Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes[†]

Scott A. Brandt Scott Banachowski Caixue Lin Timothy Bisson

Computer Science Department
University of California, Santa Cruz
{sbrandt,sbanacho,lcx,tbisson}@cs.ucsc.edu

Abstract

Real-time systems are growing in complexity and real-time and soft real-time applications are becoming common in general-purpose computing environments. Thus, there is a growing need for scheduling solutions that simultaneously support processes with a variety of different timeliness constraints. Toward this goal we have developed the Resource Allocation/Dispatching (RAD) *integrated scheduling model and the* Rate-Based Earliest Deadline (RBED) *integrated multi-class real-time scheduler based on this model. We present RAD and the RBED scheduler and formally prove the correctness of the operations that RBED employs. We then describe our implementation of RBED and present results demonstrating how RBED simultaneously and seamlessly supports hard real-time, soft real-time, and best-effort processes.*

1. Introduction

Modern embedded, special-purpose and general-purpose computing systems are becoming increasingly complex. At the same time, the traditional notions of best-effort and real-time processing have fractured into a spectrum of processing classes with different timeliness requirements including best-effort, desktop multimedia, soft real-time, firm real-time, adaptive soft real-time, rate-based, and traditional hard real-time. Many different schedulers support the important characteristics of each of these classes. However, none of them fully integrates the processing of these heterogeneous classes into a single scheduler.

Hierarchical systems can support multiple classes of processes, but do so through potentially complex hierarchies of schedulers. Because hierarchies partition the system among different schedulers, this strategy of integrating multiple schedulers may limit the ability to trade off merits of individual processes of different classes, and may complicate slack management. In contrast, a unified, integrated scheduling approach may make decisions while fully aware of the state of all processes.

To address this issue, we present a general model of real-time scheduling called *Resource Allocation/Dispatching (RAD)*. RAD explicitly separates the management of the amount of resources allocated to each process from the timing of the delivery of those resources. This separation allows the resource management to be precisely tailored to the needs of the individual processes. By allowing these two aspects of scheduling to be varied dynamically and independently, the RAD model fully captures the very different needs of the different real-time and non-real-time scheduling classes.

As a proof-of-concept, we have developed the *Rate-Based Earliest Deadline (RBED)* scheduler. RBED is a RAD scheduler that provides fully integrated scheduling of hard real-time, soft real-time, and best-effort processes. It uses dynamic rate-based resource allocation and dynamic period adjustments for fine-grained control of both resource amounts and timing.

2. The RAD CPU Scheduling Model

Traditional non-real-time CPU schedulers serve two roles in CPU resource management: *resource allocation* and *dispatching*. Resource allocation determines how much of the resource to give each process, while dispatching determines when to give the allocated resources to each process. Historically, the extreme efficiency requirements imposed on schedulers demanded unified approaches that merge these two important aspects of scheduling into a single mechanism. However, the recent rapid increase in CPU speeds enables the use of slightly more sophisticated scheduling mechanisms without significantly impacting overall system performance.

† This research was supported by Intel Corporation, a Department of Energy High Performance Computer Science Fellowship, and the USENIX Association

Although all schedulers implicitly address both resource allocation and dispatching to some degree, most do not explicitly separate their management and allow them to be independently adjusted at run-time. However, different classes of applications exhibit very different characteristics with respect to both resource allocation and delivery requirements. Dynamically and independently managing these two quantities thus enables the integrated scheduling of these different classes of processes.

Figure 1 presents a conceptual diagram of the resource allocation and dispatching requirements of each class of processes. Each axis represents the relative tightness of the requirements of the classes of processes, ranging from very loose to very tight. Hard real-time processes have extremely tight resource allocation and dispatching requirements. By contrast, best-effort processes have very loose resource allocation and dispatching requirements, generally being able to run as slow and sporadically as necessary without being thought of as having failed. However, even within best-effort scheduling there is variation in terms of these requirements. Non-interactive CPU-bound processes need greater amounts of CPU, but within very broad parameters they can use it in any size increments and at any time. I/O-bound processes, especially interactive ones, use relatively little CPU but need to receive it quickly once they have unblocked in order to provide good interactive responsiveness.

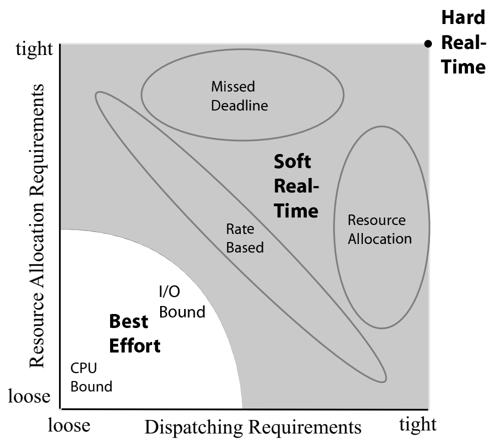


Figure 1: Resource allocation and dispatching requirements for different types of processes

Between hard real-time and best-effort lies a broad class of applications and systems referred to as soft real-time (SRT). This includes a variety of different systems with varying properties, all of which share the common property that resource allocation and/or dispatching requirements are looser relative to hard real-time. Figure 1 divides these into three broad sub-categories—Missed Deadline Soft Real-Time (MDSRT), Resource Adaptive Soft Real-Time (RASRT), and Rate-Based (RB)—depending upon which constraints are relaxed. MDSRT is real-time processing in which the time constraint is softened such that real-time processes may miss their deadlines in varying percentages or by varying degrees when all deadlines cannot be met [12, 17, 18]. By contrast, RASRT is real-time processing in which the resource allocation constraint is softened while attempting to minimize the number and amount by which deadlines are missed [4, 5, 12, 25]. In Rate-Based processing both resource allocation and dispatching can vary, but not completely independently; if more resources are provided a longer time may elapse before resources are again allocated and if less resources are allocated a shorter time may elapse before resources are again allocated [8, 11].

As this discussion demonstrates, the RAD model captures the important differences between these different classes of processes. The key to developing a fully integrated multi-class real-time RAD scheduler thus depends upon the development of mechanisms that will support different and varying degrees of requirements on both the resource allocation and dispatching. The Rate-Based Earliest Deadline (RBED) scheduler was developed to do exactly this.

3. The Rate-Based Earliest Deadline (RBED) Scheduler

RBED is a RAD scheduler for hard real-time, soft real-time, and best-effort processes. RBED resource allocation is accomplished via dynamic process rate adjustment. RBED dispatching is accomplished via dynamic application period adjustment. Based on the specific processing requirements of each process and the current system state, RBED assigns a target rate of progress and period to each process in the system. Both are enforced at runtime by a modified EDF scheduler that dispatches processes in EDF order but interrupts them via a programmable timer when they have exhausted their alloted CPU for the current period.

RBED allocates resources to processes as a percentage of the CPU such that the total allocated to all processes is less than or equal to 100%. Hard real-time processes have a period p and worst-case execution time e and are either granted their desired rate e/p or are rejected if insufficient resources are available. Soft real-time processes are given their desired rate e/p if possible, or are given less, possibly based upon a QoS specification if one is available. Like hard real-time processes, rate-based processes are given the rate that they request if possible, or are rejected. The rate of each best-effort process is a calculated share of the rate that remains after the rates of the other processes in the system have been determined. A reservation mechanism can guarantee that a minimum or maximum allocation is available to a particular class of processes, ensuring, for example, that there are always some resources available to the best-effort processes.

RBED assigns periods in cases where application do not already have them. Hard and soft real-time tasks have pre-specified periods. Periods for rate-based tasks are determined based on the processing requirements of the particular task. At deadlines, a process' desired and actual resource usage are equal, and so the difference between the desired and actual resource usage at non-deadline times are bounded by the choice of period. Thus the necessary period for rate-based tasks can be determined directly by the amount they are allowed to stray from their target rate. For example, in an audio player application this is a function of the data sampling rate and the size of the audio device's memory buffer. For best-effort tasks, a semi-arbitrary period is assigned to ensure a high degree of responsiveness, when needed.

Thus, by allocating the resources appropriately, choosing appropriate deadlines, and using a programmable timer, RBED presents to EDF exactly what it wants and guarantees that all deadlines are always met, that all rate constraints are met, and that all processes receive the correct amount of CPU. However, unlike processes in traditional hard real-time systems, the rates of soft real-time and best-effort processes may change as processes enter and leave the system, and the periods of soft real-time processes may change as they adjust to the available resources. The next section provides proofs that we can maintaining the correctness of EDF under these conditions.

4. RBED Theoretical Background

Because RBED uses earliest deadline first (EDF) to schedule periodic tasks[1], its theory is based on well-known real-time scheduling principles. However, the standard definition of EDF and proofs of its correctness and optimality usually assume that the worst-case execution time and the period of a task is fixed during its lifetime, while RBED dynamically adjusts both of these properties. In this section we prove that EDF functions correctly with the dynamically changing rates and periods of RBED.

When either rate or period change (or both), the task undergoes a *mode change*. Previous works describe the constraints under which either fixed-priority [22, 24] or proportional share [23, 2] scheduling algorithms allow mode changes. In this section, we determine the impact on schedulability when a task changes mode at arbitrary times under EDF, and describe the conditions under which EDF still guarantees deadlines after a mode change. Understanding these constraints allows us to create flexible schedules for the dynamic workloads presented to the RBED scheduler.

We introduce a slightly different task model than supplied in the Liu and Layland proof [15]. In the original proof a task consists of a sequence of periodically released jobs whose deadlines equal their release times plus the task period. The RBED task model is the same, except that each job in a task may have a different period. We first show that under this model an EDF schedule remains feasible as long as the utilization is constant, and then we relax even this requirement. We use the following notation: a task T_i consists of sequential jobs $J_{i,k}$, where each job has a release time $r_{i,k}$, period $p_{i,k}$, deadline $d_{i,k}$, and worst-case execution time $e_{i,k}$. For any k, utilization[2] $u_i = e_{i,k}/p_{i,k}$, $r_{i,k} = d_{i,k-1}$, and $d_{i,k} = r_{i,k} + p_{i,k}$. The total utilization of the system is $U = \sum_{i \in \mathbf{T}} u_i$, where $\mathbf{T}$ is the set of all tasks. In some cases, subscripts are dropped if the meaning is obvious.

Theorem 1 *The earliest deadline first (EDF) algorithm will determine a feasible schedule if $U \leq 1$.*

Our modified task model does not invalidate this theory; for reference a similar proof using the new task model is provided in Appendix A. The benefit of the new model is that it supports arbitrary period changes at job deadlines. The restriction that period changes occur only at deadline boundaries is relaxed below.

When a job completes and a new job is released with a different period, it is equivalent to the task leaving at the same instance that a new task of the same utilization enters the system. In this sense, it is safe to consider the utilization of a departing task as part of the unallocated CPU $(1-U)$ when its last deadline is reached.

Corollary 1.1 *Given a feasible EDF schedule, at any time a task with utilization $\leq (1-U)$ may enter the system and the schedule remains feasible.*

The proof is implicit in the proof for Theorem 1, because it holds for tasks of arbitrary starting times.

4.1. Increasing Utilization or Period

Theorem 2 *Given a feasible EDF schedule, at any time a task T_i may increase its utilization by an amount up to $1-U$ without causing any task to miss deadlines in the resulting EDF schedule.*

Figure 2 shows the effect of increasing the utilization of an already released job. At time t, the instantaneous utilization of the task increases from u to u', but over the life of the job, the job effectively consumes a utilization $u_{effective} = ((t-r) \times u_i + (d-t) \times u')/(d-r)$.

1 The terms "task" and "process" are used interchangeably throughout this paper.

2 Utilization is synonymous with rate. We are using the term utilization and the symbol u in the proofs instead of rate and r, because the symbol traditionally refers to release time.

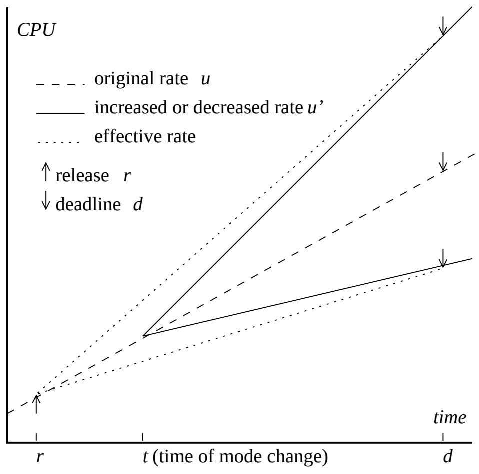


Figure 2: Increasing or decreasing the rate of an already released job

The proof uses Corollary 1.1; the schedule resulting from increasing the utilization of a task is equivalent to a schedule in which another task having the same deadlines enters the system. Assume that at time t task T_i, which already released a job at time r_i, wishes to increase its utilization to $u'_i = u_i + \Delta$, where $\Delta \leq 1 - U$. To meet its deadline with the new utilization, it requires $e'_i = u_{effective}(d_i - r_i) = u_i(t - r_i) + (u_i + \Delta)(d_i - t)$ CPU in the interval $(r_i, d_i]$. At this time, introduce task T'_i with period $p'_i = d_i - t$ and utilization Δ. By Corollary 1.1, the schedule remains feasible. During the interval $(t, d_i]$, the CPU assigned to both T_i and T'_i is used for T_i, and over the interval $(r_i, d_i]$ the total CPU allocated to T_i equals $u_i(d_i - r_i) + \Delta(d_i - t) = e'_i$. For subsequent periods, the tasks T_i and T'_i may be merged into a single task of utilization $u_i + \Delta$.

Theorem 3 *Given a feasible EDF schedule, at any time a task T_i may increase its period without causing any task to miss deadlines in the resulting EDF schedule.*

The task model described in Theorem 1 allows subsequent jobs to have different periods. We now consider increasing the period of an already released job $J_{i,k}$, effectively extending its current deadline from $d_{i,k}$ to $d'_{i,k}$. If, in the unchanged schedule, $J_{i,k}$ is followed by a job released at $d_{i,k}$ with period $p_{i,k+1} = d'_{i,k} - d_{i,k}$, the schedule is feasible. In this schedule, the total CPU consumed by both jobs over the interval $(r_{i,k}, d'_{i,k}]$ is the same as if the original period of $J_{i,k}$ was the new period. Instead of releasing the jobs sequentially, we wish to simply extend the current job's deadline, and ensure that changing it on-the-fly does not cause another task to miss a deadline due to changes in dispatch order.

A simple "interval swapping" argument shows that EDF does not miss any deadlines in the new resulting schedule: all jobs with deadlines before $d_{i,k}$ will execute in the same intervals before and after the period change. Any jobs with deadlines at or after $d_{i,k}$, but before $d'_{i,k}$, will be dispatched in earlier intervals under the new schedule, so are not at risk of missing deadlines. And because EDF is work conserving, any jobs with deadlines after $d_{i,k}$ that receive CPU in the interval $(t, d'_{i,k}]$ under the previous schedule receive the same amount of CPU in this interval of the new schedule.

4.2. Decreasing Utilization or Period

As long as the requested utilization does not exceed the CPU bandwidth, increasing the utilization or period of an already released job is unconstrained. This is not the case when decreasing utilization or period. For example, in an EDF schedule it is possible for a task to meet its deadline without any laxity. Clearly, this task cannot meet an earlier deadline, as it just barely makes its existing deadline. This section describes the conditions that allow decreasing the utilization or period of an already-released job. These conditions depend only on the state of the task, so that it may change mode without needing to determine the state of any other tasks.

The *lag* of a job is the difference between its ideal and actual service time. The ideal service time is the amount of CPU the job receives assuming ideal (fluid) scheduling, and equals $u(t - r)$. At t, if the job actually consumes x CPU since time r, $lag(t, x) = u(t - r) - x$. A negative lag means a task is proceeding ahead of its ideal service time. The lag concept is often used when characterizing algorithms that approximate fair sharing—because proportional share algorithms bound lag, it is useful for proving mode change constraints.

If a task meets its deadline, then at its deadline it has zero lag ($lag(r + p, e) = e/p((r + p) - r) - e = 0$). Depending on the schedule, there may be several times during a job's lifetime that lag is zero. If the task leaves the system when its lag is zero, the utilization may be immediately reclaimed for use by any task.

Lemma 1 *Given a feasible EDF schedule, if a task with zero lag leaves, its utilization may be used by a new task, and the resulting EDF schedule remains feasible.*

At a task's deadline, this is already known to be true. Here we sketch the proof when a task is not at its deadline. If job $J_{i,n}$ has zero lag at time t, its current service time is $x_{i,n} = u_i(t - d_{i,n-1})$. At this time the current job leaves (and hence has no deadline to miss) and the task releases a new job of period $p_{i,n+1}$. At the deadline of a later job $J_{i,m}$, the total service time requested by the task is (with ϕ_i equal to the time of the first release):

$$u_i(d_{i,1} - \phi_i) + u_i(d_{i,2} - d_{i,1}) + \cdots + u_i(t - d_{i,n-1}) \\ + u_i(d_{i,n+1} - t) + \cdots + u_i(d_{i,m} - d_{i,m-1}) = u_i(d_{i,m} - \phi_i)$$

The remainder of the proof follows that of Theorem 1 supplied in Appendix A. The total service demand of the task remains the same as in Equation (A.1), so a missed deadline requires $U > 1$, a contradiction.

A task may reduce its utilization at any time, without affecting the feasibility of the existing task set, because the overall service time is reduced. However, because the scheduler assumes that jobs will complete their periods, it is not apparent that the utilization freed by this task is available to other tasks until its existing period completes. However, if the utilization of a task is decreased within a constraint, we see that EDF can guarantee the freed utilization immediately to other tasks.

Theorem 4 *Given a feasible EDF schedule, if at time t task T_i decreases its utilization to $u_i' = u_i - \Delta$ such that $\Delta \leq x_i/(t - r_i)$, the freed utilization Δ is available to other tasks and the schedule remains feasible.*

Figure 2 shows the effect of decreasing the utilization of an already released job. At time t, the instantaneous utilization of the task decreases from u to u', but over the life of the job, the job effectively consumes a utilization $u_{effective} = ((t-r) \times u_i + (d-t) \times u')/(d-r)$.

The schedule described in Theorem 4 is equivalent to the following hypothetical EDF schedule, which is feasible: At time r_i, imagine that instead of T_i releasing a job with u_i, two tasks release jobs: T_a's job has deadline d_i and utilization u_a and T_b's job has deadline t and utilization u_b, where $u_a + u_b = u_i$. At time t T_b may leave the system and u_b is available for use by other tasks. The service time received by T_b at t is $u_b(t - r_i)$.

In the actual schedule, at time t we wish to make Δ of task $T_i's$ utilization available to another task. This is equivalent to letting a portion of the task leave the system, as though it met a hypothetical deadline. The service time of T_i at t must exceed the service time of the hypothetical task leaving the system, i.e. $x_i \geq u_b(t - r_i)$. Letting $u_b = \Delta$, if $\Delta \leq x_i/(t - r_i)$, the resulting schedule is feasible if another task uses Δ at t.

Theorem 5 *Given a feasible EDF schedule, if a currently released job $J_{i,n}$ has negative lag at time t (the task is over-allocated), it may shorten its current deadline to at most x_i/u_i, and the resulting EDF schedule remains feasible.*

Lowering the period is possible if the task is currently over-allocated. According to Lemma 1, if the lag equals zero, then it is safe to change the deadline. Note that if the task is over-allocated, it will reach zero lag by idling for $-lag(t, x_i)/u_i$ units of time. Thus, the deadline of a job may be reduced to any value of d_i' such that its current service time does not exceed its utilization, i.e. $x_i/d_i' \leq u_i$, so $d_i' \geq x_i/u_i$.

When the deadline is decreased to d_i', tasks with deadlines before d_i' are unaffected. If in the original schedule no CPU is allocated to $J_{i,n}$ between t and d_i', then the new schedule is feasible, because it is exactly the same schedule produced if a period change occurs at d_i', when it is permissible by Lemma 1. Thus, tasks at risk of missing deadlines are those with deadlines after d_i' but before d_i, as they would preempt T_i in the original schedule. Note that if any of these jobs were released before $r_{i,n}$, then $J_{i,n}$ would not have received CPU before them, and so it would not have negative lag; therefore any jobs at risk were released in the interval $(r_{i,n}, d_i]$. The proof proceeds as follows: if any of these "at risk" tasks miss a deadline after d_i', the utilization exceeds 1, which is a contradiction. The details are provided in Appendix B.

In some cases it may be feasible to lower the deadline of a task even if it has positive lag. However this is not always so, for example when the task has no laxity. To determine if a task with positive lag may safely reduce its period requires knowledge of the state of other tasks in the system, because it must be determined if tasks with pending deadlines have enough slack to potentially change dispatch order; it is therefore simpler to use the loose constraints of Theorem 5.

4.3. Superposition of Mode Changes

The above section examines the effect of a single mode change of either utilization or period, while the other parameter is held constant. It is possible for a task to instantaneously change both the utilization and period, and the bounds are determined by a piecewise combination of the above rules. When determining the bounds of feasibility, it is useful to apply the change to the less constrained parameter first.

Increasing Utilization and Period Increasing either period or utilization is unbounded by the task state, so it is always feasible to do both operations as long as $U \leq 1$.

Decreasing Utilization, Increasing Period Increasing the period is unbounded, but the amount that utilization may decrease is a function of service time (Theorem 4). Changing the period has no effect on service time, so the bound that utilization may be lowered depends only on the received service time, and remains the same.

Increasing Utilization, Decreasing Period The amount that a period may be lowered depends on the new effective utilization of the utilization change. Figure 2 shows that after the job increases its utilization, although the task begins processing at its new utilization, for the currently released job the effective utilization is an intermediate value between the old and new period. The effective period of the job is $u_{effective} = u(t-r) + u'(d-t)/(d-r)$, and the bound on lowering the period of this job is $x/u_{effective}$.

Decreasing Utilization and Period If a task is under-allocated it cannot lower its deadline. Nevertheless, after lowering utilization it is possible that under the new $u_{effective}$ the lag is no longer positive, and the period may now be lowered to $x/u_{effective}$.

Thus by following the constraints imposed by the above theorems, it is possible to vary the rate and period of processes as required by RBED while still maintaining the correctness of EDF. This enables the implementation of the RBED scheduler, as discussed in the following section.

5. RBED Implementation

After simulating all of the RBED operations, we implemented a proof-of-concept version of RBED in Linux 2.4.20 on an Intel Pentium 4 platform. RBED supports all of the processing classes shown in Figure 1 but due to space limitations this discussion is limited to our support of hard real-time, missed deadline soft real-time, and best-effort processes. The changes to the Linux kernel include less than 550 lines of modified or added code. We added several attributes to the process state structures for RBED bookkeeping: process type, period, worst case execution time (WCET), weight, and deadline. We also added two system calls to allow processes to interface with the RBED scheduler: **set_rbed_scheduler(deadline_type, period, WCET)** and **rbed_deadline_met(void)**.

All processes default to the best-effort scheduling class. set_rbed_schedule() turns a process into a real-time process. The arguments specify the period, worst-case execution time (WCET), and whether the deadlines are hard or soft. The target resource rate (u_{target}) of a process is defined as $\frac{WCET}{period}$. A hard real-time process is guaranteed to receive an actual resource rate (u_{actual}) equal to u_{target} if enough CPU is available, otherwise the process is not admitted as a real-time process. A soft real-time process receives a u_{actual} equal to its u_{target} or less, depending on the available resources. Set_rbed_schedule() returns the amount of utilization available to the process—if a soft real-time task receives less than it requested it may continue, adapt to the available resources, or abort.

Real-time processes call rbed_deadline_met() when they finish a periodic computation. It returns true or false depending upon whether or not the process met its deadline. This allows processes to synchronize their periods using system clocks and to determine if a deadline is missed without having to compute the time. If this function is called before a deadline, the process suspends until its next release time. If a soft-real time computation exceeds its WCET, it will call this function after its deadline, in which case it returns false, and the application may deal with the missed deadline appropriately.

There are two main components of the RBED scheduler: a resource allocator [14], and an EDF-based dispatcher. The resource allocator sets the periods and WCET of all tasks so that the EDF scheduler is never overloaded. The resource allocation component is triggered whenever a process enters or leaves the system or a best-effort process blocks or unblocks. The algorithm sets u_{actual} for each task according to its requirements. Hard real-time tasks receive their requested utilization, soft real-time task receive the leftover CPU, minus the portion of CPU (β) reserved for best-effort processes. Best-effort processes receive utilization in proportion to their weights, described in more detail in Section 5.1. The pseudo-code in Figure 3 describes the allocation policies. In this code, $U_{\mathbf{X}} = \sum_{p \in \mathbf{X}} p.u_{actual}$, where $\mathbf{X}$ is the set of all tasks of type X, and $W_{\mathbf{BE}} = \sum_{p \in \mathbf{BE}} p.weight$. Note that in this calculation we count only runnable best-effort processes – blocked best-effort processes have an effective weight of zero.

```
do_resource_allocation(process $p$){
  switch($p.type$){
    case Hard real-time:
        $p.u_{actual} = p.u_{target}$;
        break;
    case Soft real-time:
        $p.u_{actual} = min(p.u_{target}, (1 - \beta - U_{\mathbf{HRT}}) \frac{p.u_{target}}{U_{\mathbf{SRT}}})$;
        break;
    case Best-effort:
        $p.u_{actual} = max(\beta, 1 - U_{\mathbf{HRT}} - U_{\mathbf{SRT}}) \frac{p.weight}{W_{\mathbf{BE}}}$;
} }
```

Figure 3: Resource allocation pseudo-code. β is the minimum resource rate reserved for best-effort processes.

To ensure a feasible schedule, a process must not overrun its worst-case execution time. RBED uses a one-shot timer to interrupt a task when it consumes its WCET. If a task reaches its WCET, the one-shot timer interrupt handler will preempt the process and advance its absolute deadline to the end of its subsequent period. RBED uses the Advanced Programmable Interrupt Controller, which is capable of better than microsecond precision with little overhead, for its one-shot timer. This allows the kernel to continue to use the regular programmable interrupt timer for system time services.

5.1. Scheduling Best-effort Processes

Best-effort schedulers attempt to provide fair allocation of the CPU over the long term. In addition, to improve the responsiveness of I/O-bound applications, they give a short-term “boost” to processes immediately after they block. One goal of the RBED scheduler is to preserve this behavior for best-effort processes.

Unlike real-time processes, best-effort processes lack the time constraints required by deadline-based scheduling algorithms. RBED therefore assigns dynamic pseudo-periods to best-effort processes. The period is equal to

```
rbed_schedule() {
  1. If RT process enters or leaves and system is overloaded
     then for all $p$ in RT, do_resource_allocation($p$).
  2. Use EDF to dispatch a process.
  3. If the selected process $p$ is in BE
     /*do lazy allocation*/
     do_resource_allocation($p$).
  4. Clear one-shot timer for the previous process and
     reset one-shot timer for the selected process.
  5. Context switch if needed. }
```

Figure 4: Pseudo-code for rbed_schedule()

$N_{\mathbf{BE}} \times$ quantum, where $N_{\mathbf{BE}}$ is the number of runnable best-effort processes and the quantum reflects the scheduling quantum of a time-share system. RBED, like the version of Linux it was implemented in, assigns a default quantum of 60 ms.

Every best-effort process has a weight, which is the rate it consumes CPU relative to other runnable best-effort processes. Figure 3 shows how RBED maps weight into utilization. Given u_{actual}, the WCET is $N_{\mathbf{BE}} \times \text{quantum} \times u_{actual}$.

A task's weight fluctuates depending on its state. Whenever a best-effort process consumes its WCET without blocking, its weight is set to 0. A process that blocks before using its entire WCET retains its weight. When no runnable processes have a nonzero weight, all runnable process's weights are set to one, and all blocked processes receive $weight = weight/2 + 6$ (which is bounded to a maximum of 12). The static value 6 provides a boost similar to that of Linux.

In order to avoid frequent resource allocation recomputation any time a process enters or leaves the system or a best-effort process blocks or reenters the ready queue, RBED lazily applies the resource allocation algorithm to each best-effort task when it is selected for dispatch. Figure 4 shows the pseudo-code for the scheduler. Parameters of already released jobs are set within the constraints defined in Section 4.

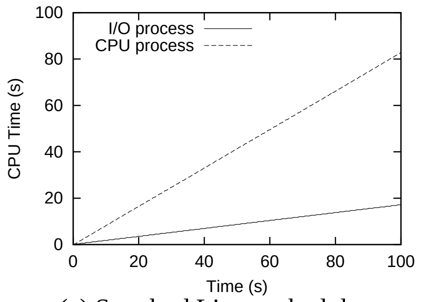
When real-time tasks complete the processing for a period, the next job is not released until the period expires. In our RBED prototype, best-effort tasks are treated differently: instead of suspending a best-effort job when it consumes its WCET, the next job is immediately released (with the deadline set to the end of its next pseudo-period). This allows best-effort processes to consume all of the dynamic slack inserted into the schedule by real-time processes. This approach is similar to the those used in other systems, such as in Portable RK [19], which runs depleted tasks at a low background priority. However RBED does not need to maintain a background scheduling algorithm; the resource allocator only needs to release the task's next job early, using the later deadline to effect a background priority with no change to the dispatcher or any other tasks' parameters. This simple technique distributes the slack among the best-effort tasks according to their relative weights. However, it does not allow other classes of processes (notably soft real-time) to take advantage of dynamic slack. In the future we plan to examine techniques for making regularly available dynamic slack available to soft real-time processes as well.

6. Performance Measurements

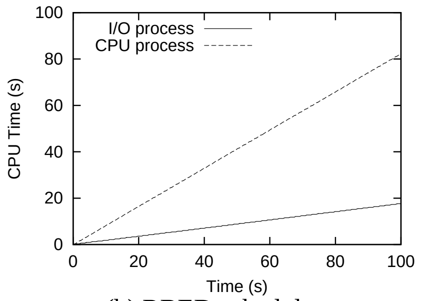
To characterize the performance of RBED, we compare it to the Linux scheduler and to a hierarchical EDF/best-effort scheduler we developed called *EDF-Linux*. Both Linux and EDF-Linux use the 2.4.20 kernel. EDF-Linux maintains two ready queues, one for periodic real-time tasks, scheduled by EDF, and another for best-effort tasks, scheduled by the default Linux scheduler whenever the real-time queue is empty. All experiments were performed on a standard PC Desktop equipped with a 1 GHz Pentium III processor, 512MB RAM, and a 40GB hard drive. In developing our prototype we have run the system over long periods of time. Our general impression is that the scheduler works well. Best-effort tasks exhibit "normal" behavior (with default scheduling quanta of 60 ms, the same as the Linux scheduler) and are never completely starved, real-time tasks meet their deadlines, and soft real-time tasks meet their deadlines or run at lower performance levels depending upon the amount of resources available. Below we present a series of snapshots that illustrate how RBED performs in practice. For simplicity we have drawn the graphs relative to an origin starting at (0,0) even though the snapshots were taken from the middle of longer executions.

Real-time workloads for these tests were generated by a tool we developed to easily generate periodic workloads with different utilization and timing behavior. Given a specification of a desired period and rate (where rate is $\frac{WCET}{period}$), the tool generates periodic hard real-time or soft real-time processes with constant or variable (with a normal distribution) execution times. We arbitrarily reserve a minimum of 5% of the CPU for best-effort processes, enough to provide a functional interactive system for running command shells during experiments.

Figures 5 shows the performance of the Linux and RBED schedulers when running two best-effort processes, one CPU-bound and one I/O-bound. The CPU-bound process performs a floating point calculation in a tight loop, while the I/O-bound process repeatedly computes for 300 ms then sleeps for 1200 ms. The RBED results are nearly identical to the Linux results. Figure 6 shows closeup views of the behavior when the I/O-bound process reenters the ready queue after blocking (a constant offset was added to the y-axis of the I/O process's data set to view the two lines in the same axes). Again the results are nearly identical. When an I/O-bound process blocks, Linux increases its scheduling quantum and priority. When it awakens, the I/O-bound process receives a long quantum of about 110 ms. After this quantum, the I/O-bound and CPU-bound process share the CPU, each using 60 ms quanta. When both quanta expire, they are both replenished to 60 ms. When two processes have equal priority, Linux prefers to avoid a context switch, so the same process consumes another quantum for a total of 120 ms. The RBED scheduler uses a similar boost formula for sleeping best-effort processes, as described in Section 5.1. The I/O-bound process has a weight of 11 when it awakes, and so its WCET is adjusted to $(2)(60)(\frac{11}{11+1}) = 110$ ms. After this computation, the weight of the I/O-bound process is reduced to 1, the same as the CPU-bound process, and the two processes share the CPU using 60 ms quanta (WCET$= (2)(60)\frac{1}{(1+1)} = 60$).



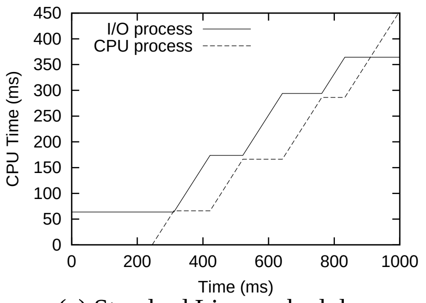
(a) Standard Linux scheduler



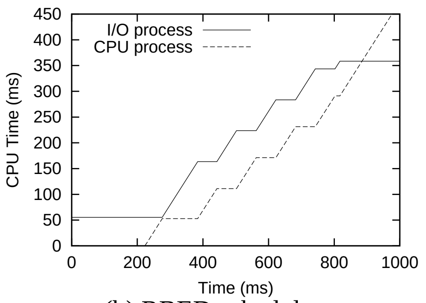
(b) RBED scheduler

Figure 5: I/O vs. CPU-bound processes on the Linux and RBED schedulers

Figure 7 shows multiple soft real-time processes with a single best-effort process. In Figures 7(a) and (b), the three soft real-time processes have (period,rate) equal to (0.2s,25%), (0.5s,30%) and (1.0s,35%). Each of the four processes receive 25% of the CPU under Linux's scheduling algorithm. In this case, SRT-1 meets its deadlines, but the other two soft real-time processes miss all of their deadlines. By contrast, RBED ensures that each of the soft real-time processes receives its required re-



(a) Standard Linux scheduler

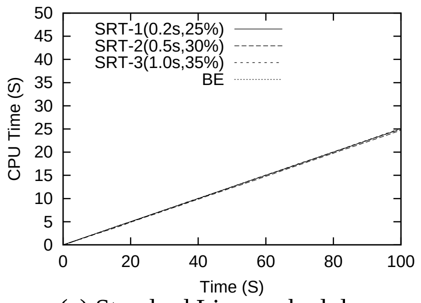


(b) RBED scheduler

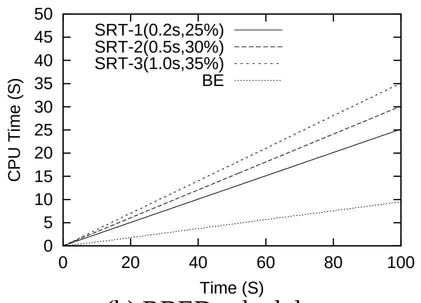
Figure 6: Closeup of I/O vs. CPU-bound processes on the Linux and RBED schedulers

sources $(25\%, 30\%, 30\%)$ and meets all of its deadlines and the best-effort process receives the remaining resources (10%). Figure 7(c) shows how RBED manages SRT resources when the system undergoes load changes, due to applications entering and leaving the system. Initially SRT-1 runs with resource rate of 45% as a best-effort process receives the leftover CPU (55%). After 40 seconds, SRT-2 enters and is allocated 45%, while the best-effort rate drops to 10%. At 80 seconds, SRT-3 enters and forces the other two soft real-time processes to decrease their resource rates so that the system is not overloaded. As a result, each soft real-time process receives a resource rate of 31.6%, and the best-effort process receives the best-effort reservation β. After the 109 seconds the three soft real-time processes begin to leave the system and the rates of the other processes increase accordingly.

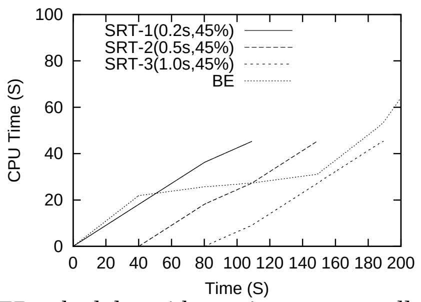
Figure 8 shows RBED running two hard real-time processes (20%, 60%), a soft real-time process (period, rate)=(0.5s,40%), and a best-effort process. The two hard real-time processes receive their required resources, unaffected by the presence of the soft real-time or best-effort processes. Because the available resource for the soft real-time process is less than 40%, the scheduler dynamically extends its period, thus reducing its



(a) Standard Linux scheduler



(b) RBED scheduler



(c) RBED scheduler with varying resource allocations

Figure 7: Soft real-time processes on the Linux and RBED schedulers

resource rate to 14%, and the best-effort process still receives at least β of the CPU, or 6% in this case.

Unlike best-effort process running in the background of a real-time scheduler, RBED's rate reservation and pseudo-periods for best-effort processes guarantee that best-effort response times will always be acceptable. Figure 9 shows the response and completion times of a best-effort process running with a real-time process in RBED and EDF-Linux. Response time is the time between when a job enters the ready queue and when it is first scheduled, and completion time is the total time between when a job enters the ready queue and is completed. The best-effort process has 10 ms burst times and block times ranging from 1 microsecond to 1 second based on a pseudo-random distribution seeded by the same initial value in all runs. The real-time process has

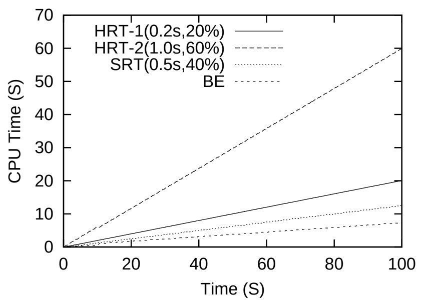
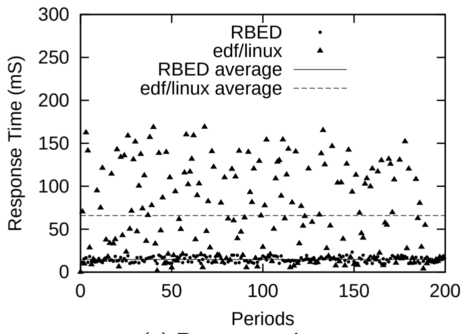
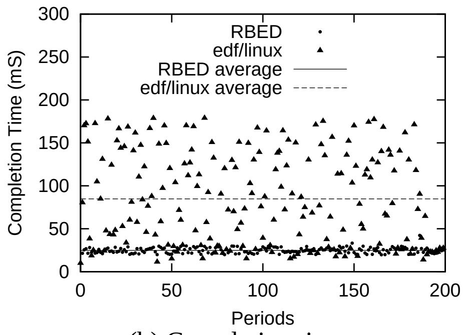


Figure 8: Scheduling hard real-time processes with soft real-time and best-effort processes

a period of 190 ms and WCET=150 ms. The RBED scheduler ensures that the real-time process meets all of its deadlines and provides much better average best-effort response and completion times than EDF-Linux.



(a) Response time



(b) Completion time

Figure 9: Synthetic best-effort process running with real-time process(190 ms, 150 ms) on RBED and EDF-Linux

In practice, many real-time systems use Rate Monotonic-based static priority schemes to reduce the runtime overhead of scheduling decisions. RBED uses EDF scheduling, which can incur more overhead because priorities change dynamically. However, EDF allows RBED to always utilize up to 100% of the CPU for real-time processes. Ever-increasing CPU speeds also enable somewhat more complex decision-making without significantly increasing system overhead. Measurements on our proof-of-concept implementation show the time it takes to allocate resources and schedule processes in RBED is typically two to two-and-a-half times greater than in Linux. We feel that this small amount of additional overhead is acceptable given the added capabilities that RBED provides.

7. Related Work

There exist many scheduling algorithms and systems developed specifically to handle the workloads of hard real-time or soft real-time applications. For scheduling a mix of applications, a typical approach combines several algorithms, by assigning priorities to each scheduler or using a high-level scheduler to dynamically determine which scheduler selects a job. RAD differs by using a single scheduler for multiple classes of applications, and dynamically adapting the requests made to the scheduler to meet the needs of the tasks.

A predecessor to RAD is the CPU reservation [17]. CPU reservations allow a process or server to receive a service guarantee over a scheduling interval, and are often enforced with a proportional-share scheduling algorithm [6, 12, 23]. Reservations make admission control policies simple, and sharing algorithms provide isolation between tasks, sheltering from overruns of other tasks. Reservations support applications with rates and deadlines requirements. Because reservations are relatively static, they are not as flexible at supporting all classes of applications. The Resource Kernel [20] extends reservations to include multiple timing constraints, by advocating a separation between resource specification and resource management. The RAD model further separates resource management into allocation and dispatching, making scheduling of resources flexible for mixed workloads.

Proportional-share scheduling is widely employed in real-time systems, because it is a natural computation model for periodic tasks. Proportional-share algorithms have been adapted to solve specific scheduling latency problems facing soft real-time applications such as multimedia [8, 18]. Proportional-sharing of CPU is similar to flow-based packet schedulers such as WF^{2}Q [3], because awareness of throughput is used to make scheduling decisions. While the goal of most proportional-share algorithms is maintaining constant rate (i.e. a fluid model) over any interval, the CBS algorithm relaxes the fairness constraint [1], only ensuring that enough proportion is received at deadlines. RBED uses this latter approach when scheduling tasks with deadlines.

Proportional-share schedulers have been employed to split the CPU between multiple scheduling algorithms—this way a system may support multiple scheduling paradigms simultaneously [9]. Each application is assigned to the scheduler using the policy best suited for its type. In hierarchical schemes, lower-level schedulers receives bandwidth allocated by a higher-level scheduler. In fact, many general-purpose operating systems use this approach to add real-time capability, running the best-effort scheduler as a low-priority task in a fixed-priority scheduler [10, 13, 27].

A framework with goals similar to RAD, but with a different approach, is HLS [21]. HLS is used to compose arbitrary hierarchies of existing schedulers in order to execute mixed class workloads. The framework provides rules for determining if a given hierarchy of schedulers gives the desired performance. Hierarchical scheduling poses many engineering difficulties, and ultimately no matter how complex the graph of schedulers, resulting in a single one-dimensional schedule. RBED produces a schedule to handle multiple classes of applications, without the added complexity of understanding interactions of multiple schedulers.

The RED-Linux system [26] aims to support three scheduling paradigms under a single scheduler. The paradigms include priority, time and share-driven scheduling. However, the scheduler emulates only one scheduling paradigm at a time, and is limited in the classes it supports. RAD assumes application resource constraints are independent of scheduling paradigm, and can support multiple classes of applications simultaneously.

In RBED, we must handle dynamic workloads, and handle changes to the system by adjusting the rates and deadlines assigned to applications. For adaptive tasks that may change their rate, Buttazzo et al. formulated an algorithm in which rate changes are modeled using spring coefficients [7]. This novel approach incorporates constraints for dynamically changing resource assignments. Our goal is similar, but the approach used by RBED differs as all resource assignments are changed within an EDF framework.

8. Conclusions

Modern real-time and non-real-time systems are becoming larger and more complex and at the same time multimedia applications have become ubiquitous in general-purpose computing environments. These trends are driving a need for integrated scheduling solutions that can simultaneously provide the flexibility and responsiveness required for best-effort processing, the guarantees required for hard real-time processing, and the combination of guarantees and flexibility required for various types of soft real-time processing. The RAD model explicitly separates and dynamically varies the resource allocation and resource delivery timing provided by all schedulers. It explains the key differences between these different classes of processes and provides a model for how to develop schedulers capable of simultaneously executing processes from all of them.

Our prototype RBED scheduler is based on the RAD model. It uses dynamic rate-based resource allocation and dynamic period adjustment to achieve the separate control of these two aspects of scheduling. Processes are managed at runtime using a variant of EDF that enforces resource allocations.

Our results show that RBED is capable of simultaneously supporting hard real-time, soft real-time, and best-effort processes. Its management of best-effort processes closely mirrors that of Linux, its management of soft real-time processes is better than that of Linux, and it provides guaranteed hard real-time performance. In addition, RBED's support of best-effort processes is shown to be better than that of two-level hierarchical systems in which best-effort processes are run in the background of hard real-time processes, and RBED's runtime overhead is only slightly greater than that of Linux.

References

[1] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs. proportional share resource allocation. In *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, June 1999.

[2] S. K. Baruah, J. E. Gehrke, C. G. Plaxton, I. Stoica, H. Abdel-Wahab, and K. Jaffay. Fair on-line scheduling of a dynamic set of tasks on a single resource. *Information Processing Letters*, 64(1):43–51, Oct. 1997.

[3] J. C. Bennett and H. Zhang. WF2Q: Worst-case fair weighted fair queueing. In *Proceedings of the IEEE INFOCOM*, Mar. 1996.

[4] S. Brandt and G. Nutt. Flexible soft real-time processing in middleware. *Real-Time Systems*, 22:77–118, 2002.

[5] S. Brandt, G. Nutt, T. Berk, and J. 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, Dec. 1998.

[6] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of the 5th ACM International Multimedia Conference*, Nov. 1997.

[7] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, Mar. 2002.

[8] 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.

[9] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, Oct. 1996.

[10] 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.

[11] 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.

[12] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 198–211, Oct. 1997.

[13] S. Khanna, M. Sebrée, and J. Zolnowsky. Realtime scheduling in SunOS 5.0. In *Proceedings of the Winter 1992 USENIX Technical Conference*, pages 375—390. USENIX, Jan. 1992.

[14] C. Lin. Managing the soft real-time processes in RBED. Master's thesis, University of California, Santa Cruz, Mar. 2003.

[15] 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(1):46–61, Jan. 1973.

[16] J. W. Liu. *Real-Time Systems*. Prentice–Hall, 2000.

[17] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the 1994 IEEE International Conference on Multimedia Computing and Systems (ICMCS '94)*, pages 90–99, May 1994.

[18] 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.

[19] S. Oikawa and R. Rajkumar. Portable RK: A portable resource kernel for guaranteed and enforced timing behavior. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS99)*, June 1999.

[20] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *Proceedings of Multimedia Computing and Networking 2001 (MMCN '98)*, Jan. 1998.

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

[22] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *The Journal of Real-Time Systems*, 1(3):244–264, 1989.

[23] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Buruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource

allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS 1996)*, pages 288–299, Dec. 1996.

[24] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *Proceedings of the 13th IEEE Real-Time Systems Symposium (RTSS 1992)*, pages 100–109, Dec. 1992.

[25] H. Tokuda and T. Kitayama. Dynamic QoS control based on real-time threads. In *Proceedings of the Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 114–123, 1993.

[26] Y. Wang and K. Lin. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, Phoenix, AZ, Dec. 1999.

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

A. EDF with Non-constant Periods

The proof for Theorem 1 is a classical real-time scheduling theory result [15]. Although it is well-known, a similar proof is provided here for reference. This following proof is a modified version of one offered by Liu [16]. The task model here differs slightly because we assume that periods of subsequent jobs of the same task need not be constant, as long as the utilization during each period is. For instance, take task T_i with a utilization u_i: given jobs $J_{i,n}$ and $J_{i,m}$ with periods $p_{i,n}$ and $p_{i,m}$, the execution time of the respective jobs will equal $u_i p_{i,n}$ and $u_i p_{i,m}$.

In this model, at the end of the period of the nth job of $J_{i,n}$ (deadline $d_{i,n}$), the total CPU used by task T_i is

$$\begin{aligned} u_i(d_{i,1}-\phi_i)+u_i(d_{i,2}-d_{i,1})+\cdots+u_i(d_{i,n}-d_{i,n-1}) \\ = u_i(d_{i,n}-\phi_i) \end{aligned} \tag{A.1}$$

where ϕ_i is the start time of the first job.

Assume that at time $d_{i,n}$ job $J_{i,n}$ of task T_i misses a deadline. If so, then it is possible to show that $U > 1$, which contradicts the tenet that $U \le 1$, and so the deadline cannot have been missed. There are two possible cases to consider:

Case 1 In the first case, all other tasks have released their current jobs after $r_{i,n}$. In this case, the total service time required by T_i plus the service time of all jobs that were completed is:

$$X = u_i(d_{i,n}-\phi_i)+\sum_{k\neq i} u_k(d_{k,recent}-\phi_k)$$

where $d_{k,recent}$ is the last deadline of task T_k occurring before $d_{i,n}$. For all tasks, $\phi \ge 0$, so:

$$u_i d_{i,n}+\sum_{k\neq i} u_k d_{k,recent} \ge X$$

The most recent deadline of every job completed by other tasks is before $d_{i,n}$, so $d_{k,recent} \le d_{i,n}$ and:

$$u_i d_{i,n}+\sum_{k\neq i} u_k d_{i,n} = U d_{i,n} \ge X$$

A missed deadline means the service time requested exceeds the elapsed time, so $X > d_{i,n}$ and now $Ud_{i,n} > d_{i,n}$, which leads to the contradiction $U > 1$.

Case 2 In the second case, some tasks $\mathbf{T'}$ have released their current jobs before $r_{i,n}$. These tasks may have received CPU before the release of $J_{i,n}$. However, since we assume $J_{i,n}$ misses its deadline, there must exist an interval between $t' \le r_{i,n}$ and t, in which only tasks releasing jobs at or after t' with deadlines before t are executed. Consider the first release of any such job belonging to task T_k to occur at ϕ'_k. Over the interval $t-t'$, the total demand for the processor is:

$$X = u_i(d_{i,n}-r_{i,n})+\sum_{T_k\in\mathbf{T}-\mathbf{T'}} u_k(d_{k,recent}-\phi'_k)$$

Where $d_{k,recent}$ is the last deadline of task T_k occurring before t. Because $r_{i,n} \ge t'$ and $\phi'_k \ge t'$:

$$u_i(d_{i,n}-t')+\sum_{T_k\in\mathbf{T}-\mathbf{T'}} u_k(d_{k,recent}-t') \ge X$$

$d_{k,recent} \le d_{i,n}$ and:

$$u_i(d_{i,n}-t')+\sum_{T_k\in\mathbf{T}-\mathbf{T'}} u_k(d_{i,n}-t') \ge X$$

Call $U' = \sum_{T_l\in T_i,\mathbf{T}-\mathbf{T'}} u_l$. The same argument from above follows. When the deadline is missed, $t = d_{i,n}$, so $U'(t-t') \ge X$. A missed deadline means the service time requested exceeds the elapsed time, so $X > t-t'$, so we have $U'(t-t') > t-t'$ yielding the contradiction $U' > 1$.

B. Feasibility of Decreasing Deadlines

The following is the remainder of the proof of Theorem 5. The problem was set up in the description of the theorem in Section 4.2. When the deadline of $J_{i,n}$ is decreased from $d_{i,n}$ to $d'_{i,n}$ under the constraints of the theorem, there is only a subset of tasks having jobs at risk of missing deadlines after time $d'_{i,n}$, and all of these jobs were released after $r_{i,n}$; we call these tasks, starting at their first "at risk" job the set $\mathbf{T_R}$, and for all members the release of the first at risk job occurs at $\phi \ge r_{i,n}$.

At time $d_{i,n}$, the service time requested by T_i since $r_{i,n}$ is $u_i(d'_{i,n}-r_{i,n})$. The service time of all jobs completed at some time t_m after $d'_{i,n}$ is $u_i(d_{i,recent}-d'_{i,n})$, where $d_{i,recent}$ is the most recent completed deadline.

If a task T_m in $\mathbf{T_R}$ misses a deadline of job $J_{m,x}$ at time $d_{m,x}$, the total service time requested by $\mathbf{T_R}$ and T_i since $r_{i,n}$ is:

$$\begin{aligned} X =& u_m(d_{m,x}-\phi_m)+u_i(d_{i,recent}-r_{i,n})+ \\ &\sum_{T_k\in\mathbf{T_R},k\neq m} u_k(d_{k,recent}-\phi_k) \end{aligned}$$

Because for all $\mathbf{T_R}$, $\phi \ge r_{i,n}$:

$$\begin{aligned} u_m(d_{m,x}-r_{i,n})+u_i(d_{i,recent}-r_{i,n})+ \\ \sum_{T_k\in\mathbf{T_R},k\neq m} u_k(d_{k,recent}-r_{i,n}) \ge X \end{aligned}$$

All $d_{recent} \le d_{m,x}$:

$$\begin{aligned} u_m(d_{m,x}-r_{i,n})+u_i(d_{m,x}-r_{i,n}) \\ +\sum_{T_k\in\mathbf{T_R},k\neq m} u_k(d_{m,x}-r_{i,n}) \ge X \end{aligned}$$

Set $U' = \sum_{T_l\in T_i,\mathbf{T_R}} u_l$. Since $U'(d_{m,x}-r_{i,n}) \ge X$, and the requested time exceeds the elapsed time $X > d_{m,x}-r_{i,n}$, we have $U' > 1$, a contradiction.