The BEST Desktop Soft Real-Time Scheduler

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

October 14, 2001

1 Introduction

It is important that desktop CPU schedulers support soft-real time processing now that multimedia applications have become ubiquitous on desktop computers. They must also gracefully handle mixes of applications with different timing requirements. The best-effort scheduling provided by desktop systems is easy to use, provides a reasonable trade-off between fairness and responsiveness, and imposes no extra overhead for specifying application resource needs. However, these schedulers provide no resource or timeliness guarantees, limiting their ability to support applications with deadlines [10]. To better support soft real-time applications, while recognizing that the best-effort model permeates desktop computing for very good reasons, we are developing the **B**est-effort scheduler **E**nhanced for **S**oft real-time **T**ime-sharing (BEST). BEST transparently provides significantly improved support for periodic soft real-time (SRT) processes while retaining the well-behaved default characteristics of best-effort schedulers.

By observing the times at which processes enter the ready queue, BEST dynamically estimates a period for each process exhibiting periodic behavior, assigns appropriate pseudo-periods for non-periodic processes, and schedules all processes using earliest deadline first (EDF) [7]. This boosts the performance of periodic processes while preserving the behavior of traditional best-effort schedulers for non-periodic processes. The result is a multi-class best-effort scheduler that handles CPU-intensive, I/O-intensive, and periodic SRT processes using a single scheduling algorithm. Our preliminary results demonstrate that without any *a priori* knowledge of the SRT applications, BEST outperforms the Linux scheduler in handling SRT processes, outperforms real-time schedulers in handling best-effort processes, and sometimes outperforms both.

Section 2 discusses the design and implementation of BEST, Section 3 presents our preliminary results, and Section 4 presents some concluding remarks.

2 Design and Implementation

Soft real-time schedulers exist [1, 3, 4, 5, 6, 9, 11, 12], but they impose constraints on developers and users that limit their practicality in generic desktop environments; they require applications to interface with special-purpose routines and, like most real-time systems, they generally require specifications of application resource usage and period—numbers that can be difficult or impossible to obtain for generic desktop applications on widely varying platforms. Consequently, despite the lack of support for such applications, best-effort scheduling remains the preferred model for most multimedia application developers and users.

Accordingly, in developing the BEST scheduler we have four main design goals:

1. One scheduler should handle all three process types—CPU-bound, I/O-bound, and SRT.
2. Neither users nor developers should have to provide any *a priori* information about the processes to be executed.
3. The default behavior of the scheduler should be reasonable:
   - I/O-bound processes should receive a priority boost over CPU-bound processes as in current best-effort schedulers, SRT processes should receive a priority boost over both I/O-bound and CPU-bound processes, and no process should starve.

- The priority boost given to SRT processes should be based on classical real-time scheduling results, *i.e.* priority boosts should be based on rate or deadline requirements.
- Increasing the load of an under-loaded system should not degrade the performance of existing processes, and performance should degrade gracefully as the system becomes overloaded.

4. The scheduler should be suitable for use as the default scheduler in general-purpose desktop systems:
   - It should be efficient.
   - Pure best-effort performance should be comparable to that of best-effort schedulers, pure real-time performance should be comparable to that of real-time schedulers, and the performance with both types of processes should be better than that of either best-effort or real-time schedulers.
   - It should allow users to increase or decrease the performance of individual applications as desired.

We have implemented a prototype BEST scheduler in Linux. Like other UNIX schedulers [2, 8], it dynamically calculates process priorities, but BEST uses an even simpler algorithm than the Linux scheduler: every process is given a deadline (dynamically recomputed as appropriate), and the ready process with the earliest deadline is executed next.

In our preliminary implementation of BEST, each application's period is calculated each time it enters the ready queue. The *current period* is the time elapsed since the process last entered the ready queue and the *effective period* is a weighted average of the current period and the previous effective period. A process's next deadline is simply the current time plus its effective period. A running process also has a deadline timer that limits the amount of CPU time it will receive before its current deadline. Every time a process is scheduled, it executes until either its deadline timer expires, it blocks, or it is preempted. If a process's deadline timer expires before it blocks, its next deadline is delayed to slightly beyond the maximum period—in effect lowering the priority of any process that runs past its deadline. This ensures that processes with detected periods or short CPU bursts will have earlier deadlines while CPU-bound processes will have later deadlines. However, because a postponed deadline is not determined until after the process is allocated the CPU, starvation is prevented.

BEST is based in part on the assumption that SRT periods can be determined by observing the times at which processes enter the ready queue. We instrumented the Linux kernel to record the entry time of some single-threaded multimedia processes and found that these processes did exhibit measurable periodic behavior (Table 1). Furthermore, because of data-dependent performances differences, no *a priori* specification is likely to be able to fully capture this information, even for a single platform—dynamic detection of periods is likely to be the only technique that will work in general.

Table 1: Average Period, Standard Deviation, and CPU Usage for Sample Multimedia Processes

Process	**Average period (ms)**	**Standard deviation**	**CPU usage**
mpeg_play (24 frame/s)	40.1	0.2	16%
mpeg_play (30 frame/s)	33.0	0.2	17%
mpg123 (128 kbit/s)	161.7	1.9	2%
mpg123 (128 kbit/s) (different mp3)	160.2	1.1	2%

3 Results

We conducted a set of experiments that compare the performance of BEST with the Linux scheduler (Linux) and a static priority-driven Rate Monotonic (RM) scheduler. As with all real-time schedulers, RM requires *a priori* knowledge of application periods, while Linux and BEST do not. While running combinations of CPU-intensive and periodic SRT processes, we measured the progress of all processes and the number of missed deadlines incurred by periodic processes. Two synthetic applications were used in the experiments: *loop*, a best-effort process that endlessly consumes CPU, and *periodic*, an SRT process with a periodic deadline.

Figure 1 shows the performance of Linux and BEST running one best-effort process (loop) and one SRT process (periodic, with $\frac{1}{10}$ second period and 40% CPU usage). Because Linux provides approximately equal amounts of CPU to each application, and the SRT process requires less than 50% (it's nominal fair share), Linux meets all application deadlines. Similarly, BEST meets all deadlines and provides the same amount of resources to each application as Linux.

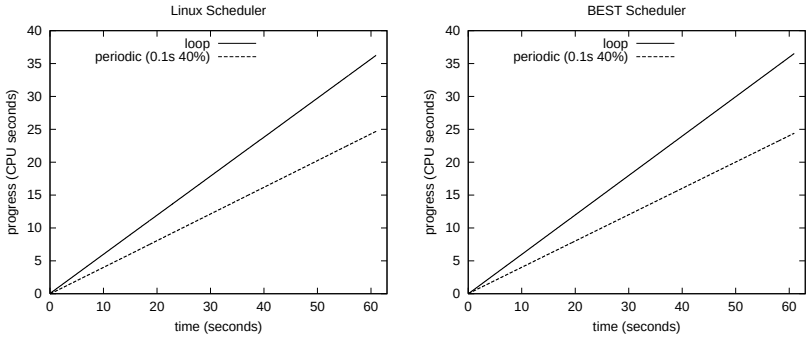

Figure 1: Linux and BEST running (1) loop and (2) periodic($\frac{1}{10}$ second period, 40% CPU usage)

Figure 2 shows the performance of Linux, BEST, and RM with one best-effort process (loop) and one SRT process (periodic, with $\frac{1}{10}$ second period and 70% CPU usage). Here we see that Linux provides approximately 50% of the available CPU cycles to each process, causing the SRT process to miss 29% of its deadlines. By contrast, RM provides the SRT process with 70% of the available cycles, enabling it to meet all of its deadlines while still allowing the best-effort process to progress at a reasonable rate. In this case, BEST provides exactly the same performance as RM. Recall, however, that BEST dynamically determines the application periods whereas RM requires that the periods be specified in advance in order to determine appropriate static priorities.

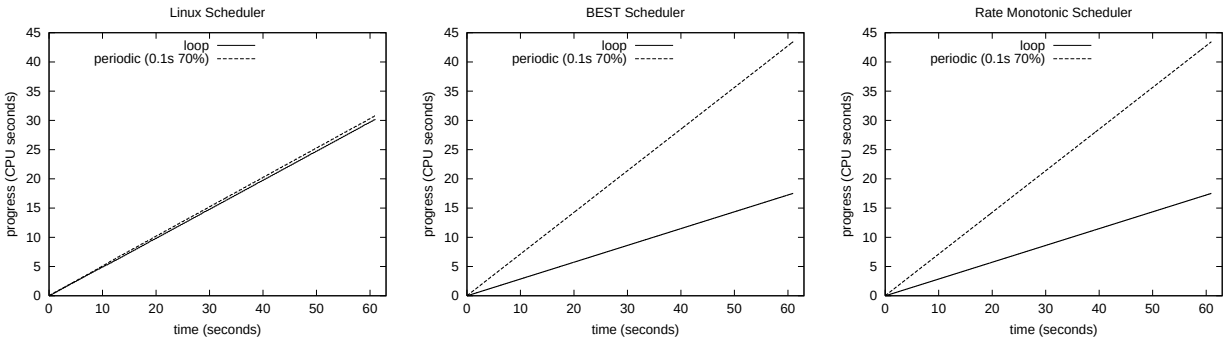

Figure 2: Linux, BEST, and RM with (1) loop and (2) periodic ($\frac{1}{10}$ second period, 70% CPU usage)

Because Linux is unaware of resource requirements or deadlines, its well-intentioned scheduling decisions can result in some processes missing deadlines that could have otherwise been met. For example, with one best-effort and two SRT processes, each of which require 30% of the CPU (one with a period of $\frac{1}{10}$ of a second and the other with a period of 1 second) Linux will cause the SRT processes to miss some deadlines even though a feasible schedule exists. Similarly, RM is generally unable to find a feasible schedule when the SRT applications collectively use between about 70% and 100% of the CPU and have non-harmonic periods [7]. Because it makes its low-level scheduling decisions using EDF, BEST correctly handles these situations (not shown here) and meets all application deadlines.

An important question about any SRT scheduler is how it performs when the SRT applications collectively require more than 100% of the CPU. Figure 3 shows the performance of Linux, BEST, and RM with one best-effort process (loop) and three SRT processes (periodic with 1 second period, periodic with $\frac{1}{2}$ second period, and periodic with $\frac{1}{10}$ second period, each requiring 40% of the CPU); no scheduler can meet all of the deadlines in this overburdened situation. Linux gives each process roughly $\frac{1}{4}$ of the CPU, allowing the best-effort process to make very good progress

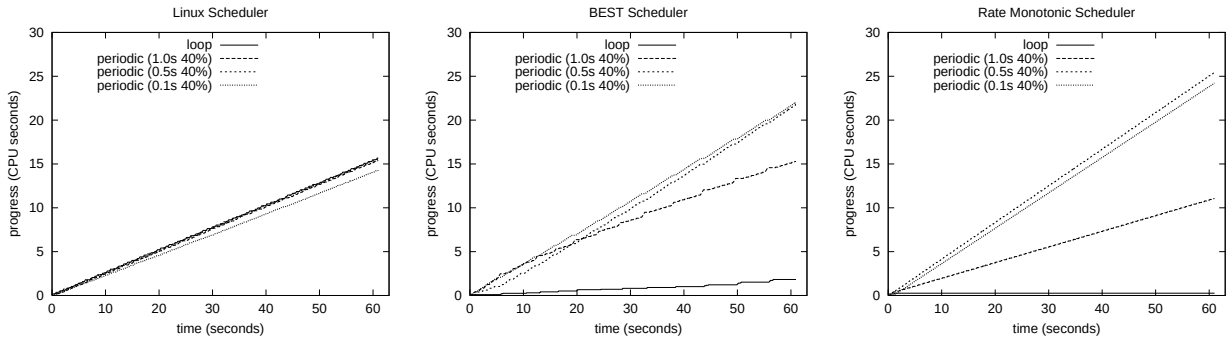

Figure 3: Linux, BEST, and RM with (1) loop (2) periodic (1 second period, 40% CPU usage) (3) periodic ($\frac{1}{2}$ second period, 40% CPU usage) and (4) periodic ($\frac{1}{10}$ second period, 40% CPU usage)

but causing the SRT processes to miss 98%, 93%, and 19% of their deadlines, respectively. RM meets all of the deadlines for the two SRT processes with shorter periods, but misses 98% of the deadlines for the SRT process with the longest deadline and starves the best-effort process completely. BEST embodies the best characteristics of both schedulers, distributing and minimizing the missed deadlines somewhat (71%, 5%, and 2% respectively) while still allowing the best-effort process to make reasonable, if not stellar, progress.

4 Conclusion

Our preliminary experiments show that BEST meets its basic purpose: enhancing the performance of periodic processes while capturing the benefits of a best-effort model. BEST performs as well as or better than the Linux and RM schedulers in handling best-effort, soft real-time, and a combination of the two types of processes, without any *a priori* knowledge of the applications. However, we have yet to achieve all of the goals we have set forward for the scheduler, and have not yet fully examined the performance of our preliminary implementation. Our ongoing work aims to further develop the BEST scheduler, analyze its performance with a variety of realistic workloads, and develop a final implementation tuned for optimal desktop performance. To create more realistic workloads, we will use real single and multi-threaded SRT applications and stress the scheduler with fluctuating workloads. We will also examine the best-effort and real-time behavior in more detail — we believe that BEST will exhibit comparable responsiveness to the default Linux scheduler and will have significantly better jitter performance, but we have yet to measure either of these important characteristics.

We also plan to examine the impact of changing the priorities of best-effort and SRT processes, both manually via "nice" and automatically inside the scheduler. It is our belief that via priority we can adjust both the relative progress of the best-effort processes and the missed deadline characteristics of the SRT processes. More generally, we believe that by automatically adjusting the priorities of the various processes we can adapt the system to provide any missed-deadline behavior and any ratio of best-effort to SRT performance desired. In so doing, we should be able to spread out the missed deadlines across the SRT processes, minimize the missed deadlines of more important processes, or provide good progress to more important best-effort processes while still meeting important SRT deadlines, as desired.

References

[1] 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.

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

[3] Scott Brandt and Gary Nutt. Flexible soft real-time processing in middleware. *Real-Time Systems*, 2001. to appear.

[4] 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.

[5] 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.

[6] M. Jones, J. Barbera III, and A. Forin. An overview of the Rialto real-time architecture. In *Proceedings of the 7th ACM SIGOPS European Workshop*, pages 249–256, September 1996.

[7] 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.

[8] 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.

[9] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE Internation Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

[10] 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 Operationg System Support for Digital Audio and Video*, 1993.

[11] 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.

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