Missed Deadline Notification in Best-effort Schedulers

Scott Banachowski, Joel Wu, and Scott A. Brandt
C.S. Dept., University of California Santa Cruz, 1156 High St., Santa Cruz, CA USA 95064

ABSTRACT

It is common to run multimedia and other periodic, soft real-time applications on general-purpose computer systems. These systems use best-effort scheduling algorithms that cannot guarantee applications will receive responsive scheduling to meet deadline or timing requirements. We present a simple mechanism called Missed Deadline Notification (MDN) that allows applications to notify the system when they do not receive their desired level of responsiveness. Consisting of a single system call with no arguments, this simple interface allows the operating system to provide better support for soft real-time applications without any *a priori* information about their timing or resource needs. We implemented MDN in three different schedulers: Linux, BEST, and BeRate. We describe these implementations and their performance when running real-time applications and discuss policies to prevent applications from abusing MDN to gain extra resources.

Keywords: Soft Real-time and Multimedia Systems

1. INTRODUCTION

Recent advances in hardware enable general-purpose computers to handle more demanding software applications and, as a result, soft real-time processing is becoming pervasive. It is common for desktop systems to execute workloads containing tasks with deadline or rate-based time constraints such as streaming video and audio, games, and software modems. As devices such as cell phones become more sophisticated, embedded operating systems will begin to provide the same type of services as desktop systems and be expected to execute similar workloads. It is therefore becoming increasingly important for general-purpose systems to provide some type of real-time scheduling capability.

Many conventional operating systems use CPU schedulers adapted from *best-effort* time-share systems. As the term "best-effort" implies, these schedulers provide no facilities for specifying or meeting performance guarantees. As a result, processes with temporal constraints may not receive timely allocation of CPU required to meet their deadlines. Best-effort scheduling is attractive due to its simplicity and ease of use; applications do not require system interfaces to specify timing or resource requirements, nor do they need to know the resource requirements, which may depend on CPU speed and other unpredictable factors. This makes best-effort systems unsuitable for hard real-time applications, where a missed deadline is considered a system failure. Nevertheless, for systems that execute soft real-time tasks mixed with conventional desktop workloads, best-effort scheduling is desirable.

One common approach for meeting soft deadlines is to over-provision resources (by either by keeping the load low or upgrading the hardware), thus eliminating resource contention and greatly increasing the likelihood that applications will meet their deadlines. This is neither efficient nor always feasible for shared systems, because the system is under-utilized. Increasing the CPU bandwidth available to real-time processes may increase the likelihood of meeting deadlines, nevertheless existing time-share schedulers are not equipped to ensure that resources are assigned with the timeliness required by tasks with deadlines. The alternative of using a special-purpose real-time scheduler adds complications: it requires complex new interfaces that violate best-effort system principles, and system partitioning to isolate and schedule the mixed workloads.

Best-effort systems typically provide the ability to set task priorities by adjusting their relative importance (e.g. in many UNIX-like systems this priority is called *nice*). This priority indicates that a task should receive more or less resources than others, but tells the system nothing about the timeliness requirements of the applications. In order to increase the information available to the system, we introduce a mechanism called **Missed Deadline Notification** (MDN). MDN is a system call that applications invoke whenever they miss a deadline. This small hint from the application allows the scheduler to provide significantly improved service to soft real-time applications. Figure 1 illustrates the information required and the type of guarantees provided by various types of systems.

Further author information:
Email: {sbanacho,jwu,scott}@cs.ucsc.edu

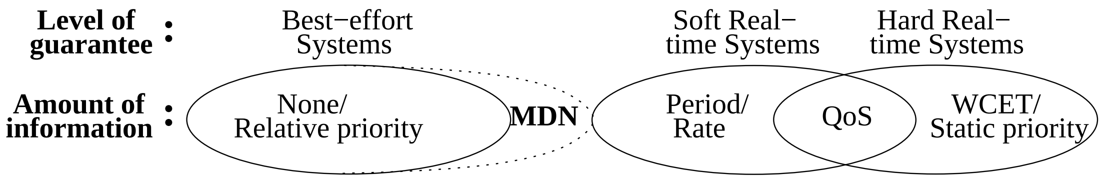


Figure 1. MDN is a relatively unexplored concept that brings best-effort systems closer to soft real-time.

Missed Deadline Notification allows tasks to indicate timeliness needs and, in keeping with a best-effort philosophy, without attaching any actual, pre-determined value to the requirement. Traditionally, a real-time scheduler must be told a task's timing constraint, for example the length of its periodic deadline, or a target processing rate of 25 packets/second, etc. Using MDN, the task tells the system when it missed a deadline or is not processing at its desired rate. MDN is a run-time system call because a task cannot *a priori* determine if it will miss deadlines. The overhead of the system call is kept low because it simply updates the state of a single process. The system may respond appropriately to the notification, in best-effort manner; if will improve the task's responsiveness if it can, but if not the request may be ignored.

The Missed Deadline Notification technique is not tied to any specific scheduling algorithm; it may augment any system. To this end, we incorporated MDN into three different best-effort schedulers: Linux [**?**], BEST [1], and BeRate [2]. The rest of this paper discusses MDN, and shows how it was implemented in each system. Although the technique is general, each system requires a different implementation to best take advantage of MDN. Each scheduler provides improved service in a slightly different way, and each requires a different mechanism to ensure that rogue applications cannot abuse MDN to greedily gain more resources.

2. MISSED DEADLINE NOTIFICATION

Missed Deadline Notification is implemented as a system call with no arguments, and no return value (the C function prototype is `void mdn(void)`). The intended use of MDN is for applications to notify the operating system scheduler of missed deadlines or insufficient rate. The system treats the call as advisory—it may take action in response, ignore it, or whatever its policies dictate. Section 3 discusses policies for the three best-effort schedulers we examined.

MDN employs a minimal interface. The idea is that the kernel uses the single bits of information to infer more about the applications needs, and responds in a best-effort manner. In order to understand the engineering decision for this interface, we briefly describe the origin of MDN in the context of past research.

2.1. Origin of MDN

A large number of soft real-time and multimedia applications process data in periodic intervals. For example video players decode frames at specific frame-rates, and audio players fill sound device buffers at the same rate data is emptied from the buffer by the hardware. When a soft real-time task consistently meets deadlines, its period may be inferred from its behavior [1]. To synchronize its processing, a task periodically blocks before the release of each job, and subsequently enters the run queue at regular periodic intervals. The BEST scheduler [1] infers periods of such tasks by measuring when they enter the run queue, and schedules using earliest deadline first (EDF) so that they meet predicted periodic deadlines.

Task behavior when meeting deadlines provides positive reenforcement for the BEST algorithm, because synchronization (by sleeping until the next release) is detected by the algorithm. This no longer holds when a task misses a deadline; instead of blocking to wait for its next job release, it begins its next job immediately, interfering with the ability of BEST to detect and track its period. Well-performing tasks provide positive reinforcement, which subsequently helps them make future deadlines, but BEST does not detect periods of tasks that miss deadlines, further hurting their chances for future deadlines. The intent of Missed Deadline Notification is to mitigate this effect by providing means for applications to provide reinforcement to the BEST algorithm even when they miss deadlines.

This mechanism is not limited to the BEST algorithm. MDN provides an advisory message to the system scheduler, and can be incorporated into any scheduling algorithm. The MDN interface was kept minimal because, in the spirit of the best-effort policy in which processing constraints are not specified by applications, it provides negative feedback about performance with the least possible amount of information from the applications.

2.2. Policies

It is up to the implementor of the scheduler to determine how to use the information offered by MDN, as the response of the system should reflect the goals of the scheduling algorithm. For example, if the goal of the scheduler is fair allocation of resources, it may try to improve responsiveness for complaining applications, but without compromising the application's fair share. Thus, in this example, MDN affects only the timing, but not the amount, of resources. On the other hand, if the goal of the scheduler is high quality of service (QoS) for soft real-time applications, the scheduler may try to adjust the allocation of resources to minimize the number of MDNs or so that application incur the same number or rate of missed deadlines.

An appropriate action in response to an MDN may be to favor the task that complains over others. A simple implementation may boost a task's priority whenever it calls MDN. However, in such a system, a misbehaving application may abuse the MDN mechanism to increase its priority and steal CPU cycles from others. Care must therefore be taken to use policies that minimize the extent of "cheating" that a task may get away with.

The following section discusses MDN in the context of three best-effort schedulers, each designed with slightly different goals. In each, we chose policies best suited to the design of the scheduler, with special attention to policies that resist the effect of rogue or misbehaving applications.

3. MDN IMPLEMENTATION

We implemented MDN in three schedulers: Linux, BEST, and BeRate. The Linux scheduler is a time-share scheduler designed for general-purpose use. The BEST scheduler is a best-effort scheduler enhanced for soft real-time that works by detecting periods of tasks and scheduling them using a real-time scheduling algorithm. The BeRate scheduler is a rate-based scheduler that reduces scheduling latency for periodic and interactive tasks by inferring rate parameters from applications as they run.

The implementations remain best-effort and cannot always instantaneously alleviate scheduling problems. When a task misses a deadline while it is not scheduled, it cannot call MDN until a later time, when the task is next allocated the CPU. And in our implementations, to keep overhead low, the action of an MDN updates the state of the calling task but this change may not impact its performance until a later iteration of the scheduler. Thus the effects may not always become apparent immediately. Despite these limitations, we find that MDN improves the performance of best-effort schedulers.

3.1. Linux

Scheduling algorithms for time-share systems are designed to provide a fair-allocation of CPU resources over the long term. Additionally, they give a short-term advantage to I/O-bound processes to improve their responsiveness. This is often achieved using a multi-level feedback queue [3], which lowers the priority of tasks when they don't voluntarily release the CPU by blocking.

Linux[*] implements an approximation of the multi-level feedback queue by maintaining for each task a *quantum counter*; this counter changes dynamically as tasks execute and the state of the system changes. Each task also has a user-settable relative scheduling priority (*nice*). Whenever the scheduler runs, it dispatches the task in the run queue with the highest non-zero counter. The counter decrements as the task executes; when the counter reaches 0 the task is not eligible for selection. When all runnable tasks consume their quanta (i.e. no runnable task has a non-zero counter), the scheduler recomputes the counters of *all* processes using the formula $counter = counter/2 + nice$. At this time, blocked tasks with remaining quanta ($counter > 0$) receive a new priority greater than their *nice* priority, making them more likely to be selected by the scheduler when awakening and thus boosting their responsiveness.

Although goals of the Linux scheduler include fairness and low response latency, it does not always meet application timeliness requirements even when the resources are available for it to do so. A periodic task should meet deadlines if its CPU consumption is less than its share of CPU bandwidth. In practice, the actual performance depends on factors such as the phasing of period with the system clock (when the kernel does accounting), and the value of counters relative to other processes. Once a task's counter runs out, it must wait until all other runnable tasks are serviced, possibly incurring large delays that cause deadlines to be missed.

[*]Versions 2.2 through 2.4.

3.1.1. Linux MDN enhancements

When the system receives a Missed Deadline Notification under Linux, the scheduler responds by temporarily increasing the process's priority. Each process is assigned, in addition to its nice priority, a value called *mdn_nice* that is initialized to 0. The function that refreshes the counter is changed to $counter = counter/2 + (nice + mdn\_nice)$. When receiving an MDN, the kernel determines if the task is eligible for a priority increase and, if so, it increments its *mdn_nice* so that the next time its counter is computed it will receive a higher priority. A task is only eligible for this priority boost if it meets the anti-cheat criteria, explained in the following section.

3.1.2. Linux anti-cheat policy

Because MDN increases the priority of a task, we do not wish to allow tasks to abuse it to gain more CPU. Otherwise applications may repeatedly "cry wolf." MDNs are ignored unless the calling task is cooperative regarding its recent use of the CPU. If a task never voluntarily blocks, then it already receives as much CPU as Linux allows under the workload. A task's recent use of CPU is considered conservative if it occasionally releases the CPU by blocking. Whenever a task wakes from blocking the kernel tracks a running average of its *block ratio*, the average ratio of time spent out of the run queue. When a task calls MDN, the priority boost it receives is proportional to its block ratio—if the task never blocks it receives no boost but if its demand for CPU is relatively low it receives a greater priority so that it is more likely to receive CPU when it needs it.

In response to MDNs the task's priority may rise, but the algorithm also limits priority inflation by automatically decrementing *mdn_nice* back to 0 as a function of time. The rate at which it decrements is inversely proportional to the block ratio; if a task becomes compute bound (i.e. the block ratio goes to zero), its priority quickly drops to normal. The only way a process may use MDN to maintain a high priority is to consume CPU conservatively; this dichotomy discourages cheating. The algorithm may be tuned so that it more or less restrictive in granting MDN requests.

It is possible that a periodic task may never block because the system is just too overloaded; in this situation MDN will have no effect and the system cannot attempt to help the application. A disadvantage of best-effort systems not addressed by our Linux MDN implementation is that in heavily overloaded situations, the system is simply limited in the resources it can provide.

3.2. BEST

The goal of the BEST scheduler is to improve the performance of soft real-time tasks by detecting periodic processes and improving their chances of meeting deadlines. By inferring the periods of running processes, BEST uses their periodic properties to make scheduling decisions without any *a priori* knowledge of process requirements. It is aimed at desktop users who desire better performance from multimedia applications without the complexity of a system with service guarantees.

The scheduler decides which programs have periodic deadline requirements by making an assumption: tasks with periodic deadlines enter a *runnable state* when they begin a periodic computation, and upon completion of a job, use synchronization primitives (such as timers) to wait for the beginning of their next period. By observing the times that tasks enter the run queue, the scheduler can make reasonable guesses about which are periodic, and may determine their period. A "well-behaved" periodic process enters the run queue in a predictable pattern. The scheduler estimates period as the time elapsed between subsequent entries to the run queue, and maintains an average period for each task; if the scheduler detects consistent period for a task then it assumes that it has a periodic deadline.

The BEST scheduler uses an earliest deadline first algorithm. Because the scheduler assigns every task a period, the EDF algorithm uses the end of each period as the deadline. For tasks that block, a deadline is set whenever the job awakens. The BEST algorithm must also assign each task a maximum run-time, so that the scheduler will update the deadline for tasks that do not block before their period expires.

Not every process that repeatedly blocks is periodic, so BEST evaluates the confidence that an estimated period is indeed due to periodic behavior. Confidence is a measure of the difference between the current measured period and the average period. A process is "well-behaved" if its confidence exceeds a threshold. A process that is not well-behaved is assigned a pseudo-period so that in the schedule, it round-robins with other non-periodic tasks. Interactive tasks are assigned shorter periods than compute-bound tasks so that they remain more responsive; best-effort workloads in BEST behave similarly to the Linux scheduler.

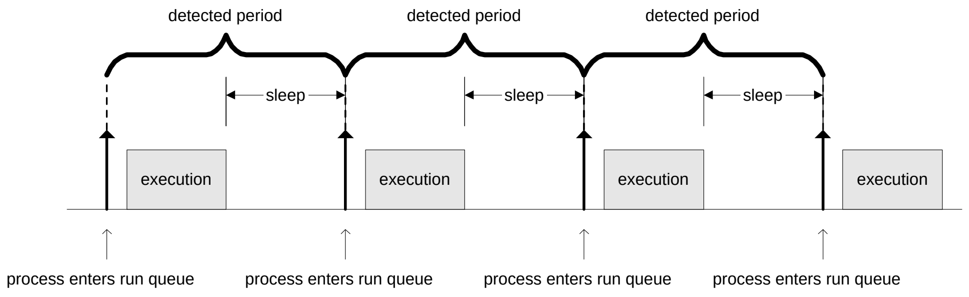


Figure 2. The detection of period by BEST in the normal case.

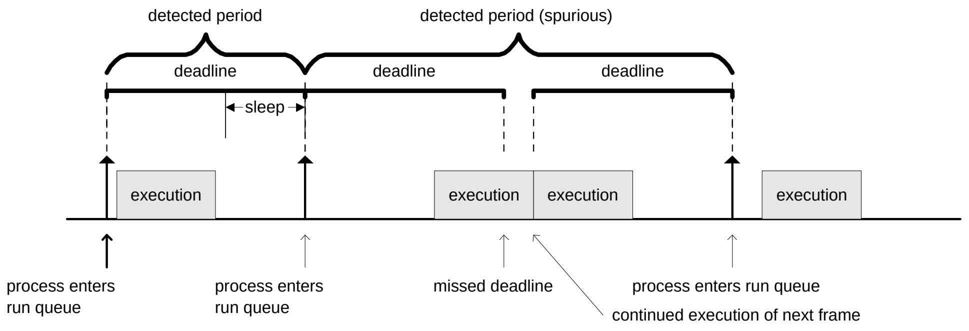


Figure 3. Missed deadlines introduce errors into BEST period detection.

3.2.1. BEST MDN enhancements

Figure 2 shows the execution of a periodic task. BEST detects its period as the duration between subsequent wakes. When missing a deadline, the task does not sleep but instead begins the following job, as in Figure 3. BEST will not detect a synchronization period until later, when a job completes on time. If the spurious value of a late, or missed deadline, is used to calculate the average period, it will mislead BEST to use an improper deadline for EDF scheduling. This is likely to cause more missed deadlines, perpetuating the cycle of miscalculated periods.

After a missed deadline, BEST should not use two successive synchronizations as the observed periods. To some

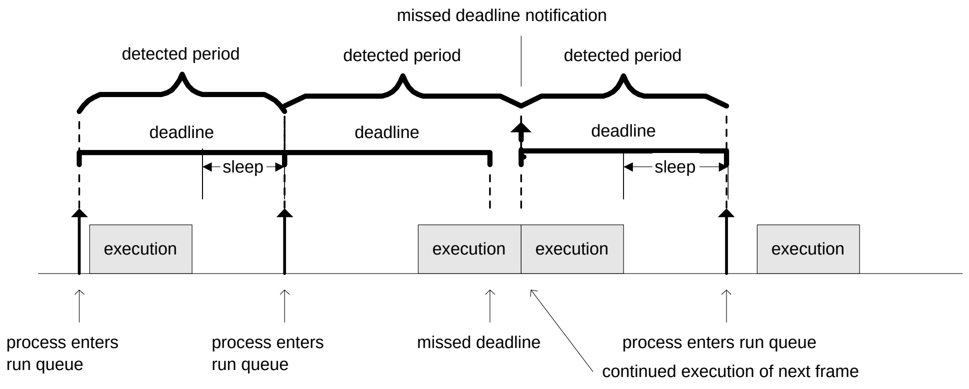


Figure 4. More accurate detection of period with Missed Deadline Notification.

extent, the confidence estimate described above alleviates this. But there is is no way for BEST to differentiate between a spurious incorrect period due to missed deadlines, or a long execution time for a single job. MDN indicates that the execution of a job completed late. BEST may use the MDN to substitute for a synchronization when a deadline is missed, as shown in Figure 4.

3.2.2. BEST anti-cheat policy

Unlike Linux, BEST does not give priority boosts to tasks that call MDN. BEST uses MDNs as indicators that the period it is attempting to detect has passed. Therefore, BEST discards any MDNs that do not arrive within a reasonable window of the expected period; when an MDN arrives within a window of the expected deadline, BEST uses the MDN as an indicator of the end of a period. If a task has not already established an average estimated period then MDN is ignored; this prevents non-periodic tasks from using MDN. It is assumed that if the task has not yet previously established a period, then the system is too overloaded to support it. If BEST has previously detected a task's period then MDN won't increase its amount of resources (only the likelihood of using estimating proper deadlines in the future), leaving little incentive for tasks to cheat.

3.3. BeRate

The Best-effort Rate CPU scheduler (BeRate) [2] schedules with a rate-based algorithm. Tasks are characterized with two parameters: the task should complete e units of computation over every p interval of time [4]. The rate-based task model is a generalization of the periodic task model [5], which characterizes tasks by worst-case execution time and periodic deadline.

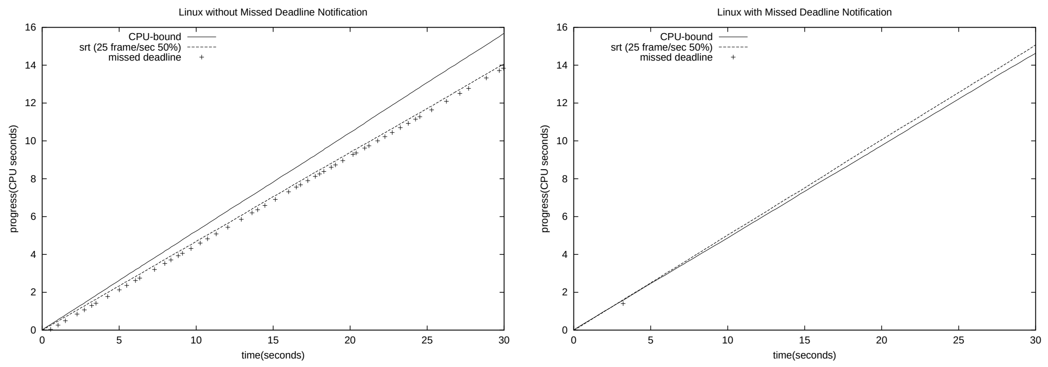
As tasks execute, BeRate measures both the amount of CPU they consume and the intervals of consumption over consecutive periods. The result, as in BEST, is that the scheduler provides better responsiveness to latency-sensitive tasks such as multimedia, because it schedules to their observed behavior. The rate-based algorithm schedules tasks so that they can continue processing at the same rate as in the past, so a periodic task that meets its deadlines receives the same type of positive reinforcement as in BEST. Also BeRate preserves the behavior of traditional time-sharing schedulers for non-periodic processes. Because BeRate has fine control over task utilizations, it may limit the utilization of tasks to their fair-share of resources, preventing interference with other tasks, and allowing users to tune performance with *nice* priorities. Thus, BeRate provides the responsiveness of BEST with the fairness policy of Linux.

Like BEST, BeRate uses an EDF scheduler, but the way BeRate assigns deadlines to tasks differs. BeRate assigns deadlines based on how the task executed in the past. Each task has a priority q (which is its default Linux quantum) based on the value of its *nice* priority. For each process, BeRate measures the duration of consecutively requested CPU. This duration e is sampled every time the task blocks, and equals the actual CPU consumed by the task since the last time it blocked. If the task does not block for the length of a quantum, then at the end of the quantum the value is sampled (so in this case $e = q$). The running average of the measured runtime e is called $\bar{e}$. BeRate does not know the period of tasks, but using $\bar{e}$, it estimates the period to be $p_{est} = \bar{e}\frac{L}{q}$, where L is the sum of all tasks' quanta ($L = \sum q_i$ over all tasks i). Using this estimate, if a periodic task may meet its deadlines given a fair share of processing (q/L), then BeRate estimates its period so that it will meet its deadlines using EDF. The following example explains how.

Assume that a periodic task has an average runtime per job of a. If the task is to meet deadlines within its fair share, then $a/p_{actual} \leq q/L$, where p_{actual} is the actual period unknown to the scheduler. As the task runs, it will block, on average, after consuming a units of CPU, measured by BeRate so $\bar{e} \approx a$. The estimated period is $p_{est} = \bar{e}\frac{L}{q} \leq a\frac{p_{actual}}{a}$, so $p_{est} \leq p_{actual}$. Because the estimated period is at most the actual period, the assigned deadline assures that the task will be scheduled on time. For a CPU-bound processes, $\bar{e} = q$, so their estimated period is proportional to system load L; this means as the load of the system increases, CPU-bound tasks are less likely to interfere with periodic tasks. When running a workload entirely consisting of CPU-bound processes, BeRate chooses the same round-robin schedule as Linux. For I/O bound processes, $\bar{e}$ depends on the rate and duration of CPU bursts, and will be $\leq q$, generally leading to deadlines earlier than CPU-bound processes, for improved responsiveness.

3.3.1. BeRate MDN enhancements

The BeRate MDN implementation is similar to Linux: in Linux, a task's priority is temporarily increased in response to MDN, but in BeRate the task's share parameter q is increased (by adding an *mdn_nice* analogous to the way the priority in Linux was increased). The effect is that the estimated deadline is earlier, and the task will be serviced quicker. Although by



(a) Without Missed Deadline Notification (b) With Missed Deadline Notification

Figure 5. Application progress under Linux with and without MDN while running (1) CPU-bound and (1) srt 25 fps 50%. Missed deadlines are indicated with crosses below the progress line.

design, this does not increase the actual amount of resources assigned to a task, because there is no control of the periodic release rate, when a job is serviced quicker its subsequent jobs may be released sooner, and it may receive a slight increase in resource consumption.

3.3.2. BeRate anti-cheat policy

BeRate controls the performance boost of MDN in the same manner as the Linux MDN, by letting any increase in share decay as a function of time, and controlling the rate that share rises or falls in proportion to a task's block ratio (see Section 3.1.2 for details). Like in Linux, this creates a dichotomy where tasks can only increase their resources by using less, making cheating futile.

4. RESULTS

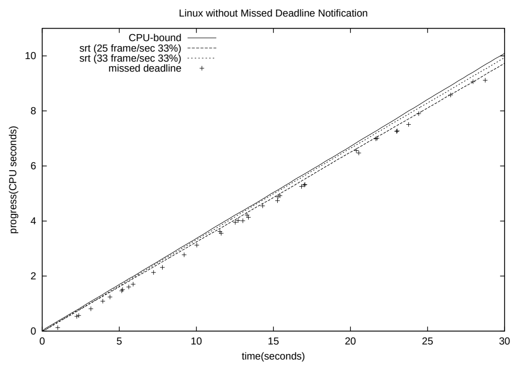
We conducted experiments comparing the performance of the three best-effort schedulers with and without the Missed Deadline Notification service. Two synthetic workload applications were used: *CPU-bound* consumes CPU using math operations, creating load and contention for CPU cycles in competition with the soft real-time (SRT) processes. The soft real-time application *srtgen* computes data using a periodic deadline workload that models frame-to-frame variability similar to the decoding of an MPEG video stream. We measured the performance by observing the number of deadlines missed with and without MDN.

All the schedulers are implemented in the Linux 2.4.19 kernel. In Linux, we added MDN to the scheduler code. Both BEST and BeRate implementations replace some functions of the Linux scheduler with new code (as described in Section 3), but otherwise use the remaining kernel functionality. Data was collected on Intel Pentium P3 and P4 processors.

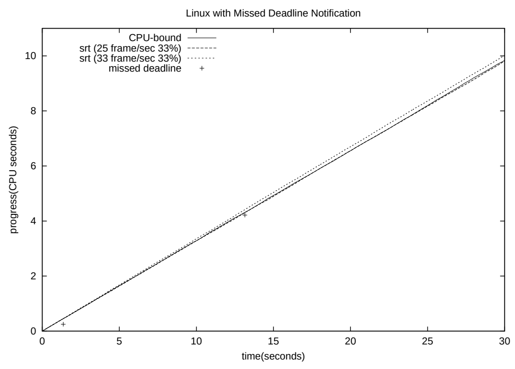
4.1. Linux

Figure 4 plots the progress when a CPU-bound process runs with an SRT process that requires 50% share of the processor bandwidth. Because the Linux scheduler provides approximately equal CPU to each application, the SRT process should meet its deadlines. However, the SRT process misses 8% of its deadlines (Figure 5(a)). Although the process receives enough CPU allocation, it does not always receive it on time. When the soft real-time process calls Missed Deadline Notifications, the scheduler gives the task a slight priority boost, enough so that it misses only a few deadlines (Figure 5(b)).

In Figure 4.1, we doubled the number of SRT tasks, setting each SRT processing requirement to a third of the CPU bandwidth. In Linux each SRT process missed more than 2% of its respective deadlines, while with MDN neither task missed more than 2 deadlines. In another experiment, we ran an SRT process with a desired utilization higher than its nominal share, and using MDN the task is able to effectively gain enough CPU to meet most of its deadlines using MDN (figure not shown, but the results are summarized in following table).

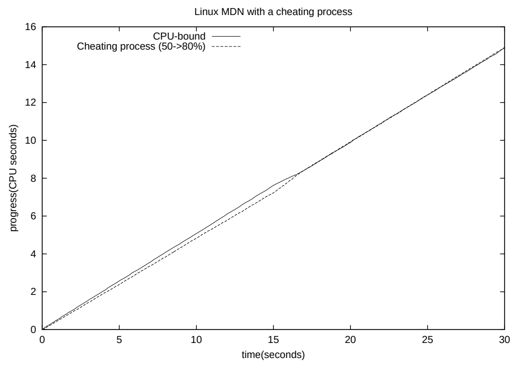


(a) Without Missed Deadline Notification

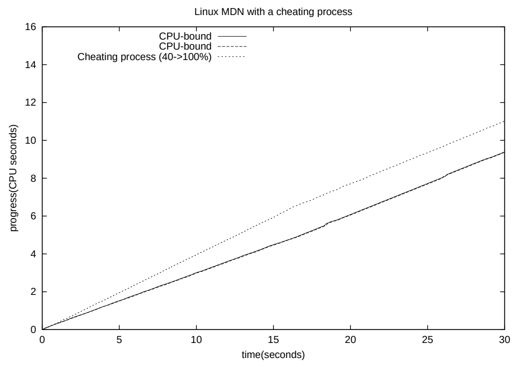


(b) With Missed Deadline Notification

Figure 6. Application progress under Linux with and without MDN while running (1) CPU-bound, (2) srt 25 fps 33%, and (3) srt 33 fps 33%. Missed deadlines are indicated with crosses below the progress line.



(a) A task trying to gain more resources.



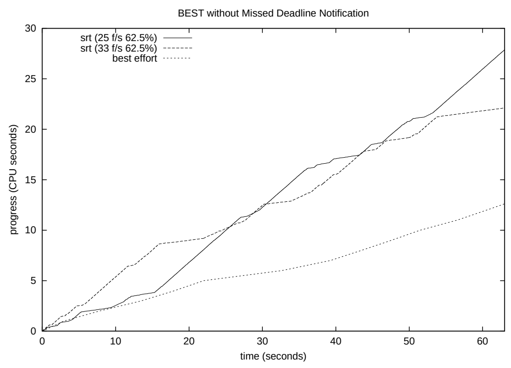
(b) A task becoming compute-bound.

Figure 7. Application progress under Linux with MDN. In both figures, at time 15 one of the processes increases its resource demand and attempts to gain more resources by calling MDN.

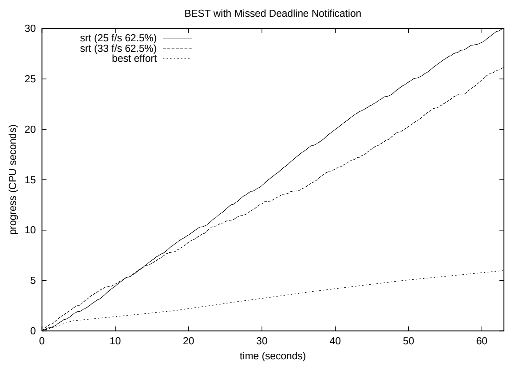
	Missed Deadlines		Percent CPU Consumed	
Processes	Without MDN	With MDN	Without MDN	With MDN
srt (25 f/s 50%)	7.9%	<0.1%	46.9%	50.3%
CPU-bound	–	–	52.3%	48.8%
srt (25 f/s 33%)	3.1%	<0.1%	32.4%	32.7%
srt (33 f/s 33%)	2.3%	<0.1%	33.1%	33.4%
CPU-bound	–	–	33.6%	32.8%
srt (25 f/s 60%)	41.2%	1.5%	50.1%	60.0%
CPU-bound	–	–	49.4%	39.1%

The total CPU utilization does not equal 100% because utilization by system overhead and other tasks such as shells and daemons are not accounted.

Figure 4.1 shows tasks attempting to cheat. In Figure 7(a), after 15 seconds a task, which previously required 50% CPU to meet its periodic deadline, begins calling MDN more frequently as it increases its CPU usage to 80%. But because it rarely blocks, and requires much more than its fair share of CPU to meet deadlines, Linux's anti-cheat mechanism

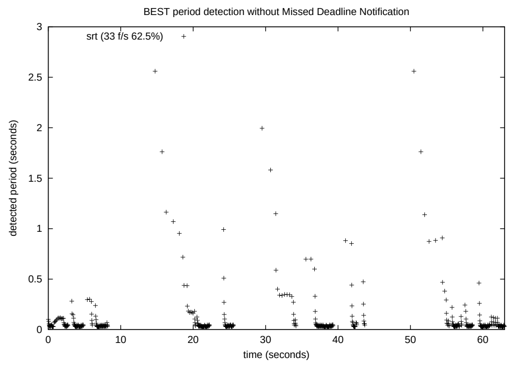


(a) Without Missed Deadline Notification

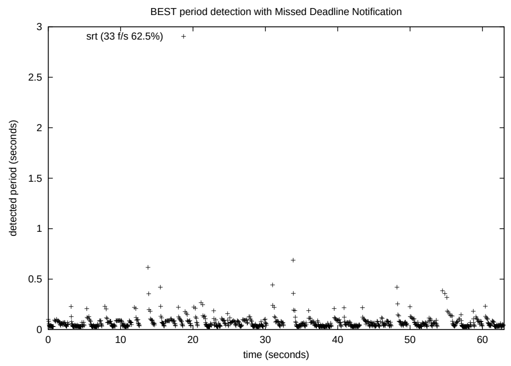


(b) With Missed Deadline Notification

Figure 8. Application progress under BEST with and without MDN while running (1) CPU-bound (2) srt 25 fps 62.5%, and (3) srt 33 fps 62.5%. The system is very overloaded, so without MDN progress is unstable.



(a) Without Missed Deadline Notification



(b) With Missed Deadline Notification

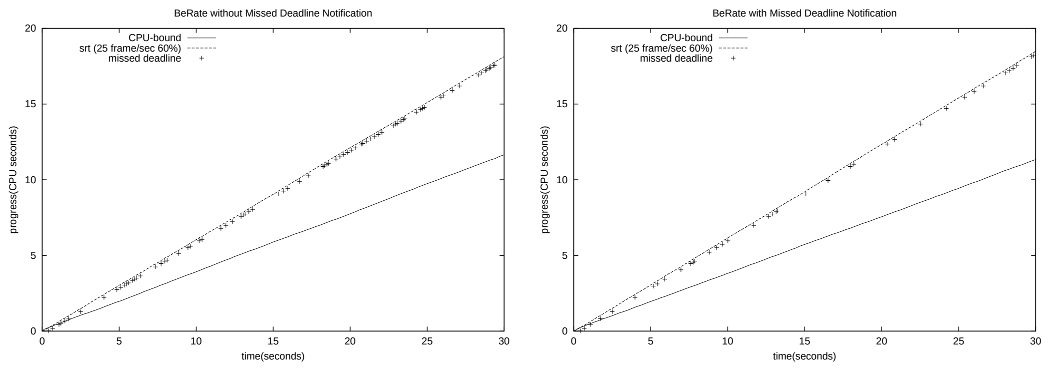
Figure 9. These plots show the period that BEST measures from one of the tasks in Figure 4.2. When the task misses many deadlines because of overload, BEST cannot detect its period, but using MDN helps alleviate this effect.

doesn't allow it to gain more resources. Figure 7(b) shows another case in which a task initially requires 40% CPU to meet deadlines, but its fair share is 33%. Using MDN, it is able to receive a priority boost and proceed at a high enough rate to meet its deadlines. After 15 seconds, the task becomes compute-bound, and repeatedly calls MDN in attempt to maintain its inflated priority. However, because the task's block ratio drops, its progress is quickly set to the same rate (rate equals slope of progress) as the other CPU-bound tasks.

4.2. BEST

The BEST scheduler is designed to provide better performance to any task that executes with a detectable period. If the total demand for CPU of a set periodic tasks does not approach 100%, then under BEST periodic applications make their deadlines, even in the presence of other best-effort workload. However, when the load of periodic tasks becomes too high, BEST cannot meet all deadlines. No scheduler can meet all deadlines during overload, but most real-time schedulers use admission control to prevent this; because BEST is a best-effort scheduler, it does not use an admission control policy.

Figure 8(a) shows two SRT tasks with a CPU-bound task in an extremely overloaded system. Because together the SRT tasks demand more than the entire CPU bandwidth, only one may proceed at its desired rate at any time. As a result, we observe one task proceeding at its desired rate for a while, but occasionally trading performance with the other task.



(a) Without Missed Deadline Notification (b) With Missed Deadline Notification

Figure 10. Application progress under BeRate with and without MDN while running (1) CPU-bound and (1) srt 25 fps 60%. Missed deadlines are indicated with crosses below the progress line.

This is due in part to the variable frame 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 so when longer frames arrive, BEST remembers its period even though it misses some deadlines. However, when faced with a sequence of longer frames that cannot be computed on time, BEST loses ability to track the period, decreasing chances of meeting future deadlines. The cumulative effect of this phenomenon on both tasks creates the unstable progress. Figure 9(a) shows the period of one of the tasks as calculated by BEST. There are long intervals where the task does not meet any deadlines, so never blocks, and BEST loses track of its period.

With MDN, the progress is much more stable (Figure 8(a)). The task with the shorter period is able to make more progress because in the earliest deadline first algorithm, it is more likely to preempt other tasks. Although both tasks still miss about the same number of deadlines (they must when the system is this overloaded), using MDN, BEST is able to better track periods even when the tasks do not make deadlines (Figure 9(b)).

4.3. BeRate

BeRate performs well when tasks require their fair share of resources in order to meet deadlines, so the cases examined in the Linux experiments (where the task demands equal their nominal share) do not miss deadlines in BeRate even without MDN. To observe the effect of MDN in BeRate, we increased the demand so that tasks required slightly more than their fair share. Figure 4.3 shows the performance of a task that needs 60% of the CPU to meet its deadline, but its fair share is half. Without MDN, BeRate almost gives it the required CPU. This is due to an optimization in BeRate that postpones the deadlines of tasks that never sleep whenever they are preempted. With MDN, the SRT task does not receive significantly more resources, but meets many more deadlines.

The effect of MDN in BeRate is that a task's estimated deadline will be slightly earlier, increasing the likelihood it meets future deadlines. As mentioned in Section 3.3.1, this also allows the task to implicitly gain more resources. In these experiments we observe that in BeRate with MDN tasks may receive slightly increased CPU consumption, but not as dramatically as Linux where they explicitly receive a priority boost. In BeRate tasks only receive about 1% resource boost using MDN, but still improve their responsiveness so that they miss less deadlines. The following table summarizes some experiments with BeRate.

	Missed Deadlines		Percent CPU Consumed	
Processes	Without MDN	With MDN	Without MDN	With MDN
srt (25 f/s 60%)	10.9%	5.5%	60.4%	61.7%
CPU-bound	–	–	33.6%	32.8%
srt (25 f/s 33%)	14.9%	9.9%	30.7%	31.7%
srt (33 f/s 33%)	10.5%	9.5%	31.6%	31.8%
CPU-bound	–	–	18.6%	18.0%
CPU-bound	–	–	18.9%	18.2%

5. RELATED WORK

The ideas in Missed Deadline Notification come from two research areas: using best-effort models to support real-time scheduling, and using feedback from applications to make scheduling decisions.

It is well-known that general-purpose operating systems may perform poorly for applications with time constraints, and that using static real-time priorities is inadequate for handling continuous sound or video [6]. Nevertheless, general-purpose systems are widely used to run soft real-time applications such as multimedia. These systems are capable of better real-time support when following some guidelines for configuring the system and applications, such as using page locks and allocating sufficient CPU bandwidth to the system's I/O activities [7]. We wish to avoid using solutions that require offline workload-dependent tuning, so our research focuses on dynamic scheduling algorithms.

Enhancements to general-purpose operating systems help improve their responsiveness. Several projects aim to reduce the latency of context switching; currently low-latency and preemptable kernels are available for Linux. The low-latency patch reduces the size of uninterpretable execution paths inside the kernel by adding opportunities for preemption [8], and the preemptable kernel allows multiple threads in the kernel, so that preemption need not be disabled during system calls [9]. Both approaches reduce major sources of latency in the kernel, so they are important for supporting real-time applications. There is also an effort to reduce the overhead of scheduling by managing queues such that selection is O(1) instead of O(n) [10]. Another technique to reduce scheduling latency is to increase system clock frequencies [11]. The processing power of modern systems may tolerate the interrupt overhead [12]. Augmented CPU reservations help preserve fairness by counteracting the fact that incorrect tasks may be charged for interrupt processing [13].

There is a new feature for Linux called "scheduler hints [9][†]." Hints provide a method for a process to inform the scheduler that it is interactive or batch, or that it is needs a time-slice increase because it is holding a lock. The hint's anti-cheat policy is to limit its use to privileged applications. MDN is similar to hints, because they both provide information to help the scheduler make decisions. MDN, with its notion of deadline, is aimed at soft real-time performance and used in reaction to poor performance, while hints are used to prevent poor performance for batch or interactive processing.

There are many soft real-time schedulers for general-purpose and multimedia systems [14–18]. While these solutions directly support the needs of soft real-time applications, they lose the generality and convenience of best-effort scheduling by requiring processes to specify resource needs or complex QoS constraints *a priori*. The BEST and BeRate schedulers attempt to provide the same support without any API to the scheduler, adhering to the best-effort model where the task and scheduler know nothing about each other. Although there is a limit to the constraints that may be satisfied by purely best-effort models (and no guarantees), results are promising [1, 2]. MDN enhances these best-effort techniques, by providing a bit more information to the scheduler in the form of negative feedback.

Missed Deadline Notification is a feedback mechanism that allows the system to learn about the performance of an application. Using feedback information from applications is not a new scheduling approach. Some systems rely on feedback from metrics, such as the rate of progress or missed deadlines [19,20] to adjust the resource levels of applications. Other systems use traditional closed-loop feedback control for maintaining resource assignments [21, 22]. In some cases, the state of application structures, such as fill-levels of application buffers, may be used as feedback actuators [23]. A more general approach of using application-specific units of performance for actuators is used in some control-feedback systems [24]. Like these other systems, MDN uses feedback to make scheduling decisions, but with a simple algorithm and a best-effort approach for improving performance.

[†]At this time it is available as a patch for Linux, and is based on a feature used in Solaris.

6. CONCLUSION AND FUTURE WORK

The best-effort model continues to be attractive for both application developers and users because it is available in general-purpose systems and simple to use. Application resource constraints do not need to be known in advance in order to use best-effort schedulers. This means that best-effort systems do not provide guarantees for the timing of resource allocations, so the performance of soft real-time applications, such as multimedia, may suffer at the expense of applications without time constraints, even on systems that are not overloaded.

We implemented a Missed Deadline Notification mechanism for three best-effort schedulers. MDN allows schedulers to receive feedback about the performance of applications. The mechanism is general, while systems that provide an MDN service have the flexibility to incorporate any policy thats fit the goals of the system's scheduler. We find the MDN technique useful for improving the responsiveness of schedulers for servicing tasks with timing constraints, and for achieving stability during overload.

We have just begin to explore ways such best-effort techniques may be used to enhance the performance of systems, and it is possible to use MDN as an indicator of quality of service performance in other best-effort and soft real-time systems. To continue this work, the algorithms described in this paper should be tuned for more workloads. We will also investigate using MDN to control other aspects of task performance, such as allowing the scheduler to control not only the amount of missed deadlines, but the rate at which they occur.

ACKNOWLEDGMENTS

This research was funded in part by a DOE High-Performance Computer Science Fellowship. Thanks to the review committee and Pau Martí for comments and suggestions.

REFERENCES

1. S. Banachowski and S. Brandt, "The BEST scheduler for integrated processing of best-effort and soft real-time processes," in *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, pp. 46–60, Jan. 2002.
2. S. Banachowski and S. Brandt, "Better real-time response for time-share scheduling," in *Proceedings of the Eleventh International Workshop on Parallel & Distributed Real-Time Systems (WPDRTS)*, Apr. 2003.
3. F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," in *Proceedings of the AFIPS Fall Joint Computer Conference*, pp. 335–344, May 1962.
4. K. Jeffay and D. Bennett, "A rate-based execution abstraction for multimedia computing," in *Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, Apr. 1995.
5. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery* **20**, pp. 46–61, Jan. 1973.
6. J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "SVR4UNIX scheduler unacceptable for multimedia applications," in *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, 1993.
7. K. Ramamritham, C. Shen, O. Gonzalez, S. Sen, and S. B. Shirgurkar, "Using Windows NT for real-time applications: Experimental observations and recommendations," in *Proceedings of the Real-Time Technology and Applications Symposium (RTAS98)*, June 1998.
8. A. Morton, "Linux scheduling low-latency patch." http://www.zip.com.au/ akpm/linux/schedlat.html, Jan. 2001.
9. R. M. Love, "Linux preemptable kernel patch." http://www.tech9.net/rml/linux, Jan. 2003.
10. J. Nieh, C. Vail, and H. Zhong, "Virtual-time round-robin: An O(1) proportional share scheduler," in *Proceedings of the 2002 USENIX Annual Technical Conference*, pp. 245–259, 2001.
11. L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of the Linux kernel," in *Proceedings of the Real-Time Technology and Applications Symposium (RTAS02)*, Sept. 2002.
12. Y. Etsion, D. Tsafrir, and D. G. Feitelson, "Effects of clock resolution on the scheduling of real-time and interactive processes," Tech. Rep. 2001-14, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Nov. 2001.
13. J. Regehr and J. A. Stankovic, "Augmented CPU reservations: Towards predicatable execution on general-purpose operating systems," in *Proceedings of the Real-Time Technology and Applications Symposium (RTAS01)*, pp. 141–148, May 2001.

14. H. Chu and K. Nahrstedt, “CPU service classes for multimedia applications,” in *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS ’99)*, June 1999.
15. K. J. Duda and D. R. Cheriton, “Borrowed-virtual-time (BVT) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler,” in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP ’99)*, Dec. 1999.
16. J. Regehr and J. A. Stankovic, “HLS: A framework for composing soft real-time schedulers,” in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pp. 3–14, IEEE, (London, UK), Dec. 2001.
17. J. Nieh and M. Lam, “The design, implementation and evaluation of SMART: A scheduler for multimedia applications,” in *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP ’97)*, Oct. 1997.
18. D. K. Yau, “Performance evaluation of CPU isolation mechanisms in a multimedia OS kernel,” in *Proceedings of Multimedia Computing and Networking 2001 (MMCN ’01)*, Jan. 2001.
19. M. A. Rau and E. Smirni, “Adaptive CPU scheduling policies for mixed multimedia and best-effort workloads,” in *Proceedings of the 7th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS ’99)*, Mar. 1999.
20. S. Brandt and G. Nutt, “Flexible soft real-time processing in middleware,” *Real-Time Systems* **22**, pp. 77–118, 2002.
21. L. Abeni, L. Palopoli, G. Lipari, and J. Walpole, “Analysis of a reservation-based feedback scheduler,” in *Proceedings of the 23nd IEEE Real-Time Systems Symposium (RTSS 2002)*, Dec. 2002.
22. C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, “Feedback control real-time scheduling: Framework, modeling and algorithms,” *Real-Time Systems* **23**, July/September 2002.
23. D. C. Steere, A. Goel, J. Gruenberg, D. McNameee, C. Pu, and J. Walpole, “A feedback-driven proportion allocator for real-rate scheduling,” in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI’99)*, Feb. 1999.
24. A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, “Supporting time-sensitive applications on general-purpose operating systems,” in *Proceedings of the 5rd Symposium on Operating Systems Design and Implementation (OSDI’02)*, Dec. 2002.