It generates workloads based on our evaluation of real video streams and behaves like a multimedia process, as described in Section 2.1.

2.2.1 Settable parameters

When executing *srtsim*, the user must specify parameters on the command line. The program requires, at minimum, the length of the periodic deadline and rate of CPU consumption. In the default mode, the process strictly consumes the same amount of CPU during each deadline period. When the process completes its computation, it goes to sleep using the *pause* command, using a timer to wake when the next period begins. When a deadline is missed, the process com-

Table 2.4: Statistics used to generate synthetic video streams.

Parameter	**Value**
periodic deadline	user specified
ave. frame time	user specified
frame distribution (I)	16.6%
frame distribution (P)	16.6%
frame distribution (B)	66.6%
ave. I frame time	1.5 $\times$ ave. frame time
ave. P frame time	1.25 $\times$ ave. frame time
ave. P frame time	0.85 $\times$ ave. frame time
std. deviation	10% of frame time

pletes the periodic computation, and like the video player, immediately begins the next period with the new deadline set relative to the frame start time. Optionally, the user may tell the simulator to instead abort computations when a deadline is missed, which is useful for modeling other types of processes such as those with firm deadlines.

The user may optionally tell *srtsim* to create video frame-time distributions. In this mode, instead of consuming the same amount of CPU during each period, the time distribution is modeled after MPEG video streams. It simulates movies either containing only I frames, or containing all (I, P, and B) frame types. Table 2.4 contains the parameters used to generate the frame distribution and times. If all frame types are present, then the frame distributions statistically approximate the values representative of several of the movies we observed (Table 2.3). The average processing time for each frame type is based on values close to the mean of observed values; since there is a wide variation in standard deviation in each video, and we chose parameters that seemed reasonable across all the observed videos.

The simulator randomly generates frame times while executing. For each period, it first selects the frame type by choosing from a uniform distribution shown in Table 2.4. Then, for each

Table 2.5: Summary synthetic MPEG video stream.

File	Frame Type	Number frames	Percent of total	Average time consumed (ms)	Standard deviation	Percent variance	Relative scale
Synth		731	–	2.84	0.76	26.6%	–
	I	102	14.0%	4.22	0.42	9.9%	1.49
	P	120	16.4%	3.53	0.37	10.5%	1.24
	B	509	69.6%	2.40	0.25	10.4%	0.84

frame type, the average frame time and standard deviation are used to generate the frame time from a normal distribution, as observed in the histogram plots (Figure 2.2).

Figures 2.3 and 2.4 show the frame distributions generated for a synthetic video stream that has the same average frame time as movie G from Section 2.1. When comparing these figures to Figures 2.1 and 2.2, the synthetic video stream appears to do an adequate job of modeling the actual video. Table 2.5 summarizes the measured statistics of the synthetic video stream.

2.2.2 Simulator output

For each frame, the simulator records the amount of time is spent processing, the time the period started and ended, and whether it met its deadline. These statistics are logged while the program is running, and then printed to the screen when the simulation is completed. The traces may be used to analyze the performance of the scheduler offline. We use this output to evaluate the BEST schedulers in Chapter 3 and Chapter 4.

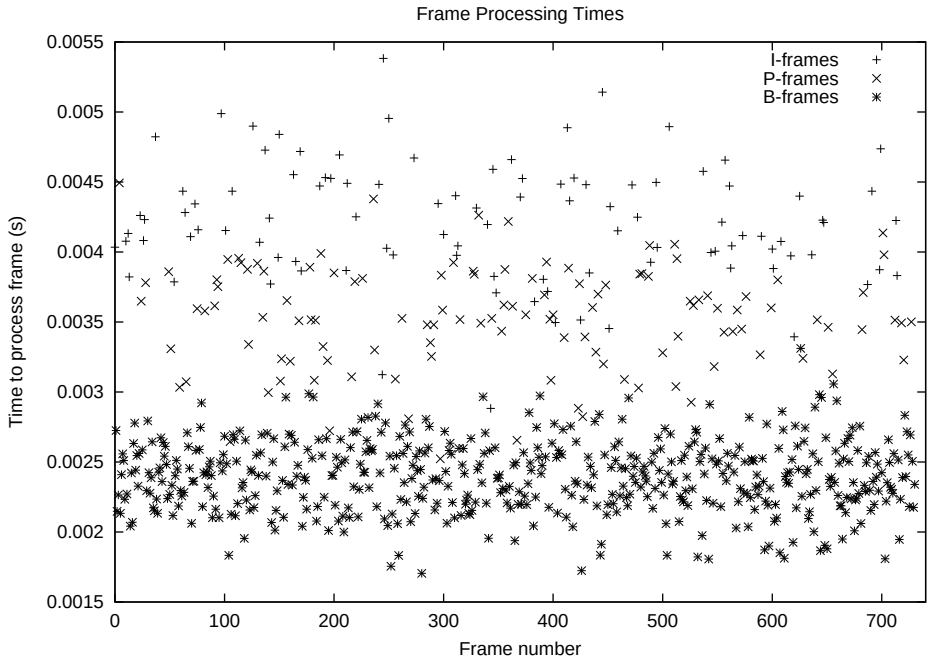


Figure 2.3: For each synthetically generated frame, this graph shows the processing time.

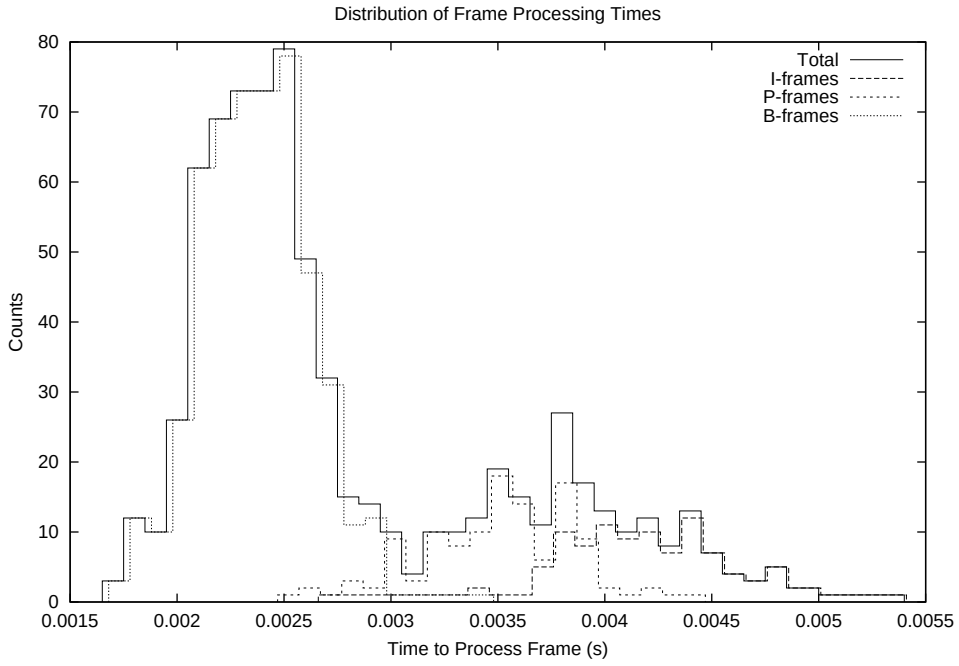


Figure 2.4: This histogram shows the distribution of processing times for each frame type of the data shown in Figure 2.3.

Chapter 3

The BEST Scheduler

The goal of the BEST scheduler is to enhance the performance of soft real-time tasks by detecting periodic processes and improving their chances of meeting periodic deadlines. Like most UNIX schedulers [5, 26], 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 their 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.

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 need to provide any *a priori* information about processes.

3. The scheduler will enhance the performance of soft real-time applications.

    - Processes that enter the runnable queue in predictable patterns should receive a performance boost by receiving a higher priority based on classical real-time scheduling results, i.e. the priority boost should be based on measured 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 will have high priority upon waking, 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 soft real-time processes. For fully loaded systems, performance should degrade gracefully.

3.2 Linux Scheduler Overview

We implemented the BEST scheduling algorithm in both the Linux 2.2 and 2.4 kernels. 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 modifications we made.

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 process state structures (including those not in the run queue), recomputing 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 remains suspended over n calculations, the priority 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.* [26]), the goal of the scheduling policy remains the same: favor I/O-bound over CPU-intensive processes while allowing no process to starve and providing equal share of the CPU. 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 [12].

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 `goodness()` function returns a value such that `schedule()` 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}, calculated 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 the process's next deadline by adding its effective period to the current time; it assumes that period and deadline is synonymous. The scheduler uses the deadline as a priority when selecting runnable processes. This function also sets an additional value called the *deadline expiration timer*. This timer indicates how much CPU time a process may consume before its deadline is reset. Like a quantum timer in Linux, the expiration timer value is decremented for every tick the process executes. The deadline and expiration timer values are stored in the process's state structure.

Every time a process is scheduled for the CPU it executes until either its deadline expires, it blocks, or it is preempted by a waking process, at which time `schedule()` is triggered. Before selecting the next process for execution, if the current process's deadline 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. 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. The deadline expiration timer for non-periodic processes is set to the Linux quantum length (which is scaled by the *nice* priority). By making their deadlines expire in the same amount of time as a Linux quantum, best-effort processes in BEST behave similar as in the Linux scheduler.

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 normalized value between 0 and 16). A process is "well-behaved" if its confidence exceeds a threshold. A process that is not well-behaved receives a best-effort deadline, and expiration timer equal to its Linux time-slice quantum.

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 2.1, 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 frame 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 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.

Because the deadline expiration timer is also treated as a interval timer, it limits the time a process may hold the CPU, similar to the interval timers found is real-time systems such as Nemesis [22]. In Linux, the default processing time quantum is 0.1 seconds,[1], which is historically the quantum used in BSD as it provides an ideal responsiveness for interactive processes [26]. The quantum may be modified between 0.005 and 0.20 seconds by using the UNIX *nice* facility. In BEST the default processing quantum is also set to 0.1 second, and scalable within the same range using *nice*. In BEST, a running process may be preempted by one with an earlier deadline; however, similar to Linux, the number of context switches are reduced by disabling preemption when a current quantum expires in less than 10 milliseconds.

The overall changes to Linux include: adding entries to each process's state structure (for keeping track of periods), instrumenting `wake_up_process()` to measure period, and changing

[1] It was 0.2 seconds in versions of Linux < 2.4.

`goodness()` to return a priority appropriate for the BEST scheduler. The changes do effect the complexity of the scheduling selection algorithm which is $\Omega(n)$ with the number of running processes, however BEST incurs some computation overhead every time a process wakes.

3.3.4 Tuning the scheduler

Several parameters affect the behavior of the BEST scheduler: the maximum period, 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 averaging weight and confidence threshold impact the effectiveness of the period detection algorithm.

For the BEST prototype described in Section 3.4, we use a maximum period of 2.56 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 greater than 0 to pass. These defaults work well in our experiments where all processes were our synthetic load generators and we found that changing their values had little impact; we expect these tuning parameters to be more significant when stressed by less "well-behaved" workloads.

3.4 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 that the calculation of priorities for RM scheduling requires knowledge of application periods while the default Linux scheduler and BEST do not.

While running combinations of greedy (CPU-intensive) and periodic (soft real-time) processes together for 100 seconds, we measured the throughput of all processes and the number of missed deadlines for periodic processes. Two synthetic applications were used in the experiments. The processes called *best-effort* endlessly consumes CPU bandwidth by crunching math operations. This process is an adversary to soft-real time programs, because it creates load for the CPU and always remains in the runnable queue. The soft real-time application, *srtsim*, is described in Chapter 2. We characterize it using two parameters, a frame rate and a percentage; the deadline for each frame is the inverse of frame rate, and the percentage is the average amount of CPU bandwidth it must consume, on average, for each frame in order to meet the deadline. If *srtsim* 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.

3.4.1 The Linux scheduler simulator

The original performance of BEST was evaluated using experiments conducted on a 200 MHz Pentium Pro, and these results were presented in a previous report [3]. The following experiments were run on *schedsim*, a Linux kernel scheduler simulator we developed to test new algorithms. The simulator consists of Linux 2.4 scheduler code with software wrappers that control events external to the scheduler module, such as the arrival or sleeping of processes. The simulator executes the dummy *best-effort* and *srtsim* processes within its own user-space memory. The simulated environment makes the same decisions as the actual Linux and BEST scheduler, while

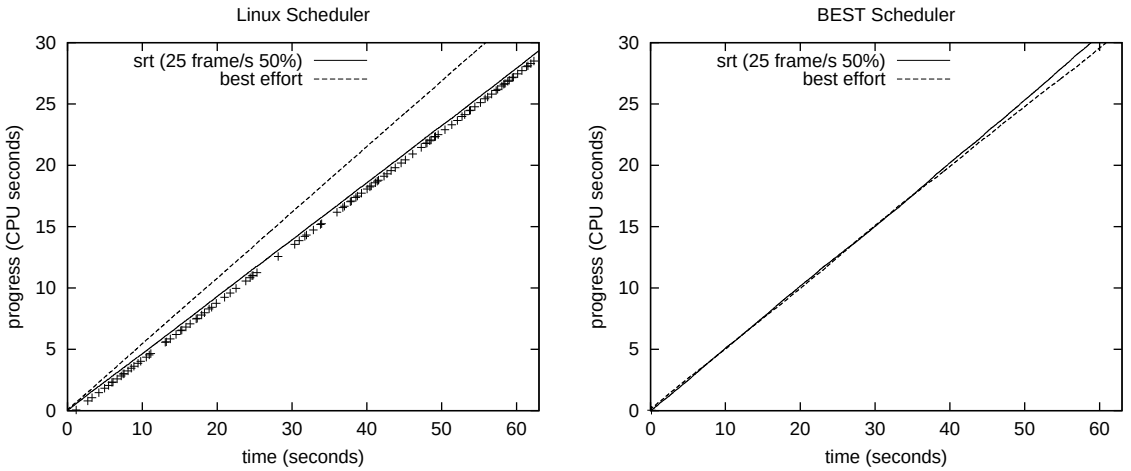
allowing testing of new scheduler algorithms without the need to build and install entire kernels or reboot processors. The original tests were run on an otherwise quiescent system, so other tasks did not interfere with the results of the processes under test, but on a desktop systems there may be page fault or interrupts that effect the processing. However in *schedsim*, there are no external influences as may appear on an actual desktop environment.

3.4.2 Interpretation of results

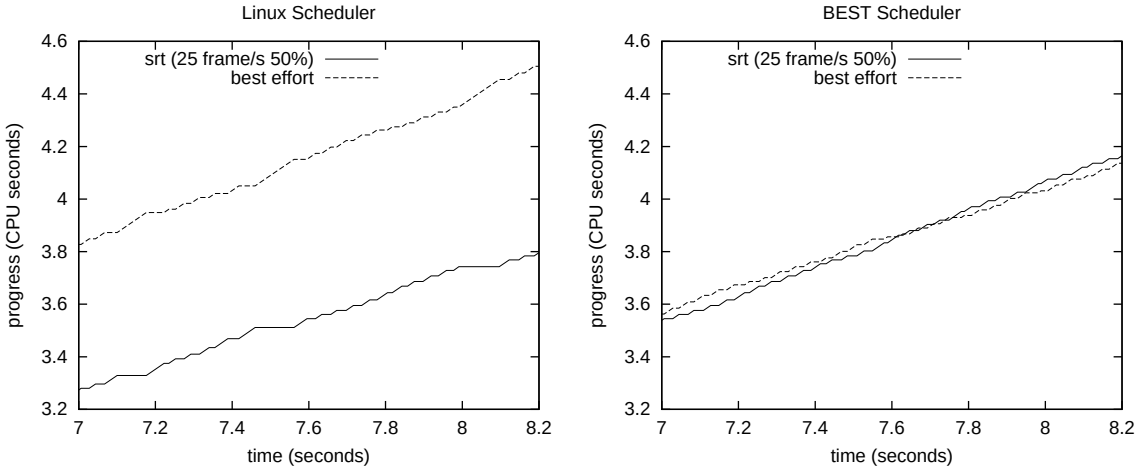
Our results show that in general the Linux scheduler performs reasonably well when the total demand of soft real-time processes is less than 100% of the CPU and a process i requires no more than than $(pri_i / \sum_{x \in n} pri_x)$ of the CPU, where n is the set of running processes, and pri_x is the nice priority of x (a value between 1 and 40). This value is often $1/n$, when users do not change the default nice priorities of the running processes. As the needs of an SRT process approach this limit, the scheduler is less effective at ensuring the process meets deadlines, because it may service processes in arbitrary order. Because time-share scheduling algorithms are unaware of resource requirements or deadlines, well-intentioned scheduling decisions can result in some processes missing deadlines that could otherwise be met.

Figure 3.1(a) shows the performance of the Linux scheduler and the BEST scheduler with one best-effort process and one SRT process (with a 25 frames/sec rate and CPU requirements averaging 50% of the CPU). The graphs show the progress of each process, with actual time on the X-axis and the CPU-time consumed by the process on the Y-axis. For example, if the progress of a process has a slope of 0.3, then the process is receiving approximately a third of the CPU resource. A missed deadline is indicated by marking a small cross below the line.

Because the Linux scheduler provides approximately equal amounts of CPU cycles to



(a) Linux and BEST schedulers.



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

Figure 3.1: Linux and BEST schedulers running (1) best-effort and (2) srtsim 25fps 50%.

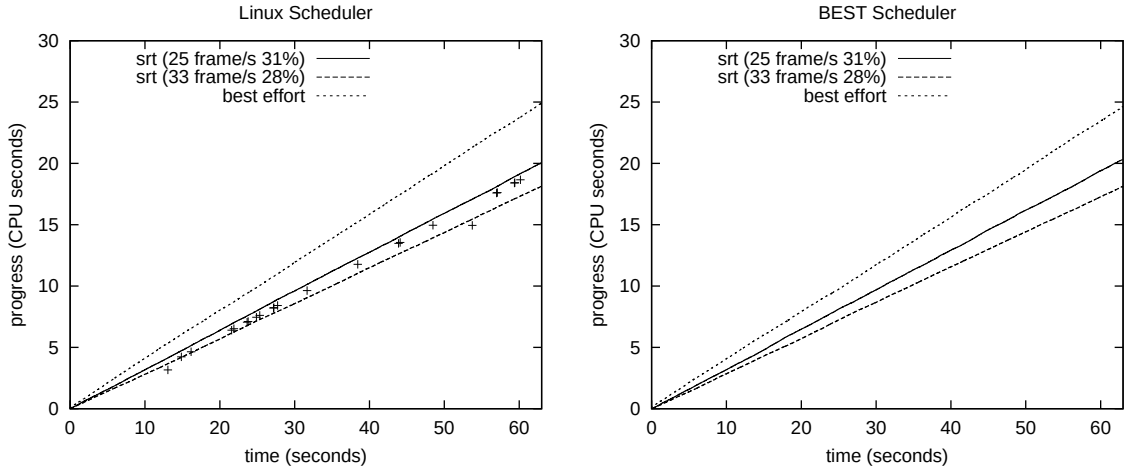


Figure 3.2: Linux and BEST schedulers with (1) best-effort, (2) SRT 25fps 31% and (3) srtsim 33fps 28%.

each application, the SRT process should be able to meet its deadlines. However, we observe the SRT process missing 7.8% of its deadlines. Although the process receives enough CPU allocation, for many frames it does not receive the CPU in time to make its deadline. The BEST scheduler met all but 2 deadlines ($< 1\%$) while providing the same amount of resources to each application. We found that with one SRT and one best-effort process, we must reduce the average usage of *srtsim* to below 40% before the Linux scheduler could meet performance of the BEST scheduler.

Figure 3.1(b) shows a magnified view of a portion of the data from Figure 3.1(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 SRT process doesn't always receive CPU at the even intervals required to meet its periodic deadlines. Under the BEST scheduler, the SRT process receives CPU allocation at evenly proportioned intervals equal to its period, and does not miss deadlines.

In Figure 3.2, there are two SRT processes, both which require approximately a third of the CPU to meet deadlines. With another best-effort process present, Linux will allocate up to

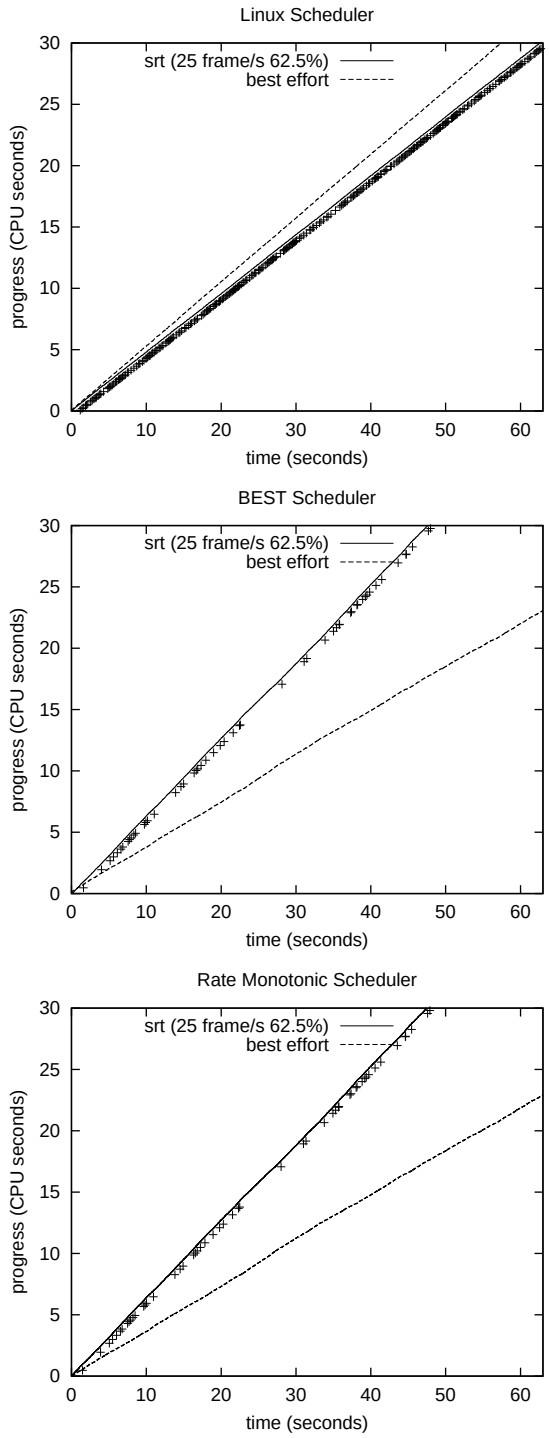


Figure 3.3: Linux, BEST and RM schedulers running (1) best-effort and (2) srtsim 25fps 62.5%.

$\frac{1}{3}$ CPU to each. In this case, Linux fares well, missing only 1.7% and 0.1% of deadlines for the two respective SRT processes. However, BEST does better, missing only one deadline of each task ($< 0.1\%$). It is interesting to note that in this case, the BEST scheduler slightly outperformed the Rate Monotonic scheduler, which missed 0.5% of the deadlines of the lower priority (longer period) task.

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 available CPU cycles (with slightly more given to processes that block occasionally). Figure 3.3 shows the performance of the Linux, BEST, and RM schedulers with one best-effort process and one SRT process (with 25 frames/sec and 62.5% CPU usage). Here we see that the Linux scheduler provides approximately 50% of the CPU cycles to each process, causing the SRT process to miss 25.5% of its deadlines. By contrast, the RM scheduler (with the SRT process having higher priority) provides the SRT process with 62.5% of the available cycles, enabling it to meet all but 4.8% of its deadlines. When generating a high average usage, the statistically driven *srtsim* workload will sometimes generate a frame that takes longer than the period to process; in this example, the SRT process missed only the deadlines that were impossible to make. The BEST scheduler provides exactly the same performance as the RM scheduler, enabling the SRT process to meet all but the impossible deadlines. Recall, however, that BEST dynamically determines the application periods whereas Rate Monotonic requires that periods be specified in order to determine appropriate priorities.

Any set of soft-real time processes with a CPU requirement less than 100% is theoretically schedulable without missed deadlines, provided that the scheduler may shed load of non

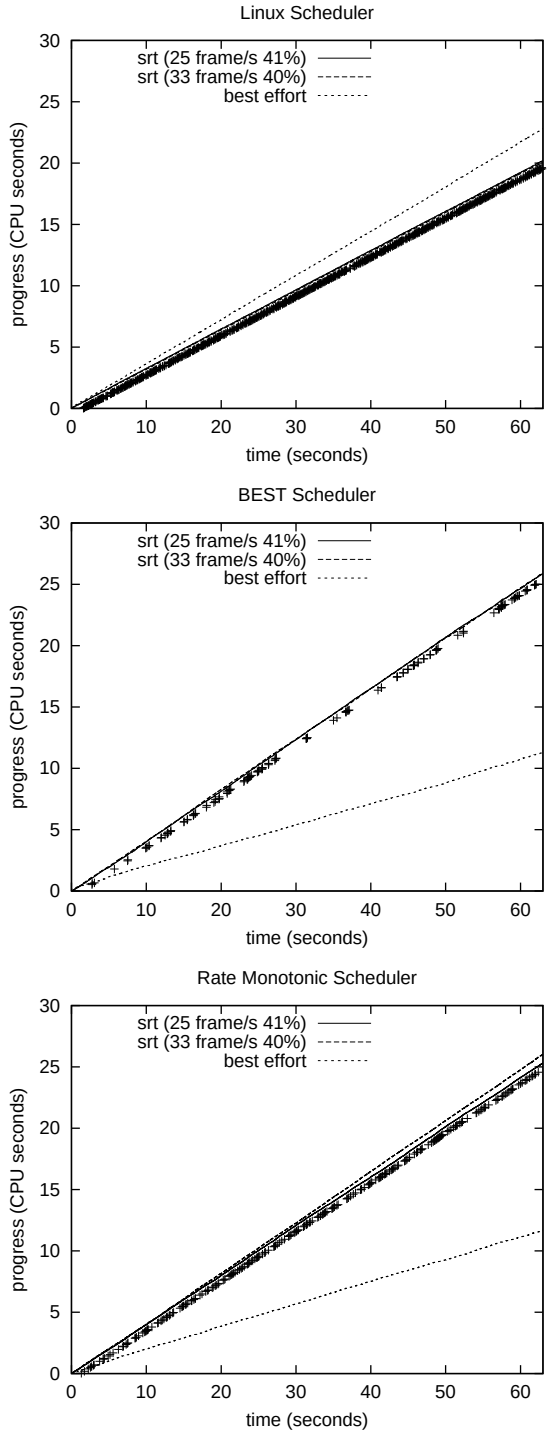


Figure 3.4: Linux, BEST and RM schedulers running (1) best-effort, (2) srtsim 25fps 41%, and (3) srtsim 33fps 40%.

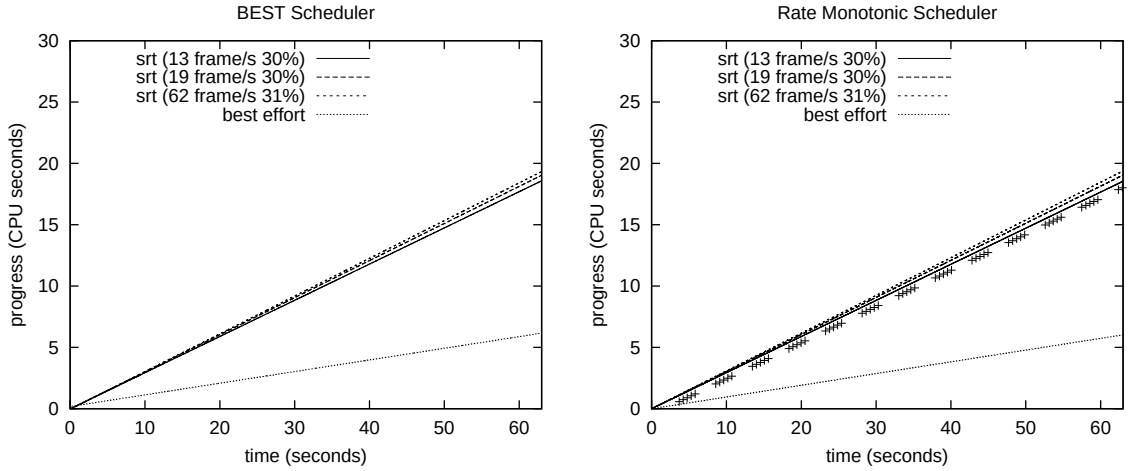


Figure 3.5: BEST and RM schedulers running (1) srtsim 13fps 30%, (2) srtsim 19fps 30%, and (3) srtsim 62fps 31%.

real-time processes. Figure 3.4 shows the Linux, BEST and RM schedulers with three processes, one best-effort and two SRT, one processing 25 frames/second and requiring an average of 41% of the CPU, and one processing 33 frames/second and requiring 40% of the CPU. In this situation, a time-share scheduler cannot allocate enough share to SRT processes due to competition from the loads of other non-SRT processes. The Linux scheduler cannot meet all deadlines because it allocates roughly $\frac{1}{3}$ of the resources to each process, and it performs poorly, missing 26.8% and 24.5% of the deadlines of periodic processes, respectively. Under the same workload, the BEST scheduler misses 4.6% and 3.3% of deadlines, while the RM scheduler misses 16.7% and 0%. In this case, BEST donated CPU cycles of the best-effort process to the SRT processes so that they could make deadlines and perform reasonably well, while preserving some fairness between the two SRT processes. With strict priorities assigned in the RM scheduler, only the highest priority (shortest period) task benefits at the expense of lower priority tasks.

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 peri-

ods [9]. By using EDF to make the actual scheduling decisions, BEST does not exhibit this property. Figure 3.5 shows the performance of the three schedulers with one best-effort and three SRT processes: one with a frame rate of 13 frames/second requiring 30% of the CPU, one with a frame rate of 19 frames/second requiring 30% of the CPU and one with a frame rate of 62 frames/second and requiring 31% of the CPU. In this case, the SRT processes need a total of 91% of the CPU and have non-harmonic periods. The Linux scheduler (not shown) provides approximately equal CPU to each process, with the result that the best-effort process makes too much progress and prevents the SRT processes from meeting enough deadlines—they miss 40.7%, 37.1%, and 20.1% 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 7.7% of its deadlines. Outperforming both other schedulers, BEST executes the processes and meets almost all SRT deadlines, missing less than 1% of each processes' deadlines.

An important question about any SRT scheduler is how it performs in situations of system overload. Best-effort schedulers will generally 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) [9], and EDF will miss all deadlines by roughly the same amount [33]. 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 3.6 shows the performance of the Linux, BEST, and RM schedulers with three processes, one best-effort, one SRT with frame rate of 25 frames/second requiring 62.5% of the CPU, one SRT with frame rate of 33 frames/second also requiring 62.5% 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 $\frac{1}{3}$ of the CPU, caus-

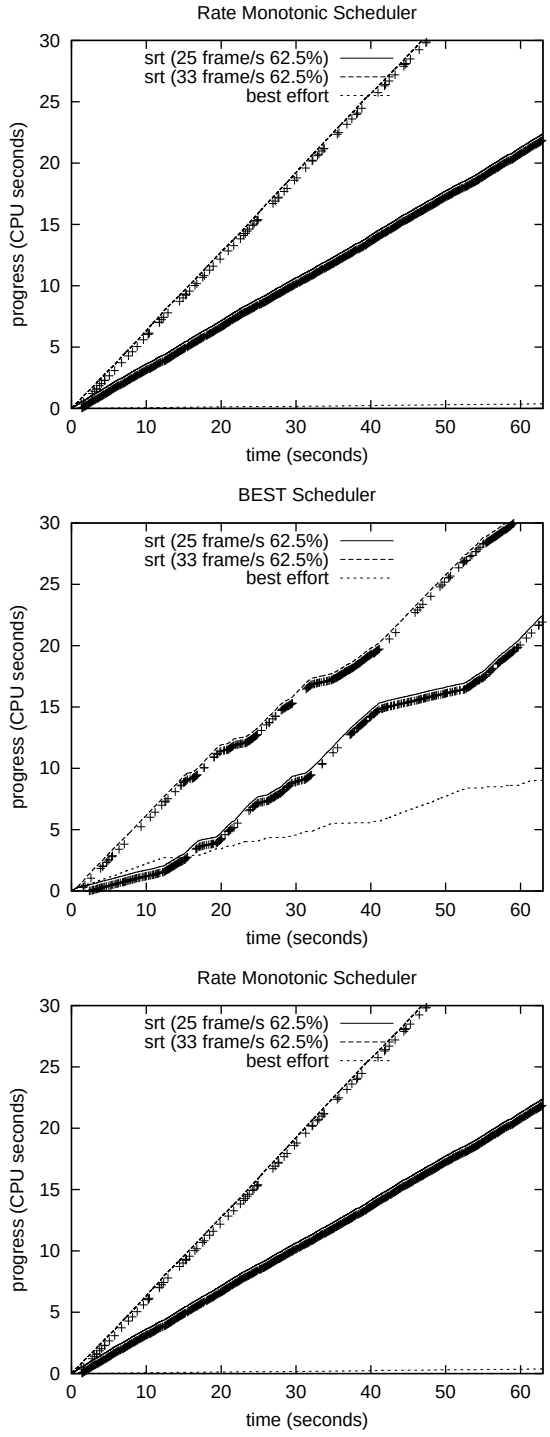


Figure 3.6: Linux, BEST, and RM schedulers with (1) best-effort, (2) srtsim 25fps 62.5%, and (3) srtsim 33fps 62.5%.

ing the SRT processes to miss 52.4% and 44.3% of their deadlines, respectively, and allows the best-effort process to make very good progress, getting a full $\frac{1}{3}$ of the available CPU cycles. The RM scheduler meets all but 5.6% of the deadlines of the SRT processes with shorter deadlines (all those that were possible to meet), but misses 83.4% of the deadlines for the SRT process with the longest deadline only assigns the best-effort process less than 1% of the CPU. The BEST scheduler embodies the best characteristics of both schedulers, distributing (somewhat) and minimizing the missed deadlines (51.3% and 16.8% respectively) while still allowing the best-effort process to make reasonable progress by allocating it 15% of the CPU.

Table 3.1: Summary of percentage of deadlines missed for all experiments.

		Scheduler		
Experiment	**Process**	**Linux**	**BEST**	**RM**
1	best-effort	-	-	-
	srtsim (25fps 50%)	7.8	0.1	0
2	best-effort	-	-	-
	srtsim (25fps 31%)	1.7	0	0.5
	srtsim (33fps 28%)	0.1	0	0
3	best-effort	-	-	-
	srtsim (25fps 62.5%)	26.7	4.8	4.8
4	best-effort	-	-	-
	srtsim (25fps 41%)	26.8	4.6	16.7
	srtsim (33fps 40%)	24.4	3.3	0
5	best-effort	-	-	-
	srtsim (13fps 30%)	40.7	0.1	7.7
	srtsim (19fps 30%)	37.1	0.1	0
	srtsim (62fps 31%)	20.1	0	0
6	best-effort	-	-	-
	srtsim (25fps 62.5%)	52.4	51.3	83.4
	srtsim (33fps 62.5%)	44.0	16.7	5.6

A primary goal of the BEST scheduler is to minimize the number of missed deadlines for soft real-time processes while providing good best-effort performance. The previous figures show that BEST approximates the performance of both Linux and RM as appropriate. Table 3.1 sum-

marizes 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 several 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. We have no reason to believe that BEST is not a satisfactory scheduler for desktop environments.

3.5 Limitations of BEST

The BEST scheduler meets its design goals: users benefit from improved real-time performance for processes that enter the runnable queue in predictable periodic patterns. However there are disadvantages to using the BEST scheduler. An obvious disadvantage includes violation of fairness, which in some cases leads to instability or inability to provide adequate performance for important processes. Also, because BEST knows no *a priori* resource demands, if it cannot detect when processes with deadlines are not performing well.

The assumption that all resources be divided equally among processes is implicit in time-sharing schedulers. Because BEST was not designed with fairness in mind (SRT processes are deemed more important), it provides processes exhibiting periodic behavior an advantage by allowing them to execute for the entire length of their period before resetting their deadline. Because the priority of the process is a function of deadline, an SRT process may retain a high priority

for the length of its period (and even longer if preempted). The result is that BEST allows SRT processes to greedily consume the CPU. We found that once a process establishes its period, if it requires the entire period to meet its deadline, it relinquishes less than 4% of the CPU for each process that competes with it. In fact, once an SRT process establishes its period, it may hinder newer SRT processes from ever meeting deadlines, with the result that they perform without any scheduling advantage because periodic behavior cannot be detected.

Figure 3.6 shows the effect of two SRT processes each greedily attempting to use more than their share of the CPU. Because together they demand more than the full CPU, only one may proceed at their desired rates at any time, and so the two processes trade the rate of progress back and forth between themselves. This is due in part to the statistically driven workload, where many frames are shorter than the average frame; when the process computes short frames, it is able to establish its period with the BEST scheduler, and benefits when longer frames arrive. However too many longer frames creates missed deadlines, occasionally preventing BEST from estimating the period and so the process loses its advantage. The cumulative effect of this phenomenon on both processes creates the wavy progress paths; both receive unstable resource allocation.

Because BEST does not attempt to allocate the CPU equally among processes, it may ignore user-assigned priority. UNIX systems allow users to adjust the relative scheduling priority of a process by changing a parameter called *nice*. In the Linux implementation, *nice* scales the time quantum assigned to processes. In BEST, non-periodic processes receive a deadline expiration timer equal to their Linux time quantum, and therefore an executing process that is not preempted relinquishes the CPU after the same amount of time in both schedulers. However, periodic processes in BEST do not take into account the *nice* value of the process—the scheduler assumes all periodic applications equally important. Without taking into account user-assigned priority, there

is no way for the scheduler to know which of several periodic processes is more important. The processes receiving scheduling priority are the ones lucky enough to meet their previous deadlines (often processes that began executing first, before the overall load was heavy). Even worse, if a best-effort process is more important than a periodic process, there is no way for the user to enforce it.

Chapter 4 describes a variation on the BEST scheduler called BEST-RATE. The goal of the BEST-RATE scheduler is to address the shortcomings described above, by allocating the CPU equally to processes in accordance with priorities assigned by *nice*. By doing so, a process will only meet deadlines if it can already do so within its allocated fair-share of resources, like in best-effort schedulers such as Linux. However, unlike Linux it ensures that the process receives CPU allocations at a rate so that deadlines are met, whereas best-effort schedulers make no effort to allocate a process' share in a timely manner.

The next chapter presents one solution to the limitations described above, but there is another limitation which our research has not yet addressed. BEST works on the principle that processes with periodic deadlines will synchronize their processing of periods using timer primitives provided by the operating system. The rate of entry to the runnable queue coincides with the processes periodic deadline, and for every deadline the process completes on time it may sleep until end of the period, triggering BEST's period detection algorithm upon waking. For processes that meet periodic deadlines, BEST detects their period and helps them meet future deadlines, essentially creating a positive feedback loop. As observed in the multimedia application described in Chapter 2, when a process does not meet its deadline, it will not sleep before beginning its next period. BEST's period detection algorithm is not triggered, and the process will be treated as a best-effort process. Since a best-effort process may not be scheduled in a timely manner, it will

likely miss future deadlines if the CPU is significantly utilized. In effect, this creates an implicit negative feedback loop, but the scheduler does not receive any feedback that an SRT process is progressing poorly, only the user observes the poor performance. It is possible for BEST to remember which processes were periodic, so that a few missed deadlines will not impact future performance (by using averaged measured periods and confidence values and smoothing these averages using hysteresis), but if the process never establishes its period because the system is already loaded, it has no recourse but to perform poorly. Section 5.1 discusses future directions for addressing this limitation.

Chapter 4

The BEST-RATE Scheduler

Like BEST, the goal of BEST-RATE is to enhance the performance of soft real-time tasks. Unlike BEST, BEST-RATE does not adjust the share of resources a process receives. It is a refinement of the BEST technique designed to address several of the issues discussed in Section 3.5 by adding allocation fairness and the ability to handle user-assigned priorities. Like BEST, the underlying scheduler of BEST-RATE uses the EDF algorithm. However, deadlines are assigned according to process rates, as described below in Section 4.0.2.

4.0.1 Design Goals

To develop the BEST-RATE scheduler we had a number of specific design criteria, most of which are identical to the BEST design goals, with the exception that the BEST-RATE scheduler allocates CPU bandwidth equally among processes. The goals 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 need to provide any *a priori* information about processes.

3. The scheduler should enhance the performance of soft real-time applications.

    - The rate of execution of all processes will be monitored. Processes that enter the runnable queue in predictable patterns should receive future CPU allocations so that they consume resources at the same rate as past allocations.
    - 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 receive timely allocation, this prevents missed deadlines.

4. The default behavior of the scheduler should be reasonable and consistent with general purpose time-sharing schedulers.

    - All processes should receive their fair-share of resources. Default behavior allocates equal share to all, but this may be tuned by the user.
    - The scheduler should favor interactive processes over CPU-bound processes.

4.0.2 BEST-RATE Scheduler Details

The BEST-RATE scheduler algorithm is simpler than the BEST 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 actual deadlines for periodic processes and pseudo-deadlines for best-effort processes.

4.0.3 Setting deadlines

Like BEST, BEST-RATE periodically sets a deadline and a *deadline expiration timer* for all processes. The expiration timer is the same as in the BEST scheduler, decrementing only for clock ticks during which the process is executing. There are two times a deadline is reset: (1) when a process is executing and its deadline expires (in `schedule()`), or (2) when a process wakes up (in `wake_up_process()`). In the former situation, the running process has the earliest deadline, and because its deadline expired and is reassigned, usually another process is eligible to run. When a process wakes up, its deadline will be set according to its past behavior, so that if the process consumed CPU in short bursts, as processes with periodic deadlines do, it will be assigned a short period in which to consume another short burst. For each process, the scheduler records a value called usage U. The usage is the weighted average number of clock ticks the process consumes before its deadline is reset, $U = (consumed + w \times U)/(1+w)$, where w is a weighting factor that controls how fast previous measurements are forgotten. For processes with a period shorter than time quantum, U will be below the average period, assuming that most deadlines are met. For CPU-bound processes, it will be exactly the quantum, and for I/O-bound processes, it will generally be lower than the quantum.

The deadline is computed using a process's target rate, which is the amount of CPU it should be allocated assuming a fair share. The total load L of the system is the sum of all nice priorities, $L = \sum_{x \in n} pri_x$, where n is the set of running processes and pri_x is the nice priority of process x. The target rate R of a process x is $\frac{pri_x}{L}$. The deadline is then computed $deadline = now +$

U/R. For all processes, the deadline expiration timer is set to the same value as the Linux quantum. When the load consists of all best-effort processes, BEST-RATE chooses the same schedule as the Linux scheduler. However BEST-RATE schedules I/O-bound and periodic processes with less latency then Linux because of the deadline ordering of processes.

Other changes

Like BEST, the BEST-RATE scheduler uses higher timer resolution than Linux (increased by a factor of 8), for the same reasons described in Section 3.3.3.

4.1 Experimental Results

To show that the BEST-RATE scheduler meets the design criteria set forth in Section 4.0.1 we conducted a few experiments comparing the performance of the BEST-RATE scheduler with that of the Linux scheduler and BEST Scheduler. We implemented the BEST-RATE algorithm in *schedsim*, and measured the throughput of all processes and the number of missed deadlines for periodic processes, using the same methods outlined in Section 3.4.

Our results show that BEST-RATE alleviates the problem seen in Linux when an SRT process requires a CPU allocation close to its fair share. Figure 4.1 shows the performance of the Linux scheduler and the BEST-RATE scheduler with one best-effort process and one SRT process that requires 50% CPU (the same experiment of Figure 3.1). In the Linux scheduler, the SRT process misses 7.8% of its deadlines, while BEST-RATE misses only a single ($< 0.1\%$) deadline.

In Figure 4.2(a), the load of best-effort processes was increased by running three best-effort processes with a single SRT process (with a frame rate of 25 frames/second and requiring 25% of the CPU). In this case, the Linux scheduler provides a quarter of CPU to each process, but

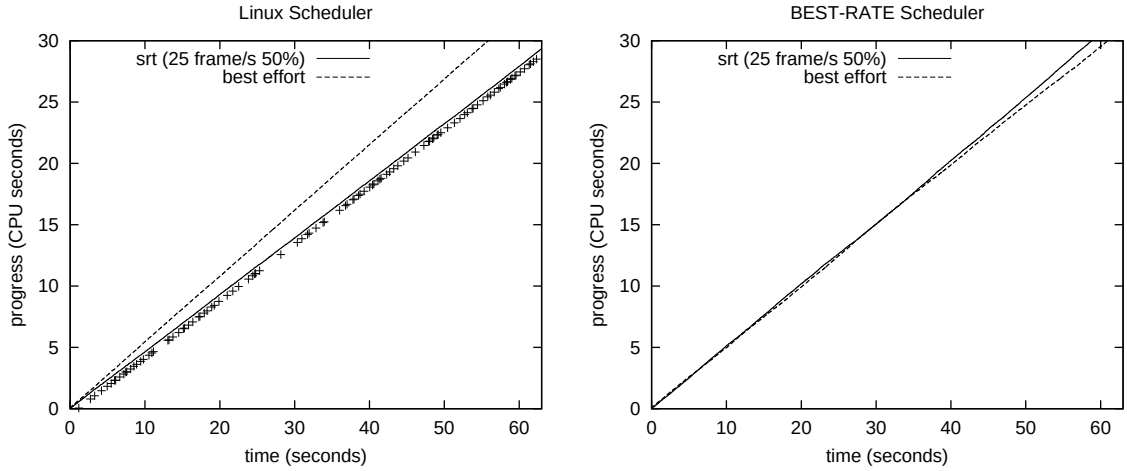
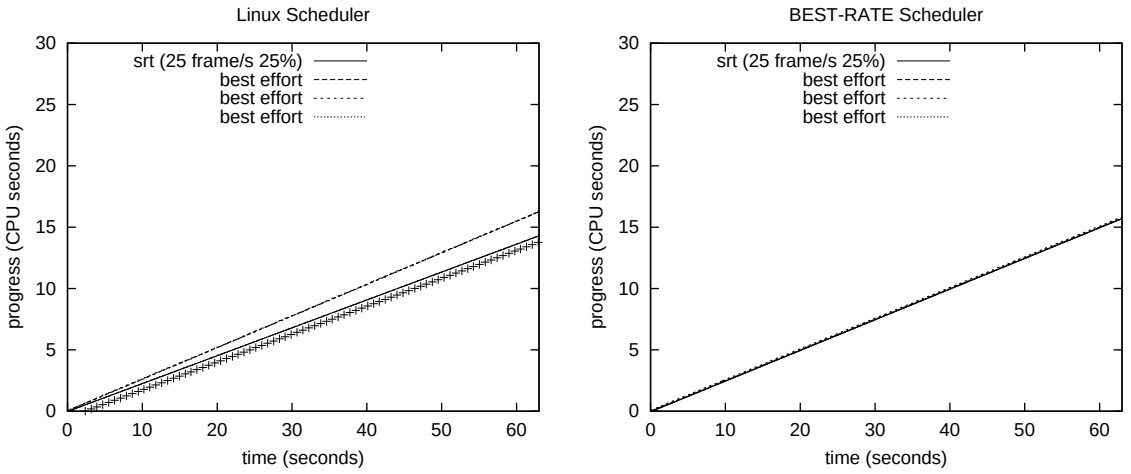


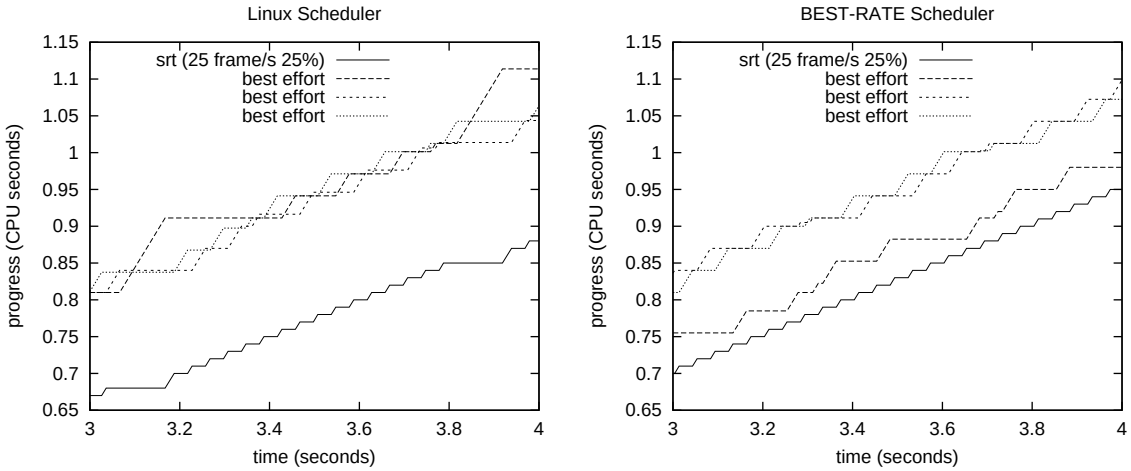
Figure 4.1: Linux and BEST-RATE running (1) best-effort and (2) srtsim 25fps 50%.

like in the previous experiment, the SRT process is unable to meet all of its deadlines, missing 5.6% of them. In the BEST-RATE scheduler, the SRT process meets all of its deadlines. Figure 4.2(b) shows the plot on a shorter time-scale, so a detailed view of CPU allocation is visible. In the Linux scheduler, we see several instances where the progress of the SRT progress is halted while a best-effort processes receives the allocation, and so misses its deadline. This is due to phasing of dynamic priorities assigned by the Linux scheduler. Because the SRT process periodically sleeps, its dynamic priority decays at a slower rate than CPU-bound processes, and usually upon waking it preempts the currently executing process. However, the dynamic priorities of all running processes are occasionally recomputed, and it is possible that the SRT process will not always be greater upon waking, in which case the executing best-effort process may complete an entire quantum without interruption. The BEST-RATE scheduler eliminates this problem, resulting in evenly spaced CPU allocations.

UNIX users may adjust the relative priorities of processes using the *nice* utility, which sets a priority value in the range -20 to +19. The default value is 0, and in the Linux implementation



(a) Linux and BEST-RATE schedulers.



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

Figure 4.2: Linux and BEST-RATE schedulers running (1-3) 3 best-effort processes and (4) srtsim 25fps 25%.

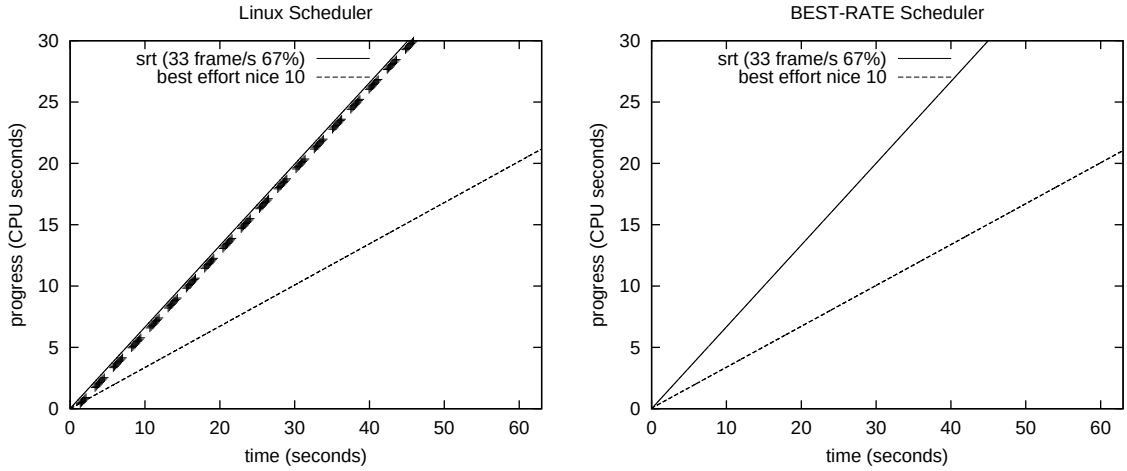


Figure 4.3: Linux and BEST-RATE running (1) best-effort (w/ nice 10) and (2) srtsim 33fps 67%.

nice scales a process's time quantum (on a linear scale, where -20 will double the time-slice, and +10 will halve the time-slice). For example, when running a program using *nice*, the default Linux behavior assigns the process a nice priority of 10, halving its quantum. If this process is competing with another process, it receives $\frac{1}{2}$ quantum for every quantum of the other process, effectively reducing its usage to $\frac{1}{3}$ of the CPU.

Because the BEST-RATE scheduler does not attempt to allocate over a process's fair share of resource, an SRT process needing more than its nominal share to meet deadlines may not perform well. The goal of the BEST scheduler is to attempt to meet any deadlines it can detect, however the goal of BEST-RATE is only to meet those which can be met within the process's fair share. In Figure 4.3 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 may allocate by assigning the best-effort process a default *nice* of +10. Like in the last two examples, even though the Linux scheduler provides the SRT process the share it needs to meet deadlines, it does not receive them in a timely manner and misses 11% of them. However in the BEST-RATE scheduler, the SRT process does not miss any

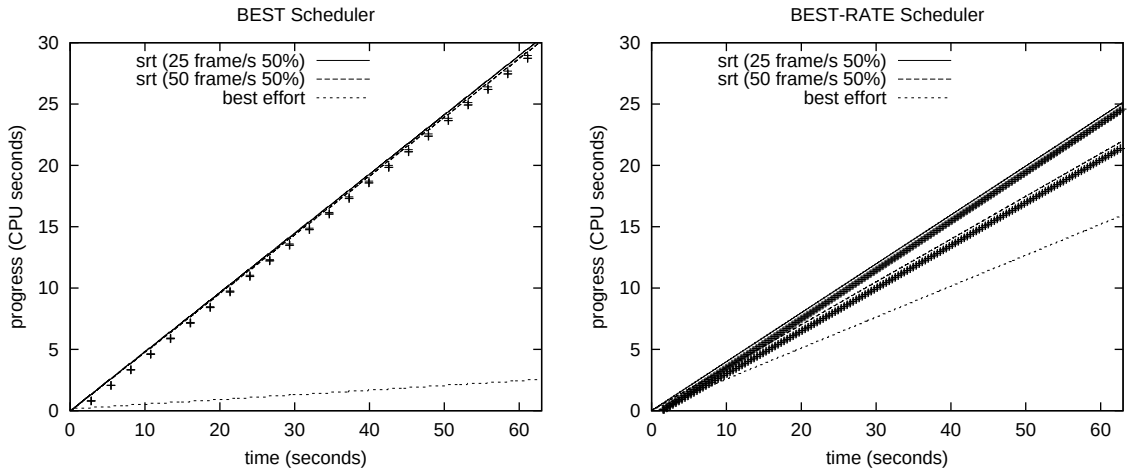


Figure 4.4: BEST and BEST-RATE schedulers with (1) best-effort, (2) srtsim 25fps 50% and (3) srtsim 50fps 50%.

deadlines.

In the last experiment, the performance of BEST-RATE when deadlines cannot be met is evaluated. Unlike BEST which attempts to violate fairness in order to meet deadlines, BEST-RATE allocates resources according to assigned priority. Figure 4.4 shows the performance of the BEST and BEST-RATE schedulers with three processes, one best-effort, one SRT with a frame rate of 25 frames/second and CPU demand of 50%, and another SRT with frame rate of 50 frames/second and demand of 50%. The BEST scheduler performs well, missing only 1.6% of each processes deadlines. As expected, BEST-RATE is not capable of meeting deadlines, but outperforms Linux (not shown) by missing only 25% and 14.2% of respective deadlines (Linux misses 32% and 21%). An interesting result is that the SRT processes missed the exact same number of deadlines under BEST-RATE (the percentages are different because the number of periods in the run differed), although further experiments show that BEST-RATE does not equally balance the number of missed deadlines during overload for all workloads.

The goal of the BEST-RATE scheduler is to minimize the number of missed deadlines

Table 4.1: Summary of percentage of deadlines missed for all experiments.

		Scheduler		
Experiment	**Process**	**Linux**	**BEST**	**BEST-RATE**
7	best-effort	-	-	-
	srtsim (25fps 50%)	7.8	0	0
8	best-effort	-	-	-
	best-effort	-	-	-
	best-effort	-	-	-
	srtsim (25fps 25%)	5.6	0	0
9	best-effort (nice 10)	-	-	-
	srtsim (33fps 67%)	11	0	0
10	best-effort	-	-	-
	srtsim (25fps 50%)	32.0	1.6	25.0
	srtsim (50fps 50%)	21.0	1.6	14.3

for soft real-time processes that are able to meet deadlines in their nominal share of resource allocation. Like the Linux scheduler, it attempts to provide fairness, and like BEST it ensures that the process gets its CPU allocation in a timely manner. The previous experiments show that BEST-RATE meets this goal. Table 4.1 summarizes the percentage of deadlines missed by the schedulers. It shows that BEST-RATE exceeds the performance of Linux in situations where deadlines can be met. However for processes in which demand exceeds share, BEST-RATE behaves like Linux and provides a fair allocation to all processes.

Chapter 5

Future Work and Conclusion

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, because BEST violates fair-sharing assumptions of time-share scheduling used in general purpose systems, the BEST-RATE scheduler was developed to provide timely allocation to real-time processes within fair-share resource allocation.

The goal of this research project is to explore an approach to soft real-time scheduling that uses the best-effort model of scheduling, in which the system knows nothing *a priori* about the workload or resource needs of processes. Most existing soft real-time schedulers were developed by relaxing the constraints of real-time schedulers. Because they are rooted in real-time techniques, most SRT schedulers use interfaces similar to real-time systems and require some level of workload characterization for proper performance. After beginning to examine soft real-time scheduling

from the best-effort point-of-view, we have identified several areas for future research in this area, discussed below.

5.1 Future Work

When comparing the performance of BEST-RATE to BEST, we found a case during overload where one scheduler causes two processes of different period to miss the same *number* of deadlines, and the other scheduler causes the processes to miss the same *percentage* of deadlines. Although we also find that this behavior is not consistent under other loads, it leads to the question: what is the "proper" way for a best-effort scheduler to degrade performance when deadlines cannot be met? The SRMS scheduler was evaluated using a measure of *intertask unfairness*, which assigns a value to the unfairness among tasks that miss deadlines [1], and in future experiments we should incorporate this metric into our evaluations.

Ultimately, "fairness" should be based on a user's concept of the relative importance of processes. Locke demonstrated that an appropriate method to deal with overload is to assign utility values for each deadline, and choose to schedule for the deadlines that provide high aggregate utility [25]. This solution allows real-time systems to make best-effort scheduling decisions, and influenced much of the later soft real-time research. To achieve a similar policy in BEST, we may define utility functions for deadlines derived from user-assigned priorities. Some real-time scheduling research provides advice for choosing utility values [10, 37], and it is known that using value-based scheduling leads to more graceful degradation in the absence of guarantees than EDF [11]. Using value in BEST, we hope to provide more consistent and predictable behavior when handling overload situations, while still preserving the best-effort model of no *a priori* re-

source demands.

BEST infers periodic deadlines of processes by measuring statistics during execution. It is reasonable to assume that if an application exhibits periodic deadlines during one execution, then it will have deadlines in future executions. If the scheduler saves some state for an application, it may be able to quickly infer its behavior patterns without needing to wait for run-time characterization. This is especially helpful when a process begins executing during heavy processor utilization—the process may not receive enough CPU to meet deadlines and therefore will not be correctly identified by BEST. If the scheduler knows that a process has deadlines, it may be able to schedule it effectively from the beginning. To implement this feature in BEST, we need to decide how to save this state, and which information to save. For example, some application behavior may depend on the media format or frame rate, which varies for each media file. In this case, it may be appropriate to associate SRT state with the media. But if two different applications execute the same media using different execution patterns, it may be necessary to associate the state with the applications, and not the media. However it may be adequate to simply flag that a process is SRT, and leave the period characterization to run-time.

One unresolved problem discussed Section 3.5 is the inability of BEST to receive negative feedback about real-time process performance. Because in the best-effort model, the application has no interface to the scheduler, there is no way for an application to notify the scheduler when it does not receive timely resources to make deadlines. Lack of interface to the scheduler is a limitation of best-effort models, and it may be worthwhile to investigate relaxing this run-time model. If there exists a system call allowing a process to relay to the scheduler that it missed a deadline, the scheduler is equipped to receive negative as well as positive feedback. Although such a system call violates the best-effort model, it does not require the scheduler to know any resource

needs of applications, and may still use the BEST model of on-line resource determination. By adding a negative feedback component to BEST we may explore more techniques for providing system support to real-time applications without need for workload characterization.

With the increasing popularity of more sophisticated mobile and hand-held devices such as digital cellular phones, a current fertile topic in systems research is reducing power consumption. These devices often run soft real-time applications that process voice, video, and other media content. These applications need to be scalable for adapting to available processor and communication bandwidth. Recent research finds that dynamically adjusting energy usage to adapt to workloads leads to considerable power savings [2]. An interesting research path is to investigate how we might exploit the run-time characterization techniques used by BEST to control the resource allocations of these scalable SRT applications so that they effectively use power.

5.2 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 and BEST-RATE are CPU schedulers that adhere 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.

BEST-RATE allocates CPU to processes so they progress at the rate of their weighted share, but ensures that processes that need the CPU within time constraints receive the resources with less latency.

This paper presents the design and implementation of the prototype BEST and BEST-RATE schedulers in the Linux kernel. It includes the results of a set of experiments demonstrating the effectiveness of both schedulers 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 under-load and processor overload and is done with no *a priori* knowledge of the applications, their resource usage, or their periods. We also show that BEST-RATE is a weighted-share scheduler that is effective at allocating CPU with less latency to processes that exhibit periodic behavior.

Bibliography

[1] Alia K. Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, Madrid, Spain, December 1998. IEEE.

[2] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Majía-Alvaraz. Dynamic and agressive scheduling techniques for power-aware real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 95–105. IEEE, December 2001.

[3] Scott Banachowski and Scott 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.

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

[5] Michael Beck, Harold Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner. *Linux Kernel Internals*. Addison Wesley Longman, 2nd edition, 1998.

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

[7] Scott Brandt, Gary Nutt, Toby Berk, and James Mankovichr. A dynamic quality of service middleware agent for mediating application resource usage. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, pages 307–317, Madrid, Spain, December 1998.

[8] Scott A. Brandt. Performance analysis of dynamic soft real-time systems. In *Proceedings of the 20th IEEE International Performance, Computing and Communications Conference (IPCCC '01)*, pages 379–386, Phoenix, AZ, April 2001.

[9] Alan Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, 6:116–128, May 1991.

[10] Alan Burns, Divya Prasad, Andrea Bondavalli, Felicita Di Giandomenico, Krithi Ramamritham, John Stankovic, and Lorenzo Strigini. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46(4):305–325, January 2000.

[11] Giorgio Buttazzo, Marco Spuri, and Fabrizio Sensini. Value vs. deadline scheduling in overload conditions. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS 1995)*, 1995.

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

[13] Hao-hua Chu and Klara Nahrstedt. A soft real time scheduling server in UNIX operating system. In *European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, September 1997.

[14] Kenneth J. Duda and David 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*, December 1999.

[15] Bryan Ford and Sia Susarla. CPU inheritance scheduling. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, October 1996.

[16] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[17] Christopher J. Hughes, Praful Kaul, Sarita V. Adve, Rohit Jain, Chanik Park, and Jayanth Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[18] The Institute of Electrical and Electronics Engineers. *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 edition, 1994.

[19] Kevin Jeffay and David 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*, April 1995.

[20] E. Douglas Jensen, C. Douglass Locke, and Hideyuki Tokuda. A time-driven scheduling

model for real-time operating systems. In *Proceedings of the 6th IEEE Real-Time Systems Symposium*, December 1985.

[21] Sandeep Khanna, Michael Sebrée, and John Zolnowsky. Realtime scheduling in SunOS 5.0. In *USENIX Winter 1992 Technical Conference*, pages 375–390, January 1992.

[22] Ian M. 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. In *IEEE Journal on Selected Areas in Communications*, pages 1280–1297, September 1996.

[23] Chih-han Lin, Hao-hua Chu, and Klara Nahrstedt. A soft real-time scheduling server on the Windows NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*, August 1998.

[24] C. L. Liu and James W. Layland. Scheduling algorithms for multiprograming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[25] C. Douglas Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie-Mellon University, May 1986.

[26] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing, 1996.

[27] Clifford W. Mercer, Stefan Savage, and Hideyuki 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.

[28] Bob Mullen. The Multics scheduler. http://www.multicians.org/mult-sched.html, August 1995.

[29] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard 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.

[30] Jason Nieh and Monica Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the Sixteenth Symposium on Operating System Principals*, October 1997.

[31] John Regehr. Inferring scheduling behavior with Hourglass. In *Proceedings of the Freenix Track: 2001 USENIX Annual Technical Conference*, pages 143–156, Monterey, CA, USA, June 2002. USENIX.

[32] John Regehr, Michael B. Jones, and John A. Stankovic. Operating system support for multimedia: The programming model matters. Technical Report MSR-TR-2000-98, Microsoft Research, September 2000.

[33] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio Buttazo. Implications of classical scheduling results for real-time systems. *Computer*, 28(6):16–25, June 1995.

[34] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Buruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the Real-Time Systems Symposium*, pages 288–299, December 1996.

[35] Hideyuki Tokuda, Tatsuo Nakajimi, and Prithvi Rao. Real-time Mach: Towards a predictable real-time system. In *Proceedings of USENIX Mach Workshop*, October 1990.

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

[37] Lonnie Welch and Scott Brandt. Toward a realization of the value of benefit in real-time systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, San Francisco, CA, April 2001.

[38] David K.Y. Yau and Simon S. Lam. Adaptive rate-controlled scheduling for multimedia applications. In *ACM Multimedia Conference*, November 1996.

[39] Victor Yodaiken and Michael Barabanov. Real-time Linux. In *Proceedings of Linux Applications Development and Deployment Conference (USELINUX)*, January 1997.