Developing a Complete Integrated Real-Time System

Scott A. Brandt, Scott Banachowski, Caixue Lin, and Joel Wu
Computer Science Department
University of California, Santa Cruz
{sbrandt, sbanacho, lcx, jwu}@cs.ucsc.edu

Abstract

Modern systems are frequently called upon to support mixes of applications with different types of timeliness requirements. Current solutions for supporting such mixes are ad hoc *and do not guarantee the requirements of all types of processes. We discuss the need for better systems support for such mixes and present partial solutions toward the development of such systems. These include an integrated real-time scheduler that focuses on best-effort performance, a slack scheduler designed to improve the performance of soft real-time processes, and an integrated soft real-time disk bandwidth manager.*

1. Introduction

Application timeliness requirements vary from hard real-time to best-effort with many flavors in between, including soft real-time, firm real-time, rate-based, *etc.* Many application scenarios ranging from desktop multimedia to large distributed real-time systems require the ability to simultaneously support these different types of requirements, yet existing systems provide little direct support for more than one type. We argue that (1) all systems should support the full range of possible timeliness requirements, and (2) this can only be accomplished with resource management algorithms that are hard real-time at their core. Toward this goal we present a CPU scheduler appropriate for use as the only scheduler in such a system, a slack scheduling algorithm designed to optimally address the needs of *other* tasks, and a disk bandwidth manager capable of handling a variety of timeliness requirements.

Our CPU scheduler, HodgePodge, supports integrated hard real-time, soft real-time, and best-effort scheduling. Unlike many previous real-time schedulers, HodgePodge focuses primarily on the performance of best-effort applications running concurrently with hard or soft real-time applications. Our results demonstrate that it is feasible and practical to combine the processing of these different types of applications without degrading the performance of any of them. We believe this is a necessary first step towards the development of the type of integrated real-time systems that we envision.

In integrated systems with many different types of processes, hard (and occasionally soft) real-time tasks frequently over-reserve resources in order to ensure that they meet deadlines. Whenever these processes use less resources than they have reserved, dynamic slack is generated. The performance of soft and non-real-time processes heavily depends upon the availability and efficient distribution of this slack. Integrated real-time systems therefore demand the development of algorithms that distribute slack so as to best support the goals of the other processes. Our slack scheduling algorithm, SLASH, does this by distributing slack as soon as it is available, to the process with the earliest deadline, and allows jobs of a task to borrow resources from future jobs of the same task. This provides the resources to the most critical job as early as possible while guaranteeing the correctness of the schedule.

Real-time systems research often focuses on CPU resource management. Other resources, and especially I/O devices, are managed minimally, if at all. Solutions that do address these resources tend to focus on a single type of timing requirements: hard, soft, or non-real-time. However, integrated systems that will support a variety of timing requirements must manage resources other than the CPU, and must do so in ways that directly support different timing requirements. Our Hierarchical Disk Sharing (HDS) algorithm begins to address this problem in the domain of disk I/O. Based on a technique developed for networking, HDS partitions the disk bandwidth and allows processes to reserve fixed or relative shares of the available bandwidth. HDS addresses some of the unique issues that arise in managing disk I/O, including the non-uniform (and only partially deterministic) service times associated with disk requests.

So far we have developed CPU and disk allocation algorithms for integrated systems, but are just beginning to combine them into a single system. The following sections discuss HodgePodge, SLASH, and HDS in more detail.

2. HodgePodge

General-purpose operating systems are designed to serve a wide variety of applications. Yet because these systems use best-effort CPU scheduling, the growing body of applications that have time constraints remain unsupported. We envision a merging of real-time scheduling techniques with general-purpose systems. The advantage comes from two perspectives: (1) general-purpose systems need not treat real-time applications (*e.g.* multimedia) in an *ad hoc* fashion, and (2) real-time systems need not treat non-real-time applications in an *ad hoc* fashion.

The definition of "general-purpose" has grown to include the kind of tasks traditionally thought of as real-time. Examples include: games, video players and encoders, home studio software for multi-track audio generation/recording/sequencing, voice recognition, and hardware emulation tasks such as soft-modems. Many other applications benefit from the fine-grained partitioning and isolation of resources real-time schedulers enforce, such as virtual sharing of processors by web servers, or reservation of CPU for highly compute-intensive data consumers such as scientific applications or search engines. In our experience these applications are treated in an *ad hoc* fashion. Typically, because they tolerate some degree of missed deadlines, the tasks are scheduled in the default best-effort manner and mostly meet deadlines because the CPU resource is over-provisioned (or by luck when processor load is high). Other options include playing with the *nice* priority, which is not all that predictable or robust, or overriding the scheduler by choosing a high static priority (a technique sanctioned by a POSIX standard [12]). The latter approach is unsafe when the task is not designed to be cooperative or is buggy [17]. And this approach is not scalable in the case where multiple tasks need real-time support. Our approach is to provide an integrated real-time scheduler for all tasks.

Conversely, many real-time systems share some general-purpose requirements. Although the primary functions of flight, defense, and manufacturing control systems are real-time, they commonly include many non-critical, yet non-trivial, tasks that are best served using traditional time-sharing techniques. In our experience these applications are treated in an *ad hoc* fashion. In a real-time system, non-real-time tasks are often deemed unimportant, and left to background processing, when they instead prefer time-share disciplines. More sophisticated approaches use hierarchies of schedulers [6, 10], allowing co-existence of multiple schedulers for different kinds of tasks. However, an arbitrary scheduling hierarchy may become (needlessly) complex, and even in simple hierarchies the effect of stacking schedulers must be well-understood to ensure meeting constraints [19]. Our approach is to provide a simple mechanism for time-sharing the non-reserved resources of a real-time system.

The goal of the HodgePodge (Holy-Grail, Pipe-Dream) CPU scheduler is to support a veritable hodgepodge of processing or timeliness constraints. HodgePodge uses a real-time scheduler, EDF [15], for all tasks, whether they have time constraints or not. Time-sharing is provided by a resource allocation layer that uses the reservation capability of the real-time scheduler [3]. To make such a system desirable for general-purpose, it should in all appearances mimic a time-share scheduler except when called upon to run a real-time task. Our previous experience showed that, using aperiodic bandwidth servers for non-real-time tasks, we may get behavior similar to time-share algorithms in terms of responsiveness and overhead [2]. The novelty is to provide time-share-like service by adapting each application's aperiodic server parameters during run-time based on behavior.

2.1. The Best-effort Bandwidth Server

General-purpose systems behave unpredictably, because it is not known *a priori* which tasks will run, or when. In contrast, a periodic real-time task is predictable: it is a sequence of *jobs*, where each job begins at the start of a period, and completes at or before the end of the period. Non-real-time tasks may not resemble periodic tasks at all—however, during execution tasks can still be modeled as a sequence of jobs. These jobs may not begin or end in periodic (or even predictable) intervals, so it is therefore natural to treat these tasks as *aperiodic*.

An *aperiodic server* is an algorithm that assigns periodic deadlines to tasks that are not necessarily periodic, or which have no deadlines. Modern processors have the processing headroom for dynamic scheduling, and our previous work shows that the overhead of using an aperiodic server for each task is akin to existing time-share schedulers. It follows that it is practical to use a real-time scheduler as the core of a general-purpose scheduler, with aperiodic servers for non-real-time tasks. The advantage is providing real-time scheduling as a native feature, without resorting to an *ad hoc* addition or combination of schedulers.

The Best-effort Bandwidth Server (BEBS) is an aperiodic server that addresses the two main goals of time-share scheduling: fairness and better responsiveness for interactive tasks. It achieves fairness by adjusting the reservations of tasks equally, and allocating the reclaimed slack fairly. Slack is any CPU that is unreserved, and any reserved CPU that is unused. IRIS [16] is a server designed to reclaim slack fairly, and BEBS is similar to IRIS, with differences noted in our previous report [2]. To meet the interactive goal, the server adjusts its period according to the run-time behavior of the task: interactive servers have shorter periods for better responsiveness, while compute-bound servers have longer periods (which incur less scheduling overhead). Each server is assigned a utilization equal to a fair-share allowance of CPU bandwidth.

2.2. A Brief Comparison

To illustrate the difference in operation between a traditional time-share scheduler and HodgePodge we describe a simple scenario. Imagine N tasks executing, all using equal amount of CPU. In a time-share system based on multilevel feedback queues, such tasks will reside in the same priority queue and receive service in round-robin quanta of length q. In the worst-case the longest wait for service is $(N-1)q$, and the task will receive at least q amount of service in $q \times N$ amount of time.

In HodgePodge, each task is assigned a reservation by the resource allocation algorithm. In the above workload, a reservation equal to $(p := qN, u := 1/N)$ will, in the worst-case, give q amount of service in $q \times N$ amount of time, the same as the time-share system. In HodgePodge, once set, this reservation will be guaranteed, independent of other activities in the system. This is an advantage over traditional time-share scheduling.

Now consider what happens if tasks differ in interactivity. In the time-share system, they will be served from different queues, with higher priority tasks preempting lower priority tasks. It becomes difficult to predict exactly when a task will receive a full quantum q of service, because the service of its queue may be preempted by other tasks for any unknown number of durations.

In HodgePodge, the performance of tasks is adjusted by controlling server reservations at run-time, based on the past task activity. Interactive tasks do not receive higher priority, but instead receive reservations consistent with their past execution, for example a reduced utilization but increased periodic rate (scaled in accordance with the level of interactivity and other factors such as *nice* setting). Interactive tasks likely preempt less-interactive tasks because their periodic deadlines are likely earlier when active; in the average case they remain responsive. However, each task is still guaranteed a reservation, so all receive a predictable level of service in accordance with the time-share goals. We have found that while running a mix of real and non-real-time applications, the performance of time-sharing in this approach is significantly better than assigning real-time tasks higher priorities, while at the same time there are no violation of real-time constraints [2].

The reservation policies can be tuned to mimic the expected performance from Linux or any other time-share scheduler. An enhancement to the algorithm also tries to auto-detect multimedia and other periodic soft real-time applications while they execute, and make reservations consistent with their inferred requirements (such as by the measured period of frame synchronizations). In this way, the system better supports legacy periodic applications.

2.3. HodgePodge Implementation

In order to build a HodgePodge prototype and test and use BEBS in a general-purpose environment, we implemented EDF in Linux [14]. We replaced the Linux scheduler while leaving as much of the existing infrastructure intact. This is not the approach we'd take if implementing from scratch; the existing structure of Linux definitely impacts our design and performance. For example, some process accounting occurs during a periodic timer interrupt that is not necessary for EDF. However, disabling this interrupt also disables mechanisms for timeouts and synchronization used by many device drivers and applications. Also, removing the interrupt would require significant changes to some of the process accounting. Rather than disable and re-implement portions of the kernel, we decided to leave them intact, and when applicable leverage them for our purpose.

Linux uses a 1000 Hz periodic timer to drive many operations (a.k.a the *tick* timer). We leverage this interrupt to schedule the release of jobs. All processes in the system are treated as sequences of jobs that begin at periodic intervals. Each job has a processing budget (or quantum) equal to $u \times p$ (determined by the reservation tuple (p, u)). When its budget is used, a job suspends until the start of the next period. By using the tick timer for scheduling these events, we do not need to support arbitrary release times or periods, and reduce the number of interrupts and task preemptions.

Because all task jobs must be released on 1 ms. intervals, the minimum period of a reservation is bounded to an integral number of milliseconds. We call the occurrence of 1000 Hz clock interrupt a *major tick*. Currently there are no sub-millisecond periods. For this system we expect most real-time workloads to involve multimedia rates of at most 50 Hz, so this is currently sufficient for our purpose.

The EDF scheduler enforce reservations by using a timer interrupt to prevent tasks from overrunning their budget (similar to the R-EDF implementation [23]). Periods are scheduled at relatively coarse-grained times, but task budgets may be a fraction of a *tick* interval, requiring a higher resolution timer to trigger a reschedule when a job's budget expires (we use the Pentium-class APIC timer). Thus a one-shot timer only needs to be programmed if an expiration occurs before the next *major tick*. This implementation resembles firm timers [9], because the actual overhead of setting up hardware is avoided when a task is preempted before its budget expires. Also, only a single one-shot timer must be maintained at any time, so we need not maintain lists of timer events. The one-shot timer is programmed to the nearest microsecond, and we call these clock intervals *minor ticks* (although unlike *major ticks*, there are not periodic interrupts at every *minor tick*).

Every task is assigned a utilization u, which is its allocated fraction of CPU, and dictates the maximum amount of time (budget b) it may execute per period p ($b = up$). The granularity of the one-shot timer limits the budget we may assign to at most 1 μs.[1] To simplify reservations, the system requires the utilization be set in an increment of 0.1% (1 μs per ms period). Since a task must have some utilization, this limits the number of servers we may admit to at most 1000 (however a server may service multiple tasks, so this is not a limitation on task number).

EDF selects the task with the earliest deadline, requiring an $O(n)$ search of n runnable tasks. Previous versions of Linux (< 2.5) also required $O(n)$ selection, but the newer versions bound the search to a fixed number (locating the first non-empty priority queue). Our EDF algorithm is not quite as scalable as Linux's new constant-time algorithm, however it is better than the previous version of Linux.

We shift overhead from the selection code to the queue insertion by keeping the run queue sorted in deadline order. On average, inserting into an already sorted list is better—we found that for random task sets, sorted inserts averaged less than 3 times fewer operations than searching the unsorted list. This gives us less overhead than the previous generation Linux. We are looking into further optimizations in queuing structures to reduce overload for large sets of servers. [2]

2.4. Future Directions

In order to build the HodgePodge prototype and test and use BEBS in a general-purpose environment, we implemented EDF in Linux [14] by replacing the kernel's scheduler. Changing Linux's CPU scheduler alone does not make it real-time, but better equips it to handle workloads with time constraints. An existing problem is that CPU used by the kernel is charged to the currently running task, even if the work is on behalf of another. Using a technique such as Augmented CPU reservations [18] may track the time "stolen" from tasks by OS operations, making our CPU allocations better match reservations.

BEBS could be incorporated into a hard real-time environment by adapting techniques used by DROPS [11], which allow time-share and real-time applications to coexist on a real-time micro-kernel, or RTLinux [22], which runs Linux as a low-priority task on a real-time executive. Both approaches treat the time-share portion of the system as a single user-program; our approach dictates that the time-share portion be treated as a set of servers, with each a corresponding user-program. This is an area for future research.

3. SLASH

The increasing demand for more powerful computing platforms and applications requires modern operating systems capable of simultaneously supporting applications with a variety of different time constraints. The hierarchical HLS scheduler [19], and the flat integrated RBED [3] and closely related HodgePodge schedulers are examples. Such systems simultaneously support (1) critical hard real-time applications such as external signal sampling and processing, (2) non-critical soft real-time applications such as desktop multimedia, and (3) best-effort applications such as compilers, word processors, *etc*. Hard real-time applications make worst-case resource reservations to guarantee their constraints; soft real-time applications may reserve less than worst-case to achieve good average-case performance; and best-effort applications generally make no reservations beyond what is necessary to avoid starvation. Any variance in execution times below what has been reserved leads to dynamic slack—reserved but unused resources. Efficient distribution of this slack to processes whose current needs exceed their reservation can significantly improve the performance of both soft real-time and best-effort applications.

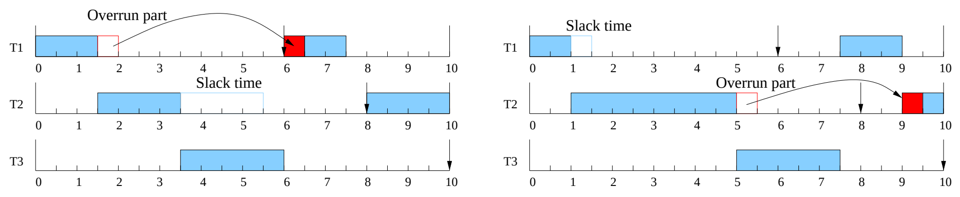
SLASH is a slack scheduling mechanism system specifically designed to improve the performance of soft real-time applications while guaranteeing the worst-case reservations of hard real-time processes. Our evaluation shows that SLASH reduces the number of missed deadlines and decreases the average tardiness of late deadline for soft real-time applications when both hard real-time and soft real-time applications coexist. In our experiments, SLASH always misses fewer deadlines than CBS [1], BEBS, and CASH [5], reducing missed deadlines by 70%, 70% and 10% (respectively) in the best scenario we observed.

3.1. SLASH Design and Implementation

SLASH is implemented in RBED [3], which uses an integrated scheduler for hard, soft, and non-real-time processes. The low-level scheduler is earliest deadline first (EDF). Processes use the scheduler by associating their tasks with a rate-based server, conceptually similar to CBS and other bandwidth servers. A server is characterized by a reservation tuple (B_s, P_s), where B_s is the execution budget and P_s is the period (both in units of time). The server utilization is $U_s = \frac{B_s}{P_s}$. A deadline occurs at the end of the each period. Each hard or soft real-time task is associated with

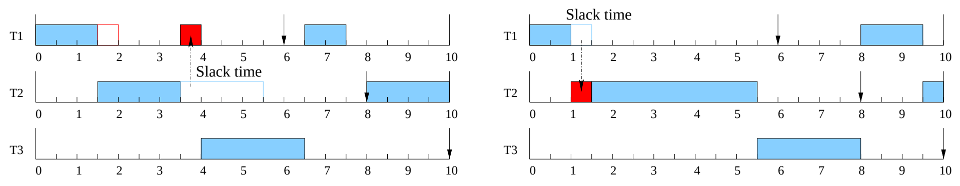
[1]Budgets this small are impractical for Linux. We measured the average context switch time on a 2.4 Hz P4 to be about 5 μs, so a task with 1 μs budget will consume more time in context switch, not to mention cache warming, than its budget allows.

[2]For handling a larger number of tasks we may consider using a sorted heap with $lg(n)$ insertion and deletion.



(a) case 1 (b) case 2

Figure 1. Drawbacks of idle-time slack management



(a) case 1 (b) case 2

Figure 2. SLASH solves both problems

its own server. Periodic and aperiodic best-effort tasks are scheduled as soft real-time tasks. All other best-effort tasks are served by one server in first come, first serve order.

SLASH combines our **SLA**ck **D**onation (SLAD) algorithm [13] with the CASH slack management algorithm [5]. In SLAD, when a task completes, it *immediately* donates any remaining budget to the task whose server has the earliest deadline. This server is the most critical, the most likely to be near the completion of its current job, and the least likely to benefit from any later donations of slack.

Figure 1 shows two problems that can occur when slack is only made available to tasks when all other tasks are idle, a technique common among other algorithms. The problems are that slack cannot be used to prevent a deadline miss caused by either (1) a past overrun or (2) a future overrun. The examples show three soft real-time tasks, $T1$, $T2$, and $T3$, with the following respective reservation configurations: $(B_1 = 1.5, P_1 = 6)$, $(B_2 = 4, P_2 = 8)$ and $(B_3 = 2.5, P_3 = 10)$; the CPU is 100% utilized. Each task has an actual deadline coinciding with its server deadline, and may overrun its reserved budget. In Figure 1(a), the first job of $T1$ has an actual execution time of 2, exceeding its reservation by 0.5, and the first job of $T2$ has an actual execution time of 2, 2 less than its reserved budget. With idle-time slack management, the overrun portion of $T1$ does not resume execution until time 6, missing its deadline. In Figure 1(b), the first job of $T1$ has an actual execution time of 1, 0.5 less than its reservation; the first job of $T2$ overruns by 0.5. Again with idle-time slack management, the slack is "pushed back" and unused until a later idle point, resulting in $T2$ missing its deadline. By donating slack to other processes as soon as it is available, SLASH solves both of these problems. The resulting schedules are shown in Figure 2. In both cases no task misses its deadline, despite the overruns.

In CASH [5], when a server becomes idle, any remaining budget is recorded in a queue. When a server runs, it first consumes all queued budgets with deadlines $\leq$ its own. When a server consumes its budget, it is recharged and its deadline extended by one period, allowing it to borrow against its future budget to complete the current job. This has some benefits, allowing current jobs that need more CPU to safely borrow from future jobs of the same task, which may need less CPU (or may borrow from still more future jobs). Unfortunately, this borrowing may prevent slack from getting to the tasks that need it most—when a task overruns its budget, its server deadline will be postponed before the task completes its current job, reducing its EDF priority and making it less likely to receive slack.

SLASH addresses this problem by combining the SLAD EDF-based slack donation mechanism with the CASH greedy budget replenishment mechanism, donating slack to the tasks that need it the most and executing overrun tasks as early as possible (so that they improve their chance of meeting deadlines) by not forcing servers to remain inactive when their budget is consumed.

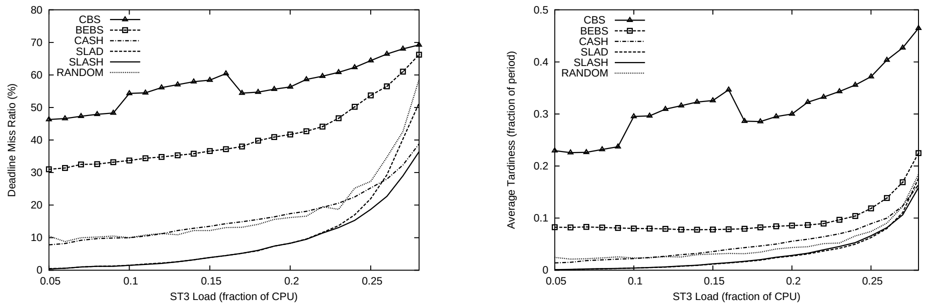
3.2. The SLASH algorithm

SLASH uses an earliest virtual deadline first (EVDF) for slack scheduling decisions. The virtual deadline of a server is calculated as follows:

$$vd_{s,k} = d_{s,k} - \left\lfloor \frac{d_{s,k} - t}{P_s} \right\rfloor P_s$$

where t is current time (note that the condition $d_{s,k} > t$ always holds). Like CASH, SLASH has no *expired* status. When a SLASH server consumes its budget, the budget is recharged, its deadline is advanced by one period, and its state is reset to *waiting*. The algorithm is as follows:

1. At the beginning of each period, the current budget of a server c_s is set to its reservation budget B_s, its



(a) Deadline Miss Ratio as a function of load (b) Average Tardiness as a function of load

Figure 3. Load effect on performance (one soft real-time task, $p = 310$)

dynamic deadline $d_{s,k}$ is set to $d_{s,k-1} + P_s$, and its state is set to *waiting*.

2. The *waiting* server with the earliest deadline becomes *running*.
3. A *running* server executes its pending task on the CPU until it has finished its task or consumed its budget, and decreases its budget c_s by the actual amount of CPU consumed. If it has no pending task, it donates any remaining budget to:
   (a) the task of the *waiting* server with the earliest virtual deadline; or, if none exist, to
   (b) the idle task
4. When c_s of a *running* server equals zero, the server is recharged with full budget $c_s = B_s$, its deadline is incremented $d_{s,k} = d_{s,k} + P_s$, and its state is reset to *waiting* (or remains *running* if it still has the earliest deadline).
5. When a *running* server is preempted, its state is set to *waiting*.

Step 3 implies that if servers always have their own associated tasks to execute, then no slack scheduling occurs. If there is slack available, it is immediately donated to other pending tasks whose servers have the highest priority determined by EDF. It is possible for other slack scheduling choices besides EDF to perform better for certain applications or in certain easily contrived circumstances. Nevertheless, SLASH is simple, straightforward, and effective in practice.

3.3. SLASH Performance

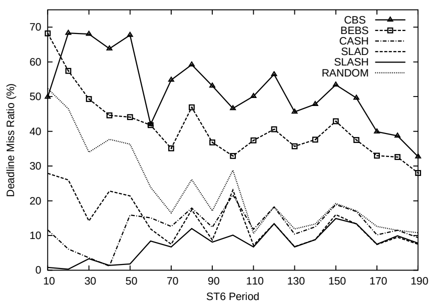
We compare the performance of SLAD and SLASH to CBS, BEBS, CASH, and a RANDOM algorithm (which provides aggressive slack donation like SLAD but, instead of using EDF, assigns slack to a random task). All hard real-time tasks meet their deadlines. Our metrics of soft real-time performance are deadline miss ratio (DMR), and average tardiness (ATD).

The first experiment examines soft real-time performance as a function of system load. The workload consists of two periodic hard real-time tasks and one periodic soft real-time task. In this experiment HRT1 has constant execution time equal to its server budget, HRT2 has normally distributed execution times with its server budget equal to the worst-case, and SRT3 has normally distributed execution times with its server budget set to their average. SRT3 will often overrun its budget but should meet most of its deadlines by taking advantage of the slack from HRT2.

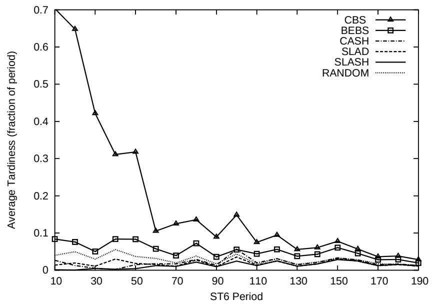
Figures 3 and 4 show SRT3's deadline misses and tardiness, using the different algorithms, as a function of utilization between 5% and 29% (in all cases, the total average sum of server utilization is 100%). RANDOM, SLAD, and SLASH outperform CBS and BEBS, demonstrating the benefit of donating slack at the earliest possible time. SLAD and SLASH outperform RANDOM, demonstrating the additional benefit of giving the slack to the process with the earliest deadline. SLAD and CASH outperform each other in different circumstances. Finally, SLASH outperforms both SLAD and CASH, demonstrating the effectiveness of combining SLAD slack donation with CASH budget replenishment.

The second experiment shows soft real-time performance as a function of server (and task) period. The workload consists of five periodic hard real-time tasks and one periodic soft real-time task. Every hard real-time task has execution times fitting a normal distribution and a server budget set to their worst-case execution time. SRT6 has normally distributed execution times with its server budget set to the average. Each hard real-time task reserves 10% of the CPU and SRT6 reserves the remaining 50%.

Figure 4 shows SRT6's performance as a function of period ranging from 10 to 190. We see results similar to the previous subsection: (1) RANDOM, CASH, SLAD, and SLASH outperform CBS and BEBS, (2) SLAD and CASH outperform each other in different circumstances, and (3) SLASH is always the best. Interestingly, as we increase the number of soft real-time tasks (and decrease their target utilization and reservations accordingly), we find that



(a) Deadline Miss Ratio as a function of period



(b) Average Tardiness as a function of period

Figure 4. Period effect on performance (one soft real-time task, $u = 50\%$)

the performance of SLASH improves (not shown).

In our experiments, SLASH always outperforms CBS, BEBS, and CASH in this regard, reducing missed deadlines by 70%, 70% and 10% (respectively) in the best scenario we observed. Although designed for our integrated real-time system, RBED [3], SLASH should work equally well with any deadline-aware scheduler.

4. HDS

Systems that use or serve multimedia data require timely access to data on hard drives. To ensure adequate performance in an integrated real-time environment, users must either prevent overload of disk resources, not generally feasible in a general-purpose environment, or use real-time algorithms that rely on intricate knowledge of disk internals to meet deadline requirements. Hierarchical Disk Sharing (HDS) allows disks to be (nearly) fully utilized while sustaining bandwidth reservations, without requiring detailed knowledge of the drive internals. Derived from hierarchical link sharing for networks [8], HDS uses a hierarchy of token bucket filters to isolate disk access among clients and groups of clients, and to allow for reclaiming of unused bandwidth, capabilities that are absent in current commodity operating systems and which are necessary to support time constraints in an integrated system.

One of our main design goals is that the reservation mechanism be independent of high-level features like file-systems, and low-level features like disk schedulers, so that it can be employed across many systems, including storage network devices. Therefore we chose to implement our prototype of HDS in the block device layer of the Linux kernel. We discuss the design of HDS and present our Linux implementation, demonstrating both the effectiveness (and limitations) of this approach.

4.1. HDS Design and Implementation

Traditional disk access is best-effort, with no timing guarantees. Acceptable performance is achieved when the disk is not overloaded. When demand for disk bandwidth exceeds the supply, all applications may experience performance degradation, including those with time constraints. Our approach to this problem is to provide a mechanism that allows reservations of disk bandwidth, graceful degradation under heavy load, and reclaiming of unused reservations. A hierarchical structure for resource sharing provides a basis for meeting all of these goals.

Specifying a disk reservation by bandwidth is intuitive, but disk bandwidth is not constant; service times vary depending upon the initial position of the read-write head, the position of the requested data on the disk, the low-level disk scheduling algorithm, *etc.* Translating bandwidth requirements into low-level disk operations is a complicated task [7]. Alternatively, specifying a disk reservation by a reserved time-slice (instead of bandwidth) may result in different amounts of data being retrieved per time-slice.

Existing reservation-capable schedulers contend with this issue. The Cello [20] scheduler presents two methods of accounting, either by size or time. Some schedulers allow reservations in terms of number of requests [4, 21]. However, requests may also vary in size and service time. To fulfill QoS goals, the system must provide performance in line with the users' expectations, regardless of the amount of work that the disk is actually doing on behalf of different users. We expect users to perceive the quality of service for disk by the bandwidth (data rate) that it can provide. Therefore HDS accounts for disk usage in terms of bandwidth and does its accounting based on the actual amount of data transferred.

In HDS, a disk's bandwidth is divided between applications in a hierarchical tree structure; an example is pictured in Figure 5. Each leaf node represents a point of control for accessing the disk, and is associated with a Linux process. When a process first attempts to access the disk, a leaf node is created and added to the tree. When it quits, its node is removed. Non-leaf nodes are called *classes*, and represent a group of clients. The children of a class node may be leaf nodes or other class nodes.

Figure 5 demonstrates using classes to isolate best-effort

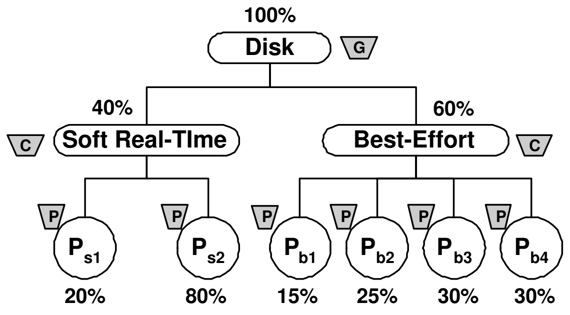


Figure 5. HDS allows arbitrary mix of shares controlled by Global bucket, Class buckets, and Process buckets

and real-time processing, an approach useful for multimedia servers where the requests with time constraints should be isolated from other traffic. Our system has an interface for constructing the desired class hierarchy, including dynamically adding and deleting classes.

Each node x has an associated reservation r_x, determined by the reservations of nodes above it in the tree structure. There are two modes for a node to specify its reservation: either an *absolute* fraction f_x, or a *relative* fraction, based on a weight w_x, of the parent node's reservation. The root node has an absolute fraction of 1. If a node x has an absolute reservation f_x (between 0 and 1), its reservation is this fraction of its parent's reservation. For example, because the root node has $r_0 = 1$, a child of the root with $f_x = 0.4$ will have a reservation of $r_x = f_x r_0 = 0.4$. The sum of absolute fractions among any node's children may not exceed 1. A class's bandwidth that is not used by absolute reservations is shared by its other children in proportion to their relative weights.

By default, clients are added to a parent node with equal weight to promote fair sharing. When a client needs a higher level of service than others, it may do so by either increasing its weight or requesting an absolute fraction of its parent's bandwidth. If the nodes on the path from a client to the root all have absolute reservations, then the client effectively reserves a static fraction of the total disk bandwidth; if any node in this path has a relative reservation, the node's reservation may vary when other nodes join or leave the structure. HDS allows administrators to set permissions for adding classes or nodes, changing reservations, admission control, *etc.*

Disk bandwidth may be controlled at different points in the I/O stack. HDS resides at the block-device layer, between the file system and disk scheduler. The regulation of disk bandwidth in HDS is implemented using token bucket filters. In order to make disk requests, a client must possess tokens. In HDS, each token represents 1 KB of data, meaning a request for 16 KB of data requires 16 tokens. Each node x in the hierarchy has an associated bucket, which may hold up to N_x tokens. When a client request is serviced, tokens are removed from its bucket. Tokens are replenished at a rate corresponding to the client's reservation. If the root token rate is T_0, then its child node x with reservation r_x will replenish tokens at rate $T_x = r_x T_0$. The root token rate represents the entire available bandwidth of a disk.

Although every node has a token bucket, only leaf nodes make requests. The token buckets of non-root nodes facilitate sharing of bandwidth. In addition to its own tokens, a node may use tokens from its parent (which in turn may use those of its parent). The effect is that unused bandwidth is shared first among nodes of the same class, then among parent class, and eventually, globally.

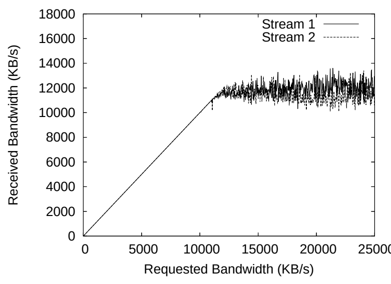
When a node drains tokens from a bucket, it also drains those from buckets in the path up to and including the root. The result is that when a class's children make disk requests, the class's tokens will be drained as well. If some of the children are not fully using their reservation, the parent will have surplus tokens. These tokens are available to other children when their own supply runs out, so that a node that has exceeded its reservation may still be able to proceed. Bandwidth isolation is preserved not only between leaf nodes, but at the class level and, in fact, at every level of the hierarchy.

4.2. HDS Performance

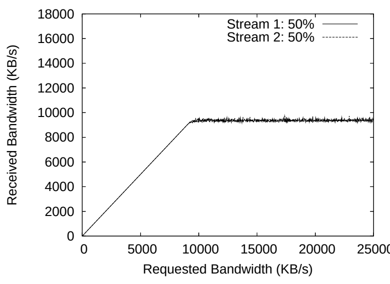
We ran several experiments to demonstrate the ability of HDS to shape disk traffic, using synthetic applications to generate disk workload. We focus mostly on read workloads both because multimedia is typically read-intensive and because write performance is often aided by buffering. Our test system is a 1.5 GHz P4 with 512MB of RAM. The disk is a Seagate ST340810A IDE drive formatted with the ext2 file system.

Figure 6 shows a situation where disk bandwidth has become saturated by two processes reading from the disk. The x-axis shows the requested bandwidth and the y-axis shows the measured received bandwidth. Figure 6(a) shows the result on unmodified Linux. Both processes receive their desired bandwidth until the disk becomes saturated with requests. There is no isolation, so at that point actual throughput is unpredictable and varies considerably.

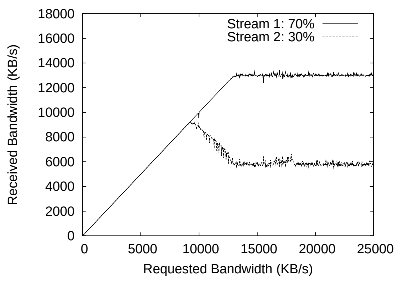
HDS provides reservation and isolation of bandwidth. Figure 6(b) shows the same experiment with HDS, where each task reserves equal relative weight. At saturation the bandwidth divides evenly between the streams and achieved throughput is very stable. This fair-sharing comes at the expense of slightly lower overall disk throughput because we limit the number of requests. Figure 6(c) demon-



(a) Normal Linux system behavior

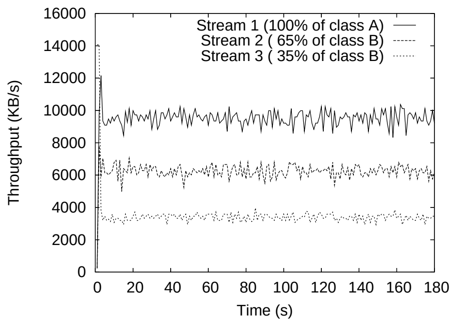


(b) With HDS sharing of 50%-50%

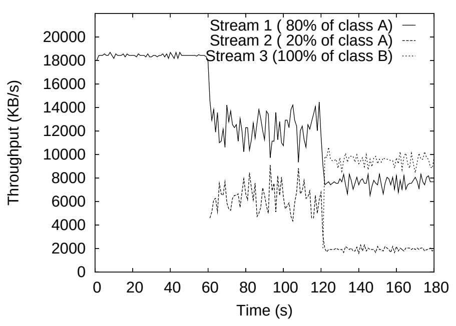


(c) With HDS sharing of 70%-30%

Figure 6. The effect of overload on throughput



(a) Isolation of bandwidth (Class A and B reserve 50% each. All streams are greedy)



(b) Using unassigned bandwidth (Class A and B reserve 50% each. All streams are greedy)

Figure 7. HDS isolation and slack reclamation

strates the effect of allocating 70% of the disk to stream 1 and 30% to stream 2.

The next experiment shows the ability of HDS to provide hierarchical resource sharing. We created two classes, A and B, each reserving 50% of the disk. Stream 1 belongs to Class A, so it reserves 100% of the class reservation. Streams 2 and 3 belong to Class B, and reserve 65% and 35% of Class B's reservation, respectively. Figure 7(a) shows that all three streams receive bandwidth corresponding to their allocation, with no interference from each other.

Excess bandwidth may be available when a process needs more than its reserved share. Figure 7(b) shows this scenario. In this experiment, there are two classes and three streams. Class A and B each reserve 50%. Stream 1 and 2 belong to Class A and reserve 80% and 20% of its bandwidth. Stream 3 belongs to class B, so receives 100% of its bandwidth. At the beginning, only Stream 1 is active. Although its total share is only a fraction of the total bandwidth (its share is 40%), because no other tasks are active it receives the total disk bandwidth. At time 60 Stream 2 becomes active. There is still excess bandwidth because Class A's share is only 50%. The excess bandwidth is distributed to Streams 1 and 2. From time 60 to time 120, they receive their fair-share plus the excess bandwidth. Excess bandwidth is allocated on first-come first-serve basis, accounting for the observed variation in actual rate (this variation is a topic for future investigation). At time 120 Stream 3 begins and, now fully loaded, the nominal reservations are enforced.

5. Conclusion

We are developing flexible integrated real-time solutions based on real-time scheduling algorithms. This is a first step in providing better real-time support in general-purpose systems, better general-purpose support in real-time systems, and, ultimately, a general framework to fully integrate applications of different types of processing constraints.

The longer-term goals of this project include combining these solutions in a single system, and developing complete solutions for other resources including, network I/O, memory, cache, and others. A key challenge will be the development of a framework that supports the combined management of all of the system resources so that, for example, failure to meet a soft deadline in one resource will not negate the benefit of meeting the same deadline with another resource. Our ultimate goal is the complete merging of real-time scheduling techniques with general-purpose systems, supporting a range of timing constraints from hard real-time to background best-effort batch processing.

Our research is funded by a DOE HPCS Fellowship and the Intel Corporation.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, pages 4–13, Dec. 1998.

[2] S. Banachowski, T. Bisson, and S. A. Brandt. Integrating best-effort scheduling into a real-time system. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004)*, Dec. 2004.

[3] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

[4] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, June 1999.

[5] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings of the 21th IEEE Real-Time Systems Symposium (RTSS 2000)*, pages 295–304, Dec. 2000.

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

[7] S. Childs. Portable and adaptive specification of disk bandwidth quality of service. In *Proceedings of the 9th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 1999.

[8] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.

[9] 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.

[10] 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.

[11] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the Fifth Parallel and Real-time Systems (PART98)*, 1999.

[12] 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.

[13] C. Lin and S. A. Brandt. Efficient soft real-time processing in an integrated system. In *Work in Progress Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS WIP 2004)*, Lisbon, Portugal, Dec. 2004.

[14] The Linux kernel archives. http://www.kernel.org, Jan. 2004. A web site with the latest Linux kernel and information.

[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] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A new reclaiming algorithm for server-based real-time systems. In *10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS04)*, May 2004.

[17] 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.

[18] J. Regehr and J. A. Stankovic. Augmented CPU reservations: Towards predictable execution on general-purpose operating systems. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS01)*, pages 141–148, May 2001.

[19] 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.

[20] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55. ACM Press, 1998.

[21] R. Wijayaratne and A. L. Reddy. Integrated QOS management for disk I/O. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 487–492, June 1999.

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

[23] W. Yuan, K. Nahrstedt, and K. Kim. R-EDF: A reservation-based EDF scheduling algorithm for multiple multimedia task classes. In *Proceedings of the Real-Time Technology and Applications Symposium (RTAS01)*, May 2001.