BERT: A Scheduler for Best Effort and Realtime Tasks

Andy Bavier and Larry Peterson
Department of Computer Science
Princeton University

David Mosberger
Hewlett-Packard Laboratory

Abstract

We describe a new scheduler, called BERT, that runs both best effort and realtime tasks on a multimedia workstation. BERT exploits two innovations. First, it extends the proportional share scheduling algorithm to allow one task to dynamically *steal* cycles from another. This mechanism allows BERT to address a main drawback of scheduling MPEG video decoding using proportional share: the difficulty of making the right reservation. Second, BERT combines the stealing mechanism with task importance to maintain QoS guarantees in overload. In BERT, important realtime tasks can steal to make deadlines and important best effort tasks can be made immune from stealing. The result is a graceful degradation of performance under heavy load. We have implemented BERT in the Scout operating system and we present our experiences using it to schedule multimedia workloads.

1 Introduction

One of the challenges of multimedia workstations is that they must support workloads with diverse CPU requirements, ranging from compute-intensive applications capable of consuming all available CPU cycles, to I/O-bound jobs that need enough cycles to keep a bottleneck I/O device fully utilized, to realtime tasks that must be allocated cycles in a timely enough fashion to meet their deadlines. This paper describes a CPU scheduler, called BERT, that addresses this challenge.

BERT is designed to support a mix of *best effort* tasks (e.g., compiles, file transfers, web browsers) and *realtime* tasks (e.g., video, audio, interactive games); BERT is an acronym for **B**est **E**ffort and **R**eal**T**ime. The tasks running on the system are not known *a priori*, but instead enter and leave the system dynamically and unpredictably. This makes off-line scheduling algorithms inappropriate.

The best effort tasks can be either compute- or I/O-bound, and they include bursty work that is common to human/computer interaction. The system would like to provide a simple quality of service to these tasks: give them some fair share of cycles.

The realtime tasks are exemplified by multimedia applications like decoding MPEG compressed video. Such tasks need to complete each piece of work by a pre-defined *deadline* (e.g., decode a video frame once every 33ms). Since the perceived performance of a multimedia task is related to how many deadlines it makes, we would like the system to provide a quality of service that enables such a task to meet its deadlines. This turns out to be tricky, however, since multimedia tasks often require varying rates of service to meet all of their deadlines. For example, the cycles needed to decode back-to-back video frames can differ by a factor of five or more. The long-term load that a multimedia task imposes on the system may change too, for instance, if it adapts its behavior to available resources.

An important assumption of this work is that we expect users to run the system at, or near, full load. A video played at full speed can consume hundreds of millions of cycles per second—just a few multimedia tasks can easily demand more resources than the system has available. When the system runs out of resources, a multimedia application may be able to adapt its behavior to the available load; for instance, a video can decrease its frame rate or resolution. We will see that BERT provides the system with another response to overload—one that complements adaptive applications.

With this characterization of the workload in mind, BERT has three important properties:

- BERT provides appropriate qualities of service to both best effort and realtime tasks, with the former receiving a fair share of the CPU and the latter meeting their deadlines.
- BERT meets deadlines of realtime tasks even when the computation requirements of the tasks change rapidly. It does not assume perfect knowledge of a realtime task's future resource requirements—it is not necessary for the task to make a conservative (or even fairly accurate) reservation to achieve its deadlines.
- System performance degrades gracefully under load. BERT implements a flexible, user-driven policy to discriminate between important and unimportant

tasks. BERT continues to provide good quality of service to important tasks in overload (i.e., best effort fair share and realtime deadlines), possibly at the expense of unimportant tasks.

BERT's achieves these properties through the use of two innovations. First, BERT introduces a new mechanism, called *stealing*, which is a fine-grain extension to a proportional share scheduler that dynamically shifts cycles between tasks at run time. This allows a realtime task to exceed its reservation by stealing cycles to meet a deadline. Second, BERT defines a stealing policy that represents task priority in a new way. While it is obvious that a realtime task should be able to steal unallocated cycles to make its deadlines when the system is underloaded, BERT's key insight is that stealing can help achieve the goals of important tasks when the system is overloaded. In this setting, importance means that a task has a capability to steal (if realtime) or immunity from having its cycles stolen (if best effort). Prioritizing tasks in this way enables the system to dedicate enough resources to satisfy important tasks, yet run unimportant tasks even when there is important work enqueued.

2 Background

This section presents the context in which BERT was designed. First, it describes strategies that have been used to schedule a mix of best effort and realtime tasks, with the goal of introducing the key issues a multimedia scheduling algorithm must address. Second, it gives a brief overview of the operating system—Scout [12]—in which BERT runs, and identifies the key assumptions BERT makes about the underlying execution model.

2.1 Scheduling

A simple approach to scheduling both best effort and realtime tasks is to implement a separate thread queue for each: an EDF queue [11] for realtime tasks and a FIFO (or priority) queue for best effort tasks. The system services the EDF queue if there is realtime work to do; otherwise the FIFO queue is serviced and best effort tasks get to run. As long as the system is not too heavily loaded, this approach works well: all realtime tasks meet their deadlines with enough cycles left over for the best effort tasks to make progress. The problem is that during loaded conditions, the EDF portion of the algorithm tends to select tasks whose deadlines have already passed—thereby wasting time on work that ultimately has no value—and the best effort tasks are starved.

Multimedia systems are often heavily loaded so the above scheme is untenable, but there is an even more fundamental problem—giving realtime tasks priority over best effort tasks is not always the right thing to do. It is possible that a best effort task (a compile) is more important to the user than a realtime task (a video). Also, one realtime task (video-conference with boss) may be more important than another realtime task (South Park video). Unfortunately, solutions that employ multiple EDF queues for different priorities of realtime traffic do not address this problem since they degenerate to strict priority scheduling. Giving tasks precedence based on their importance to the user, rather than their types, is one of the goals of the current work.

An alternate scheme to the EDF/FIFO queues is to use a *proportional share* scheduler. A proportional share scheduler allocates some some fraction of the CPU to each task, where proportions are often allocated in a hierarchical fashion [19, 7, 8, 2, 18, 5].[1] Basic proportional share schedulers have been shown to do a reasonable job at mixing best effort and realtime tasks, but they fall short in one key area: allowing realtime multimedia tasks to meet their deadlines. Because the approach has been so popular, and since BERT follows it up to a point, we give a a brief history (see [17] for a more extensive survey).

Many proportional share schedulers are descended from the Fair Queuing algorithm originally developed to schedule packets for transmission in a network switch [4]. The initial idea is to give each flow (which is the network equivalent of a task) $1/N$ of the link bandwidth, where N is the number of flows. This approach has been generalized into Weighted Fair Queueing [3], in which each flow receives an amount of bandwidth based on its weight and the weights of all active flows. Subsequent research has established many theoretical results about fair queuing, and countless variants have been proposed [6, 8]; the identifying characteristic of these FQ descendants is that the abstraction of *virtual time* lies at their core. The particular algorithm that forms the basis of BERT, cryptically known as WF^2Q+, originated in the networking world [2].

A number of CPU scheduling algorithms use mechanisms derived from or similar to fair queueing [19, 7, 17]. However, transitioning proportional sharing from scheduling bandwidth to cycles is not trivial. For instance, a main component of the network fair queueing algorithms is knowledge of how long it takes to send a packet; this is easily derived from the packet length and link speed. The analogy to packet length in CPU scheduling is work execution time, which is harder to quantify since, even with a preemptive scheduler, a piece of work may not use its whole timeslice because it blocks on an I/O request or otherwise yields the processor. In addition, an OS has to deal with interrupts and other system events that may take place outside of the control of the proportional share scheduler. For these and other reasons, much research on proportional share scheduling has proceeded independently of work in network fair queueing.

[1] Hierarchical schedulers also allow different algorithms (e.g., EDF, FIFO) to be used to order threads at the leaves of the hierarchy, but this capability is not relevant to our discussion.

More recently, research on proportional share scheduling has focused on scheduling multimedia realtime tasks. Proportional share algorithms ensure that a task receives cycles at a guaranteed rate; a realtime task makes its deadlines if its share is sufficiently large. The key question is: how large? It is not obvious how to select the right share for a multimedia task, even though the share we give a multimedia task plays a critical role in its perceived quality.

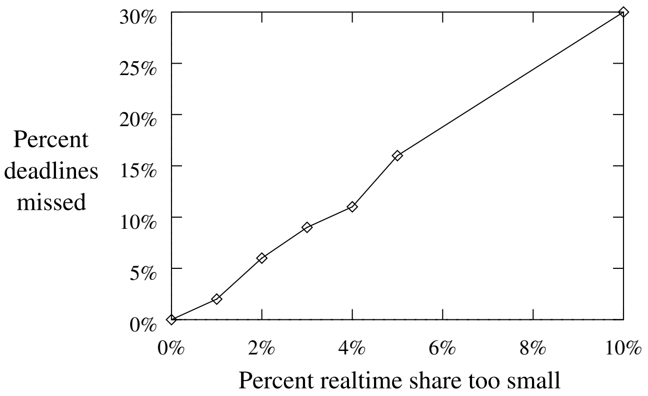

Figure 1: Share vs. Deadlines

Figure 1 shows the results of an experiment in the Scout operating system, illustrating the dependence of a realtime video task on its share. With a "perfect" share, the video misses no frame deadlines. However, for each 1% that the video's rate falls short of the perfect share, the video misses about 3% of its deadlines. To make these numbers more concrete, with a reservation of 110 million cycles per second (Mcps), the video ran at 30 frames per second without missing any deadlines; but with a reservation of 100 Mcps, just slightly smaller, it missed about a third of its deadlines. The video might miss fewer deadlines if it is able to adapt, but the point remains: with a proportional share scheduler, the quality of a realtime task such as a video depends on its share.

One possible way out of this dilemma is to allocate realtime tasks a conservatively large share. Since we wish to fully utilize our system's resources, we reject this approach. Another solution is to "fix" proportional sharing by weakening the realtime task's dependency on its share for meeting deadlines. The latter is the approach taken by the SMART [14] and Rialto [10] schedulers, which we discuss more fully in Section 5. This is the goal of BERT as well.

2.2 Scout OS

Scout is a configurable OS designed to support data streams such as MPEG-compressed video. It does this by defining a *path* abstraction that encapsulates data as it moves through the system, for example, from input device to output device. Each path is an object that encapsulates two important elements: (1) it defines the sequence of code modules that are applied to the data as it moves through the system, and (2) it represents the entity that is scheduled for execution.

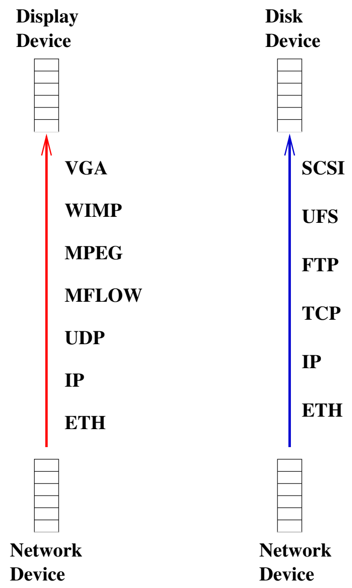

Figure 2: Sample Scout Paths

Figure 2 depicts a pair of Scout paths: the path on the left implements an MPEG video stream that transforms network packets into video frames, and the path on the right corresponds to an FTP path that moves incoming network packets to a disk device. We say the former is a realtime path and the latter is a best effort path. In this figure, each path has a source and a sink queue, and is labeled with the sequence of software modules that define how the path "transforms" the data it carries. Focusing on the MPEG path, ETH is the device driver for the network card, IP and UDP are the conventional network protocols, MFLOW is an MPEG-aware transport protocol, MPEG implements the MPEG video decompression algorithm, WIMP is the window manager, and VGA is the device driver for the graphics card.

Operationally, network packets that arrive for a particular data stream are inserted into the source queue for the corresponding Scout path. Since there may be multiple paths active in the system at a given time, Scout first classifies each incoming packet according to the path to which it belongs. Once enqueued on a path, a thread is scheduled to shepherd this message along the path; this thread inherits its scheduling parameters from the path. When the thread runs, it executes the sequence of modules associated with the path, and deposits the message in the sink queue. The sink device (display or disk) periodically removes messages from the sink queue.

The scheduling parameter assigned to a thread depends on the type of path to which it belongs. A thread associated with a realtime path is assigned a deadline in a path-specific manner. For example, the MPEG path assigns a deadline according to the number of frames currently buffered in the sink queue and the rate at which that queue is being drained by the display device. In an early version of Scout (before BERT), a thread associated with a best effort path, such as the FTP path, was assigned a priority. The Scout scheduler then implemented two thread queues: threads associated with realtime paths were inserted in an EDF queue based on their deadlines [11], while threads associated with best effort paths were inserted into a priority queue. The system then serviced the EDF queue as long as it was not empty, with the priority queue receiving any remaining cycles. (Alternatively, this early version of Scout could be configured to statically allocate some fraction of the CPU to servicing the EDF queue and the remaining fraction to servicing the priority queue.)

Three aspects of this execution model are relevant to BERT. First, each path runs to completion; Scout does not preempt a running path. In other words, we assume a cooperative system in which each path holds the CPU for a bounded—and relatively small—amount of time. One way to think of a path is that it corresponds to one iteration of a conventional server loop that (1) inputs an item of work, (2) performs some computation, and (3) outputs a result. Scout essentially preempts this server loop once per iteration, that is, once per path execution. Note that with respect to BERT, the assumption of non-preemption is only necessary for the realtime paths. It would be possible to specify that certain best effort paths be preemptable, although the modules configured into such paths would need to be programmed accordingly.

Second, we assume it is possible to know approximately how long a path is likely to execute before the path is selected to run. Most paths have fairly deterministic behavior—this is true of the FTP path—making it easy to predict execution time based on the average number of cycles needed by past executions of the path. The MPEG path is a notable exception, where the number of cycles required by a given execution may vary by a factor of two or more from the average. Fortunately, we have been able to devise a sophisticated and computationally efficient predictor for MPEG paths that is able to estimate the number of cycles needed to within 25% of the number actually required [1]. This magnitude of error is well within the tolerance of BERT. Note that such a prediction is not an issue for preemptable best effort paths since we would predict the running time to be equal to the time slice.

Third, we assume that each realtime task is runnable at any time between when it becomes available and its deadline. In other words, there are no dependencies due to inter-task synchronization or I/O dependencies.

3 Algorithm

Stealing is a fine-grain extension to proportional share that gives realtime tasks extra cycles to meet specific deadlines.[2] BERT uses stealing both to decrease realtime tasks' sensitivity to their share, and to allow realtime tasks to meet deadlines when the system is overloaded.

This section builds up an efficient implementation of the stealing mechanism from its theoretical roots. First, it explains stealing in the context of the General Processor Sharing (GPS) theoretical fluid model. It then moves from the fluid model to discrete scheduling algorithms, and discusses how to implement stealing in one particular algorithm: Worst-case Fair Weighted Fair Queueing Plus ($\mathrm{WF^2Q+}$). Finally, it presents optimizations that reduce the computing requirements of stealing to constant time.

A final subsection then describes how the algorithm is manifested in the BERT scheduler, and in particular, outlines the policy that BERT implements on top of stealing. This policy defines task importance, and shows how importance can be used to control what tasks are allowed to steal, and from whom.

3.1 Stealing in GPS

Generalized Processor Sharing is a theoretical model for fair queueing; we can think of GPS as a kind of cycle-by-cycle round robin scheduling [4]. In any time interval in GPS, each task receives an amount of service in proportion to its weight relative to the weight of all active tasks. This means that, by definition, GPS guarantees a task a certain bandwidth. For a task i with a weight ϕ_i, GPS guarantees the task a rate g_i of:

$$g_i = \frac{\phi_i}{\sum \phi_j} r \tag{1}$$

where r is the total rate available (i.e., clock speed of processor) and $j = 1, 2, ...N$ up to the number of tasks in the system.

For simplicity, we assume that a task's weight is expressed in terms of a reservation. A task reserves 100 million cycles per second, for instance, and 100 million becomes its weight. This means that $\sum \phi_j \leq r$ and thus $g_i \geq \phi_i$, that is, a task is guaranteed to receive at least its reserved rate regardless of the set of active tasks.

Suppose that a piece of work has a deadline by which it must have completed service. From the duration of the work and the task's reserved rate, we can calculate the clock time by which it will definitely have finished. If its deadline is after its guaranteed finish time, then the deadline will be met and we need do nothing more.

The idea behind stealing is simple. If the work's guaranteed finish time is after its deadline, the work cannot meet its deadline at its reserved rate. However, in this case we may still be able to satisfy the deadline if the work receives additional service time before the deadline. We know how many extra cycles the work needs—it is the

[2] We are using the term *stealing* in a different sense than in [10], where it referred to an undesirable interaction between mechanisms in the scheduling algorithm.

number of cycles it executes after its deadline if the task gets only its reserved rate. We will explicitly claim this many additional cycles for the task in the interval before its deadline; if we are successful, the deadline will be met. The challenge is to give the work the cycles it needs without unintentionally impacting other tasks.

To give the work the extra cycles it needs, we temporarily increase the weight of the task. The trick is to ensure that the sum of all weights is never greater than r, the rate of the CPU; if it ever is, then we cannot assert that $g_i \geq \phi_i$ for all tasks i and this means some tasks will not get their reservations [9]. Let us assume, without loss of generality, that $\sum \phi_j = r$ (we will see later why we can do this; as a hint, unreserved capacity is essentially no different than a permanently idle task.) In this case, if we increase the weight of one task, we must also decrease the weights of some other tasks or else risk over-subscribing the CPU. This is where "stealing" gets its name: we temporarily reduce the weights of a well-defined set of tasks so that a realtime task can increase its own weight and steal cycles from them. If we do this right, we will give the realtime task enough cycles to meet its deadline while affecting only those tasks with reduced weights.

The method by which stealing manipulates task weights can be best described in the fluid model of GPS. Let task i be the task that has its rate temporarily increased to meet the deadline of work w_i, and let j be a task that has its rate reduced. During the stealing interval, whenever it is task j's turn to execute a cycle, the cycle goes to work w_i instead. Task i continues to execute w_i during its own turn. In the interval, task i has an effective weight of $\phi_i + \phi_j$ while task j's weight is 0. This continues until task i has received enough extra cycles to be sure to meet the deadline of w_i. To summarize, the goal of stealing is to halt specific tasks for a definite interval, while one task executes at more than its reserved rate to meet a deadline.

3.2 Stealing and Virtual Time

Many discrete scheduling algorithms have been proposed to approximate the GPS fluid model. A common characteristic of these algorithms is that they use the concept of "virtual time" to trace GPS's behavior. In GPS, the virtual time of a task i is defined to flow at an instantaneous rate of $1/\phi_i$. Virtual time for the system flows at a rate of $1/\sum \phi_j$, where j ranges over the set of all active tasks at the instant in question. In the fluid model, the system virtual time and the virtual times of every task are always the same.

When a piece of work begins service, we can calculate the system virtual time at which it will finish (the VFT) as follows. Let t be the time at which work w_i of task i begins service, $l(w_i)$ be the duration of the work, and $V(t)$ be the system virtual time at time t. Then the virtual time of the task when the work completes will be:

$$VFT(w_i) = V(t) + l(w_i)/\phi_i \tag{2}$$

Since in the fluid model system virtual time is always equivalent to task virtual time, this will also be the system time.

Worst-case Fair Weighted Fair Queueing Plus (WF^{2}Q+) is a network packet scheduling algorithm that was designed to closely approximate the behavior of GPS. Like most fair queueing algorithms, it enqueues each piece of work (i.e., a packet) with a timestamp representing its VFT in the fluid model as described in Eq. (2) above. It then services the queue in order of increasing stamps. WF^{2}Q+ has several desirable properties, which will be more fully explained later, and for which we selected it to form the basis of our BERT scheduler. Having derived stealing in the context of our theoretical model GPS, we actually implement it in WF^{2}Q+.

Stealing clearly changes the virtual finishing times of packets in the fluid model. When stealing occurs in GPS, the VFT of a piece of work in the system may be moved forward or back relative to when it would have completed without stealing—forward if we are stealing to make deadline, and back if the work is stolen from. This gives us a clue how to implement stealing: we need to calculate the new VFT in the fluid model for each piece of work affected by stealing, and stamp the work with its new VFT. Then, if we service the work in order of increasing stamps, as WF^{2}Q+ already does, the cycles will be shifted to the stealing task. This means that we must quantify exactly how much the VFT of a piece of work is changed by stealing. We do this next.

First we consider the task i that we want to receive an increased service rate. Let w_i be the work in question, t be the current time, and $D(w_i)$ be the work's deadline. Between now and its deadline, we want to give the work an overall rate ϕ_i' so that it receives at least $l(w_i)$ cycles by the deadline. That is:

$$\frac{l(w_i)}{\phi_i'} = D(w_i) - t \tag{3}$$

We do not need to calculate ϕ_i' . The VFT of work w_i at the new rate will be $V(t) + l(w_i)/\phi_i'$. Using the above equation, we can substitute for the second term; the virtual time at which the work will be finished is simply:

$$VFT(w_i) = V(t) + (D(w_i) - t) \tag{4}$$

This is the timestamp that w_i receives as a consequence of stealing.

Next we turn attention to the tasks that task i steals from. Let Ψ be the set of tasks whose rates are decreased, and let ϕ_Ψ be the sum of all reservations for tasks in Ψ. Recall the idea of stealing: during some time interval, task i will take the cycles that normally would go to a task $j \in \Psi$. During this interval, the effective weight of task i is $\phi_i + \phi_\Psi$, while the weights of all tasks in Ψ are 0.

How many cycles does task i need to take from tasks in Ψ to make work w_i's deadline? At its normal rate of ϕ_i, and given that t is the time at which w_i starts service, task i is guaranteed to receive $\phi_i(D(w_i) - t)$ cycles by w_i's deadline. So task i needs to take δ worth of cycles, where δ is:

$$\delta = l(w_i) - \phi_i(D(w_i) - t) \tag{5}$$

Tasks in Ψ have a combined reserved rate of ϕ_Ψ. At this rate, the virtual time by which all these tasks together will have received δ cycles in the fluid model is:

$$V(t') = V(t) + \frac{\delta}{\phi_\Psi} \tag{6}$$

During the virtual time interval $[V(t), V(t')]$, all tasks in Ψ don't receive any cycles. Thus, the VFT of work belonging to each task is delayed by:

$$V(t') - V(t) = \frac{\delta}{\phi_\Psi} \tag{7}$$

So the timestamp of each active task in Ψ is increased by δ/ϕ_Ψ.

We must be careful though. What we are trying to do is to shift cycles between tasks' allocations in real time. The task that steals to make a deadline must receive all of the cycles before the deadline arrives or else they are too late. This means two things. First, we must track the time that we've already stolen. Assume task i has stolen all the cycles of tasks in Ψ until virtual time $V(t')$—another task cannot steal these same cycles. Thus, we must keep track of the virtual time within which other tasks are stealing the cycles of tasks in Ψ. That is, we need to save the value of $V(t')$ from the last steal in a variable $V_\Psi(t)$. When task i steals, it must calculate

$$V(t') = max(V_\Psi(t), V(t)) + \frac{\delta}{\phi_\Psi} \tag{8}$$

and save this new value of $V(t)$ in $V_\Psi(t)$.

Second, before each steal we must check to make sure that the amount of cycles that tasks in Ψ will receive before the deadline are at least what task i needs. We can do this by comparing the value $V(t')$ calculated in Eq. (8) with the timestamp assigned to work w_i by the steal in Eq. (4); that is, we must check that:

$$V(t') \leq V(t) + (D(w_i) - t) \tag{9}$$

If so, then the steal will be completed—task i will have received all its extra cycles, and tasks in Ψ will have started up again—before the deadline has arrived. The steal will succeed.

Finally, stealing affects idle tasks in Ψ differently than active ones. If a task is idle during a stealing interval in the fluid model, the VFT of its subsequent work is not affected. If work for an idle task arrives when stealing is going on, then it is delayed only until the interval is over. Since $V_\Psi(t)$ represents the virtual time at which stealing ends, the VFT of work w_j belonging to idle task $j \in \Psi$ can be calculated as:

$$VFT(w_j) = max(V(t), V_\Psi(t)) + l(w_j)/\phi_j \tag{10}$$

In other words, in the fluid model, arriving work for an idle task is delayed until the end of the stealing interval; if no task is currently stealing, the work is not delayed at all.

3.3 Stealing in WF^{2}Q+

We have outlined how stealing changes the virtual times of packets in the fluid system. The algorithm we have chosen as the foundation of BERT, called WF^{2}Q+, closely approximates GPS in its behavior. However, there are two significant differences between GPS and WF^{2}Q+ that can impact our stealing algorithm. Next we briefly discuss the salient points of WF^{2}Q+ and how they affect stealing.

WF^{2}Q+ is a fair queueing algorithm that was originally designed to schedule network packets [2]. A packet is *eligible* in WF^{2}Q+ if the system virtual time is at least as large as the time that the packet would begin service in the fluid model; there is always at least one eligible packet. WF^{2}Q+ stamps each packet with its VFT in the fluid model, as described previously, and services the eligible packet with the smallest stamp.

We chose WF^{2}Q+ because it possesses some desirable theoretical properties and it is efficient to implement. The theoretical properties that concern us are *delay* and *workahead*. In any discrete algorithm that approximates GPS, a piece of work can finish after when it would have in the fluid model; this is *delay*. WF^{2}Q+, like many fair queueing algorithms, has been shown to have delay bounded by the largest piece of work present in the system. The delay is significant for scheduling realtime tasks: a task is only guaranteed to meet its deadline in WF^{2}Q+ if it will be done in the fluid model by the deadline minus the delay bound. We also need to account for the delay when stealing; we can easily do this by subtracting the delay bound from the work's actual deadline before checking to see if we need to steal.

A scheduling algorithm is *fair* to the extent that two active tasks receive cycles in proportion to their weights. GPS is perfectly fair by definition. In contrast, the Virtual Clock algorithm [20] is infinitely unfair: when an idle task starts up in Virtual Clock, it can monopolize the processor for an unbounded period of time. The lower the unfairness of a fair queueing algorithm, the more closely it approaches GPS in its behavior. Networking researches have been able to formally quantify the maximum unfairness of many fair queueing algorithms.

The *workahead* of a fair queueing algorithm is closely related to its fairness. Workahead refers to a positive difference between a task's virtual time and the system virtual time—that is, the task has worked ahead relative to system time[3]. For instance, the reason that Virtual Clock is completely unfair is that a task's virtual time can run arbitrarily far ahead of the system virtual time (in the case of Virtual Clock, this is the system clock). Since a new or idle task has its virtual time initialized to system time, it gets to run until its virtual time has caught up with the times of other tasks. Virtual Clock is unfair because the workahead of a task is unbounded. By contrast, Bennett and Zhang have shown the workahead of a flow i in WF^{2}Q+ to be bounded by the quantity $(1 - r_i/r)L_{i,max}$, where r_i is the reservation of the flow, r is the rate of the server, and $L_{i,max}$ is the maximum-sized chunk of work allowable for the flow. Similar results have been shown for other algorithms [17].

The stealing mechanism must be mindful of the workahead of tasks. This is because the workahead of tasks in Ψ represents future cycles (in GPS) that these tasks have already received (in WF^{2}Q+). In other words, even though the cycles were available to be stolen in GPS, they have already been used in WF^{2}Q+. To avoid the problem of a task stealing cycles that it cannot later receive, we must subtract the potential workahead of the active tasks in Ψ when we calculate the cycles available for stealing.

Finally, we chose WF2Q+ for its low computational complexity. One of the most important features of a fair queueing algorithm is how it maintains the system virtual time. In GPS, virtual time flows at the rate of the reciprocal of the sum of the weights of all active tasks. This means that as the active task set changes, the rate of change of GPS system virtual time changes. Many fair queueing algorithms try to approximate the GPS system time, as this leads to improved fairness; some even run a GPS simulator to track the exact state of the fluid model. However, completely simulating GPS is expensive—*O(n)* each time the active task set changes—and complicated. Most recent algorithms use simplified approaches.

WF^{2}Q+ approximates the system virtual time by looking at the virtual times of active tasks. The virtual start time of a task is its virtual time before it puts a piece of work on the queue—it is the time that the work would start service in the fluid model. It has been shown that the system virtual time must be at least as great as the smallest virtual start time of active tasks. However, the system virtual time also flows at least as quickly as clock time. Therefore, the system virtual time can be more cheaply extrapolated—in *O(log n)*—by looking at the minimum virtual start time of a task and the change in clock time. This gives a straightforward and efficient way to approximate the GPS system virtual time.

[3] In [17], workahead goes by the name of *negative lag*. Also, *positive lag* is another name for delay.

3.4 Implementation and Optimizations

Now that we have explained stealing in the GPS fluid model, quantified its effects using the virtual time abstraction, and shown how it can be realized in WF^{2}Q+ by adjusting the timestamps of packets to account for differences between the fluid model and the discrete algorithm, we turn our attention to how stealing can be implemented in practice. In particular, this section describes optimizations that reduce its complexity to linear time.

WF^{2}Q+ approximates the GPS fluid model by stamping work with its virtual finishing time and servicing the eligible piece of work with the smallest stamp. To implement stealing, we simply add the following few steps to the WF^{2}Q+ process of timestamping a piece of work.

Task i is a realtime task that is permitted to steal to meet the deadline of work w_i. $VFT(w_i)$ is the virtual finish time of the work calculated by WF^{2}Q+, $V_i(t)$ is the current virtual time of task i, and $V(t)$ is the system virtual time. $D(w_i)$ is the deadline for work w_i minus the WF^{2}Q+ delay bound, and $l(w_i)$ is the duration (length) of the work. As before, Ψ is the set of tasks from which stealing is allowed—tasks in Ψ are not also allowed to steal, though they may be realtime—and ϕ_Ψ is the sum of weights of tasks in Ψ. Finally $V_\Psi(t)$ is the timestamp representing the virtual time until which tasks in Ψ have had their cycles stolen. So, if stealing is allowed for task i, we perform the following prior to stamping the work:

1. $VD \leftarrow V(t) + (D(w_i) - t)$
2. If $VD \geq VFT(w_i)$ then work will make deadline; stamp work w_i with $VFT(w_i)$ and stop
3. $\delta \leftarrow l(w_i) - \phi_i(D(w_i) - t)$ and $V_\Psi(t) \leftarrow max(V_\Psi(t), V(t))$
4. If $VD < V_\Psi(t) + \delta/\phi_\Psi$ minus the workahead for active tasks in Ψ, there are not enough cycles to steal; stamp work w_i with $VFT(w_i)$ and stop
5. Stamp work w_i with VD; $V_i(t) \leftarrow VD$
6. For all $j \in \Psi$, increment $V_j(t)$ by δ/ϕ_Ψ. This involves changing stamps of already enqueued work belonging to active tasks.
7. Increment $V_\Psi(t)$ by δ/ϕ_Ψ

If stealing is allowed for task i, at (1) we calculate VD, work w_i's deadline in virtual time. In (2) we check to see if the work will make deadline at the task's usual rate. If not, in (3) we calculate δ (the amount of cycles we need to steal) and $V_\Psi(t)$ (the timestamp which keeps track of previously stolen cycles). At (4) we check to see if tasks in Ψ will receive enough cycles by the deadline. If so, in (5) we stamp the work with its virtual deadline and update $V_i(t)$ (task i's virtual time), and in (6) we update the virtual timers and work timestamps for tasks in Ψ. In (7) we update $V_\Psi(t)$ to remember that a task has already stolen cycles up until this virtual time.

The problem with the algorithm as outlined above is step (6); in this step, the stamps of all work belonging to tasks in Ψ are updated to reflect stealing. A straightforward implementation of this step would be to dequeue each piece of work belonging to these tasks, change the stamp, and then enqueue it again. If the ready queue is a simple linked list, the complexity of this implementation would be $O(n^2)$. This is too high. To complete the stealing algorithm, we present a two-stage optimization of this step to reduce its computational complexity to constant time.

We can trim the compute time of step (6) to $O(n)$ by a simple mechanism. In the naive implementation described above, we must dequeue and re-enqueue a piece of work when we change the stamp because its relative position in the output queue may change. However, stealing adds the same amount to the stamps of all work belonging to active tasks in Ψ, so the order in which work for these tasks is executed does not change. We can break the ready queue into *two* queues—one for tasks in Ψ and one for all other tasks—and execute the work with the lowest timestamp for both queues. The result is that we don't need to dequeue/enqueue pieces of work in Ψ when we change their stamps; since they are in their own queue, our algorithm can change their stamps in place. Traversing the Ψ queue is $O(n)$.

Once we have a separate queue for the tasks in Ψ, another optimization brings the computation time of this step to $O(1)$. Stealing adds the same amount to the stamps of all work in the Ψ queue. Rather than traverse the queue and re-stamp all the work in it, we can associate a delay counter (let this be called D_Ψ) with the queue and add the constant to the counter. The stamp of a piece of work in the Ψ queue thus becomes its actual stamp plus the counter.

Simply incrementing a counter in the manner described above penalizes idle tasks in Ψ. Recall from section 3.2 that, in GPS, stealing affects tasks which are idle at the start of a stealing interval differently than active ones. If work for an idle task is submitted in the middle of a stealing interval, it is only delayed for the remainder of the interval; if no work is submitted during the interval, subsequent work for the task is not delayed at all.

Recall that using the D_Ψ counter was simply a shortcut for actually changing the timestamp of each piece of work and the virtual timer of its task. We can skip this shortcut if we want to. Our solution is, when a task changes from idle to active status, we update the virtual time of the task directly and bypass the counter. We associate a credit variable C_j with each task $j \in \Psi$; initially all C_j are 0. When the task becomes active, we set $V_j(t) \leftarrow max(V_\Psi(t), V(t))$ and $C_j \leftarrow D_\Psi$. The variable C_j is used to cancel out the delay that does not apply to the task because it is reflected in the task's virtual time counter $V_i(t)$. Now the timestamp of a piece of work w_j of task $j \in \Psi$ becomes its actual stamp plus $(D_\Psi - C_j)$. When we compare the stamps of the tasks at the heads of our two ready queues, we use this value as the stamp of the head of the Ψ queue.

The additional ready queue, and D_Ψ and C_j counters reduce the stealing algorithm from quadratic to constant time. These optimizations add only a small overhead—a few arithmetic operations—to each scheduling decision.

3.5 BERT Scheduler

The BERT scheduler combines the WF^{2}Q+ proportional share scheduling algorithm with our new stealing mechanism to seamlessly schedule both realtime and best effort tasks. The only remaining issue is one of policy: what tasks are allowed to steal, and from whom.

When the system is not fully loaded, BERT addresses the weaknesses of using a proportional share algorithm to schedule realtime tasks by allowing realtime tasks to steal the cycles needed to meet their deadlines. Specifically, BERT maintains all of the system's unreserved CPU cycles in an "unallocated pool". The unallocated pool is essentially the same as a permanently idle task—this means it can be treated like a task in Ψ and stolen from. When the system is not fully loaded, BERT allows a realtime task that may not make a deadline with its current share to steal the cycles it needs from the unallocated pool.

In overload, BERT allows system performance to degrade gracefully by differentiating between important and unimportant tasks. In such a situation, there are no unallocated cycles available for stealing. As a consequence, improving the performance of the video task implies that the performance of other tasks must decline. BERT prioritizes tasks in overload, and then uses the stealing mechanism to satisfy the differing quality of service metrics of best effort and realtime tasks. BERT's priority scheme is simple: tasks are either important or unimportant. In our implementation, the user selects the importance of a task through clicking a button on the task's window. In this way the system acquires a new pool of cycles—those belonging to unimportant tasks—from which to steal.

	Realtime	Best Effort
Important	steal	immune
Unimportant	stolen from	stolen from

Figure 3: Priority in BERT

Figure 3 summarizes the effects of BERT's two-level priority scheme. Important realtime tasks can steal to meet their deadlines if their shares are too small. Important best effort tasks should not be inconvenienced for the sake of other tasks; they are not stolen from. Unimportant tasks—whether realtime or best effort—are fair game for stealing. BERT's method of allocating resources based on task importance allows the system's performance to degrade in an intuitive and graceful way. Note that this priority scheme could be generalized beyond two levels, but it is not clear from our experience that additional priority levels would be of any value to the user.

BERT resembles a hybrid of proportional sharing and EDF that avoids the downside of each. If a realtime task's share is sufficient to allow it to make its deadlines, BERT schedules according to share. If the share is too small, BERT schedules the task's work according to its deadline, like EDF, by stealing cycles. Unlike EDF, BERT checks that there are enough spare cycles to ensure the work will meet deadline once it is scheduled. BERT seamlessly blends EDF and proportional sharing to increase the quality of realtime tasks whose shares are too small, in both underload and overload.

Finally, we observe that a complementary mechanism—feedback between the application and scheduler [13, 15]—meshes nicely with BERT. Feedback can be used to adjust the application's share in the scheduler, based for instance on how many deadlines are being missed or how full/empty the task's work queues have become. Feedback also allows the application to adapt its behavior to the available resources of the system. However, feedback is a coarse-grained solution that cannot ensure that a realtime task meets all its deadlines over all intervals. BERT uses the fine-grain stealing mechanism to fill in the cracks left by such feedback mechanisms. It ensures individual deadlines are met while the system negotiates resource levels, providing fine-grain smoothing while coarse-grain mechanisms operate. Also, if an attempt to steal fails, BERT can inform the application that it cannot guarantee this deadline, thereby allowing the application to discard the work or otherwise adapt its behavior.

4 Demonstration

We have implemented BERT in the Scout operating system. This section demonstrates how BERT distributes cycles to paths in underloaded and overloaded conditions.

Figure 4 shows BERT allowing a realtime path to steal the cycles it needs from the unallocated pool. This test was run on a 300MHz Alpha and lasted about 30 seconds. Each row in Figure 4 corresponds to one path; the first two are best-effort paths and the third is a realtime path. In the first interval (A), two best-effort paths are running, each with a reservation of 100 million cycles per second (Mcps). (Each path's reservation is shown by the dotted line; the solid line shows its actual usage.) The two best effort paths each receive closer to 150 Mcps because BERT

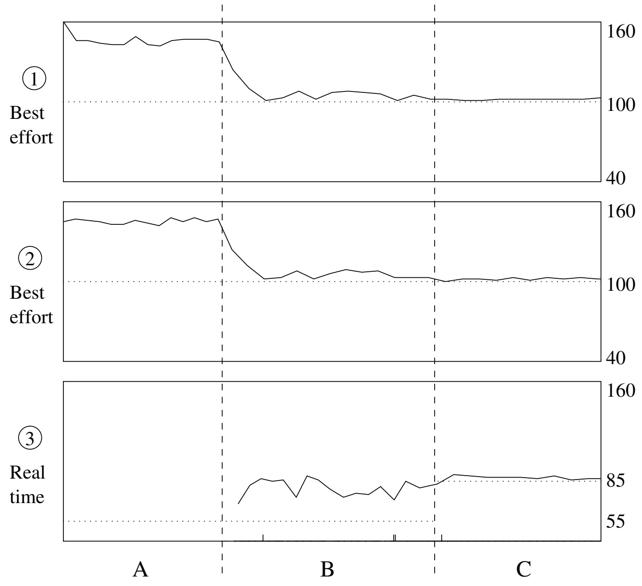

Figure 4: BERT Underload Demonstration

is fairly dividing the unreserved cycles. At the beginning of interval B, a realtime path (path 3) starts up with a reservation of 55 Mcps—this was the system's guess as to what reservation the path would need. However, this reservation is too small; the realtime path requires about 85 Mcps to meet its deadlines. BERT detects that deadlines would be missed at the realtime path's reservation, and steals extra cycles from the unallocated pool to meet them.[4] In interval C, the system has observed that the reservation of the realtime path is, in fact, too small and adjusts it. After this point, the realtime path meets nearly all of its deadlines without stealing.

Figure 5 shows a larger trace, which also demonstrates how BERT performs in overload. It traces the CPU usage of four Scout paths for about a minute on a 300MHz Alpha. At the start of trace (interval A), paths 1 and 2 are running; each is a best-effort path with a reservation of 75 Mcps. As in the previous example, the paths fairly share excess cycles between themselves.

In interval B, path 3 start up; it is a realtime video path with a reservation of 55 Mcps. Again this was too low, but path 3 is able to steal the unallocated cycles it needs to meet its deadlines. In the middle of the interval, the system corrects path 3's reservation to 85 Mcps. At this point, reservations in the system total 235 Mcps.

Another realtime path (path 4) starts up in interval C. It could only obtain a reservation of 50 Mcps, since our system allocates a maximum of 285 Mcps (the remaining 15 Mcps is reserved for system events). This is inade-

[4]To confirm that stealing is actually happening, we note that 255 Mcps of the total machine capacity is reserved, leaving only 45 Mcps unreserved. If the 45 Mcps is divided up fairly (in proportion to path weight), the realtime path would receive only 10 Mcps over its reservation; in fact, it receives closer to 30 Mcps because it steals them.

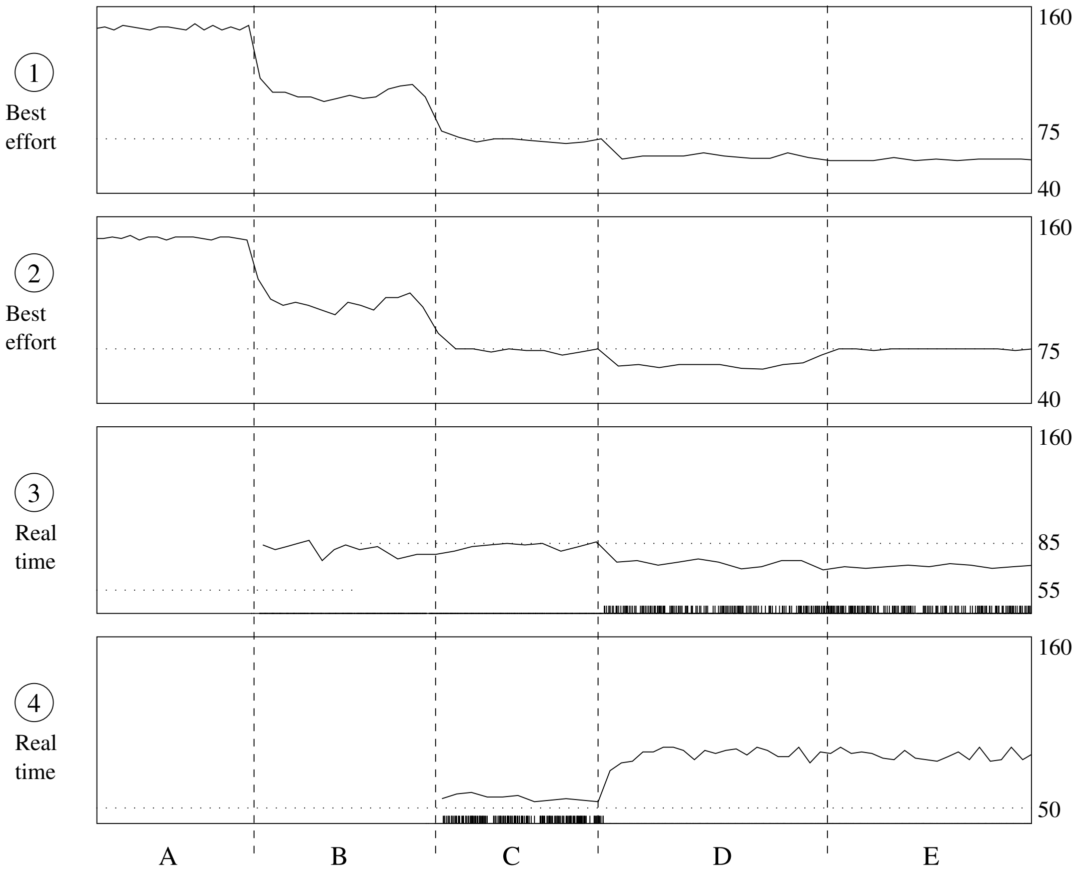

Figure 5: BERT Overload Demonstration

quate, and the task starts missing deadlines. (Each small tic on the X axis denotes a missed deadline). The system is clearly overloaded.

At the start of interval D, the user selects path 4 as important. By default, the other three paths are unimportant and so path 4 steals cycles from them to meet its deadlines. Paths 1 through 3 all receive fewer cycles than they have reserved; the loss of cycles causes path 3 to miss some deadlines. However, subsequently path 4's deadlines are all met.

The user chooses path 2 as an important path at the start of the final interval. Since path 2 is a best-effort path, it does not steal; instead, importance means it cannot be stolen from. The usage of path 2 climbs back up to its reservation of 75 Mcps. Path 4 is now stealing cycles from paths 1 and 3 only; their usages each fall a little more. The test concludes with important paths 2 and 4 enjoying good performance, while unimportant paths 1 and 3 receive less than their reservations. At the end of the test, only path 2 is actually receiving the number of cycles it had reserved. Assuming that the user is pleased with the behavior of the system at this point, each task's reservation could be reset to its actual usage (this is not shown in the test).

The previous experiments illustrate how BERT distributes bandwidth among paths to achieve different qualities of service. When the system is underloaded, BERT ensures that the best-effort paths receive their reserved shares and a realtime path met its deadlines. In overload, an important realtime video path whose reservation is too small steals to meet its deadlines, while the immunity from stealing granted to an important best-effort path allows it to receive its fair share. BERT responds to both inaccurate reservations and user preferences to produce intuitive, desirable system behavior.

5 Related Work

The ideas of BERT most closely resemble some recent packet scheduling work from the networking community. Generalized Fair Queueing (GFQ) contains the germ of the idea behind stealing [9]. GFQ is a generalization of the class of fair queueing packet algorithms that permits individual packets within a flow to have different weights. A flow can temporarily increase its weight as long as it does not cause the server capacity to be exceeded. The GFQ work was mainly theoretical and did not describe mechanisms by which an implementation could avoid overallocating the server; however, its approach inspired our arguments in Section 3. In fact, the operation of the stealing mechanism in underload could be interpreted as an implementation of GFQ for WF^2Q+ (or for any fair queueing algorithm with known delay and workahead bounds): stealing provides a way of claiming unallocated cycles up to, but not exceeding, the CPU capacity. BERT goes beyond GFQ by describing the algorithm; providing additional mechanisms to allow stealing from other tasks; presenting a working implementation; and placing task speedup in the framework of importance to the user.

The H-FSC packet scheduler by Stoica and Zhang [18] bears some resemblance to BERT as well. H-FSC is a fair-queueing scheduler that decouples bandwidth from delay for realtime packet flows. In a sense, BERT does this too when it steals to make a deadline. The two approaches are intended to solve different problems though. In packet scheduling, the working definition of a "realtime" flow is simply that a bandwidth reservation can be made for it. Realtime flows are interested in keeping their delays under a certain bound, but they do not have deadlines in the same sense that the realtime CPU tasks do. The H-FSC algorithm can deliver tighter delay bounds for higher priority flows. It is not clear, however, if H-FSC's ideas could be applied to CPU scheduling to guarantee that important realtime tasks meet their deadlines.

The SMART scheduler [14] starts with a proportional share CPU scheduler and then tries to account for deadlines; this sounds very similar to BERT. However, SMART treats work with deadlines very differently. Specifically, SMART orders tasks based on logical timestamps. Whenever the task at the head of the run queue is a realtime task, SMART looks down the queue to find all runnable realtime tasks up to the first best effort task, and then reorders those realtime tasks according to their deadlines (using EDF). The advantage of BERT's approach is that it can make stronger guarantees about meeting deadlines than SMART. Also BERT can achieve graceful behavior in overload by providing good performance to tasks that are important to the user.

Tasks can have priorities in SMART, but in the conventional way: priorities are layered on top of the primary algorithm. This means that high-priority tasks are always scheduled before low-priority tasks. As a result, low-priority tasks can miss deadlines or starve even if the high-priority tasks have deadlines far in the future. In other words, SMART gives precedence to priorities over deadlines, and therefore can not seamlessly integrate the two like BERT.

The Rialto scheduler [10] has many of the same capabilities as BERT. An "activity" in Rialto reserves an execution rate by specifying an amount of cycles and a period. Rialto then computes a schedule, whose repeated execution guarantees that the activity receives the cycles during each period. Furthermore, realtime activities that need extra cycles to make a deadline can claim empty slots in the schedule; otherwise the empty slots are evenly distributed among activities. Both Rialto and BERT allow CPU reservations to be met. Both provide reservations at a fine granularity—Rialto at the granularity of the reserved period and BERT at the granularity of the GPS fluid model (instantaneous) plus WF^2Q+'s delay bound. Both algorithms can guarantee that realtime work will be completed by its deadline when the work is accepted by the scheduler, and can allow tasks to react when a deadline cannot be guaranteed.

An important difference between BERT and Rialto is the conceptual level at which each approaches the problem of scheduling multimedia tasks. Rialto's low-level solution depends on explicitly ordering all the tasks in the system via the repeating schedule. If a realtime activity needs more than its reservation, it can be assigned empty slots in the schedule. In contrast, BERT builds its solution on an abstraction—virtual time. It doesn't matter in BERT what the ready queue looks like; theory tells us that the work will be done by a certain time. BERT's theoretical model also provided the context for deriving the stealing mechanism. Not only does stealing allow realtime tasks to claim unused time, like Rialto, but it also enables important realtime tasks to take time from unimportant tasks *in a precise and controlled manner*. By abstracting the problem away from the ready queue, BERT provides a simple and elegant approach to multimedia scheduling.

Jeffay *et al.* distinguish between a realtime task's *share* and its *weight* [16]. The task's weight is set by the system and remains constant, but its share—the fraction of the processor that the task actually receives—fluctuates depending on the weights of the other active tasks. Jeffay argues that we care primarily about a realtime task's share, not its weight, and proposes a scheduler that computes the correct weight for a realtime task given its share. We did not try to exploit this insight in BERT. However, we could avoid unnecessary stealing from the unallocated pool by calculating a realtime task's share and checking if the share is sufficient to make the deadline even if the task's reservation is not.

6 Conclusion

BERT is a new CPU scheduler that combines a fair queueing packet scheduling algorithm (WF^2Q+) with the novel mechanism of stealing. BERT allows best effort tasks to receive a fair share of the CPU while realtime tasks meet their deadlines. Moreover, by supporting a policy that allows important realtime tasks to steal cycles and important best effort tasks to be immune from stealing, BERT provides graceful degradation of system performance in overload. BERT does not require the user to specify accurate reservations, and it does not presume that realtime tasks are always more important than best effort tasks. We have implemented BERT in the Scout operating system, and shown it to be both practical and effective in scheduling multimedia workloads.

References

[1] A. Bavier, B. Montz, and L. Peterson. Predicting MPEG execution times. In *Proceedings of the SIGMETRICS/PERFORMANCE '98 Symposium*, pages 131–140, June 1998.

[2] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. In *Proceedings of the SIGCOMM '96 Symposium*, pages 143–156, Palo Alto, CA, Aug. 1996. ACM.

[3] D. Clark, S. Shenker, and L. Zhang. Supporting realtime applications in an integrated services packet network: Architecture and mechanism. In *Proceedings of the SIGCOMM '92 Symposium*, pages 14–26, Aug. 1992.

[4] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proceedings of the SIGCOMM '89 Symposium*, pages 1–12, Sept. 1989.

[5] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, WA, Oct. 1996. USENIX Assoc.

[6] S. J. Golestani. A self-clocked fair queueing scheme for high speed applications. In *Proceedings of IEEE INFOCOM '94*, pages 636–646, Apr. 1994.

[7] P. Goyal, X. Guo, and H. Vin. A hierarchial CPU scheduler for multimedia operating systems. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 107–122, Seattle, WA, Oct. 1996.

[8] P. Goyal, H. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of the SIGCOMM '96 Symposium*, pages 157–168, Palo Alto, CA, Aug. 1996. ACM.

[9] P. Goyal and H. M. Vin. Generalized guaranteed rate scheduling algorithms: a framework. *ACM Transactions on Networking*, 5(4):561–571, Aug. 1997.

[10] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 198–211, Oct. 1997.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(20):46–61, Jan. 1973.

[12] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 73153–168, Oct. 1996.

[13] S. J. Mullender, I. M. Leslie, and D. McAuley. Operating system support for distributed multimedia. In USENIX Association, editor, *Proceedings of the Summer 1994 USENIX Conference: June 6–10, 1994, Boston, Massachusetts, USA*, pages 209–219, Berkeley, CA, USA, Summer 1994. USENIX.

[14] J. Nieh and M. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the Sixteenth Symposium on Operating System Principles*, pages 184–197, Oct. 1997.

[15] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 145–158, New Orleans, LA, Feb. 1999. USENIX Assoc.

[16] I. Stoica, H. Abdel-Wahab, and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. In *Proceedings of Multimedia Computing and Networking 97*, pages 207–214, Feb. 1997.

[17] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, 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*, pages 288–299, Dec. 1996.

[18] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, realtime and priority services. In *Proceedings of the SIGCOMM '97 Symposium*, Cannes, France, Sept. 1997. ACM.

[19] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, Nov. 1994. USENIX Assoc.

[20] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, 9(2):101–124, May 1991.