BERT: A Scheduler for Best Effort and Realtime Tasks

Andy Bavier, Larry Peterson, and David Mosberger*

Department of Computer Science
Princeton University
Princeton, NJ 08544

August 10, 1998

Abstract

We describe a new algorithm, called BERT, that can be used to schedule both best effort and realtime tasks on a multimedia workstation. BERT exploits two innovations. First, it is based on the virtual clock algorithm originally developed to schedule bandwidth on packet switches. Because this algorithm maintains a relationship between virtual time and real time, our algorithm allows us to simultaneously factor realtime deadlines into the scheduling decision, and to allocate some fraction of the processor to best effort tasks. Second, BERT includes a mechanism that allows one task to *steal* cycles from another. This mechanism is valuable for two reasons: it allows us to distinguish among tasks based on their importance, and it results in a robust algorithm that is not overly sensitive to the task having made a perfect reservation.

1 Introduction

One of 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 scheduling algorithm, 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. In terms of CPU scheduling, the goal of each best effort task is to make progress by getting some fair share of the CPU.

*Mosberger's address: Hewlett-Packard Laboratories, Palo Alto, CA.

Multimedia realtime tasks are exemplified by decoding MPEG compressed video. The primary goal of such tasks is to complete certain pieces of work by a pre-defined deadline (e.g., decode a video frame once every 33ms), although the user can tolerate occasional missed deadlines (i.e., we are interested in soft realtime). To complicate matters, the computational requirements of a given task may vary significantly over short time intervals; for example, the cycles needed to decode back-to-back video frames can differ by a factor of five or more. Moreover, many of the realtime tasks will be adaptive, which means the load they impose on the system may change over longer time intervals.

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. On the other hand, multimedia applications can adapt their behavior to the available load (e.g., a video can decrease its frame rate or resolution) and since users want to receive as much value as possible out of their system, we expect high load to be the rule rather than the exception.

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

- It defines a unified framework that allows both best effort and realtime tasks to achieve their goals, with the former receiving their fair share of the CPU and the latter meeting their deadlines.
- It is robust in the face of imprecise reservations. Stated another way, BERT does not avoid overloaded conditions by depending on a conservative admission control mechanism. Neither does it assume that a realtime task must have made a perfect reservation.
- It degrades gracefully under load. Specifically, BERT discriminates between important and unimportant tasks, with the former allowed to meet their goals, possibly at the expense of the latter.

BERT accomplishes this through two innovations. First, it applies the *virtual clock* packet scheduling algorithm [15] to the problem of scheduling cycles. Virtual clock defines a common framework for scheduling both realtime and best effort tasks. Second, BERT exploits the fact that virtual clock maintains a relationship between real and virtual time: it uses a new mechanism called *stealing* to factor the deadlines of important tasks into the scheduling decision. The innovation of stealing allows BERT to tolerate imprecise reservations, and discriminate among tasks based on their importance.

2 Related Work

This section describes various strategies for scheduling a mix of best effort and realtime tasks, with the goal of both reviewing related work and introducing the key issues a scheduling algorithm must address.

A simple approach to scheduling both best effort and realtime tasks is to implement a separate thread queue for each: an EDF queue [8] for realtime tasks and a FIFO (or PRIORITY) queue for best effort tasks. As long as the system is not fully utilized, this approach works perfectly well: all realtime tasks meet their deadlines and there are enough cycles left over for all the best effort tasks to make progress. The problem, of course, is that under loaded conditions, the EDF portion of the algorithm tends to select tasks whose deadlines have already passed, and the best effort tasks are starved as a result.

Giving realtime tasks priority over best effort tasks—even if some fraction of the CPU is reserved for best effort so as to prevent starvation—is an arbitrary decision. It is possible that some of the best effort tasks are more important than some of the realtime tasks. It is also possible that some realtime tasks are more important than other realtime tasks. An intuitively appealing way around this difficulty is to extend the notion of priority to the realtime tasks as well as the best effort. In other words, a realtime task of priority p would never fail to meet its deadline due to a task, realtime or best effort, of priority less than p. Unfortunately, such an approach is problematic—in a realistic environment, this scheme would quickly deteriorate into strict priority scheduling.

To see this, consider a pair of realtime tasks A and B. Task A is low-priority, requires one unit of execution time and has a deadline at 1; task B is high-priority, requires one unit of execution time and has a deadline at 2. Should the scheduler run task A or task B? The answer is that if it is to *guarantee* that task B does not miss its deadline because of a lower-priority task, it *must* schedule task B right away. Otherwise, it could become necessary to run another high-priority task at A, in which case it would be impossible for both high-priority tasks to make their deadlines. In other words, because the scheduler cannot foresee the future, it has to be conservative and schedule higher-priority tasks as soon as they are ready to run—completely independent of task deadlines.

An alternative is to use a *proportional share* scheduler to allocate some some fraction of the CPU to each task, where proportions are often allocated in a hierarchical fashion [13, 6, 7, 2, 12, 4].[1] Proportional share schedulers are all descended from the fair queuing algorithm originally developed to schedule packets for transmission in a network switch [3].

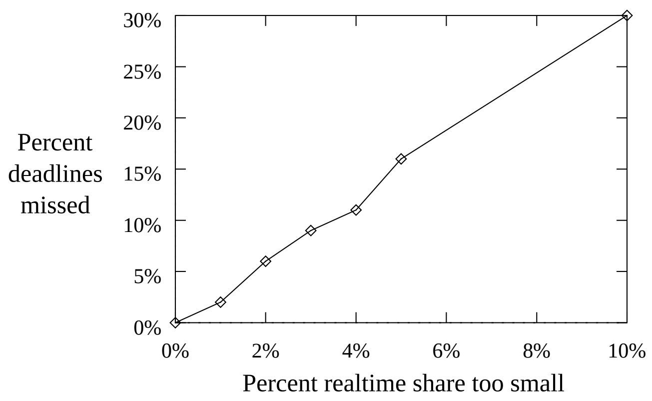

Figure 1: Share vs. Deadlines

[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.

The primary difficulty with proportional share algorithms is that realtime deadlines don't play any part in the scheduling decision. The intention is simply to give a realtime task a large enough share so that its deadlines are met. However, for dynamic and variable tasks such as MPEG video, there is no apparent connection between shares and deadlines to indicate the share such a task needs. Figure 1 highlights the problem by showing the relation between a realtime video task's share and its ability to make deadlines. With a perfect share, the video misses no deadlines. However, for each 1% that the video's rate falls short of the perfect share, the video misses about 3% of its deadlines. With a proportional share scheduler, the quality of a realtime task such as a video can depend heavily on finding the right share.

This presents a dilemma: On the one hand, realtime tasks can be allocated a conservatively large share. We do not find this approach appealing due to our desire to fully utilize the system's resources. On the other hand, we can accept that realtime tasks will sometimes have inadequate shares and distinguish between tasks based on importance. The problem is, proportional share algorithms do not provide the means to do so. For example, consider two realtime tasks of varying importance, both living within their reserved shares, and suppose there are only enough cycles for one to make its next deadline. Proportional share schedulers can not even detect this situation, let alone react to it.

This is not an issue in packet scheduling since the working definition of a "realtime" task—a task is typically called a *flow* in a packet switch—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. In this context, one recent algorithm of particular note is Stoica and Zhang's H-FSC packet scheduler [12], which decouples bandwidth from delay. The algorithm can deliver tighter delay bounds for higher priority flows. What's not clear, however, is how a CPU scheduling algorithm can take advantage of this decoupling to guarantee that high-priority realtime tasks meet their deadlines.

One promising solution is to use feedback to adjust the share, or alternatively, to adapt the application's requirements [10]. Feedback might be based on how many deadlines a given realtime task is missing or how full/empty the task's work queues have become. While such feedback seems like a good idea, our experience is that it should be viewed as a coarse-grained solution used to keep allocations roughly in line with actual usage for the sake of admission control; it is not fine-grained enough to ensure that a realtime task meets all its deadlines over short intervals. In other words, a feedback mechanism is complementary to the scheduler; a multimedia system should include both.

Another general approach—the one explored in this paper—is to start with a proportional share scheduler, but then to factor deadlines into the equation. This is essentially what the SMART scheduler does [11]. Specifically, SMART orders tasks based on logical timestamps, but 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. BERT takes a different approach to factoring deadlines into the scheduling decision, as spelled out in later sections. The advantage of BERT's approach is that it can make a stronger guarantee about meeting deadlines, and it is able to arbitrarily discriminate among tasks based on their priority.

Note that SMART does have priorities, but they are layered on top of the primary algorithm. This means that high-priority tasks are always scheduled before low-priority tasks, even if it is possible to meet the latter's deadlines. In other words, SMART gives precedence to priorities over deadlines, and therefore suffers from the limitations outlined above.

3 Framework

This section lays the foundation for understanding BERT. First, BERT has been designed to run in the Scout operating system [9]; Section 3.1 gives a brief overview of Scout, with the goal of identifying the key assumptions BERT makes about the underlying execution model. Second, BERT uses the Virtual Clock (VC) packet scheduling algorithm [15] to provide a "common currency" managing both realtime and best effort tasks; Section 3.2 outlines the VC algorithm and shows how it can be applied to scheduling CPU cycles in Scout. Finally, a key feature of the VC algorithm is that it maintains a relationship between virtual and real time; Section 3.3 explains how BERT exploits this relationship to discriminate between high and low priority tasks.

3.1 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 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

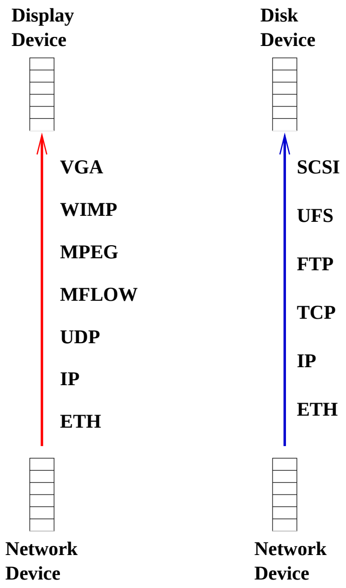

Figure 2: Example Scout Paths

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. A thread associated with a best effort path, such as the FTP path, is assigned a priority. The scheduler in an early version of Scout implemented two thread queues: threads associated with realtime paths were inserted in an EDF queue based on their deadlines [8], 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, as discussed later. 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.

3.2 Virtual Clock

Virtual clock is an algorithm originally devised to deliver a reserved rate on a network link, but it can be used to manage other resources as well. Each task i—here we use the word "task" in the generic sense of a resource consumer—begins with a reserved average rate AR_i for a resource. For each task, the algorithm maintains a counter, VC_i, used to assign timestamps to units of work. Each piece of work submitted by a task is assigned a timestamp and enqueued as follows:

1. Upon receiving the first piece of work for task i, $\mathsf{VC}_i \leftarrow \mathsf{REALTIME}$.
2. Upon receiving each subsequent piece of *work* for task i:
   (a) $\mathsf{VC}_i \leftarrow \mathsf{max}(\mathsf{REALTIME}, \mathsf{VC}_i)$;
   (b) $\mathsf{Vtick}_i \leftarrow \mathsf{duration}(work)\ /\ \mathsf{AR}_i$;
   (c) $\mathsf{VC}_i \leftarrow (\mathsf{VC}_i + \mathsf{Vtick}_i)$;
   (d) Mark *work* with timestamp VC_i.
3. Insert *work* into service queue, ordered according to increasing timestamps.

The virtual clock algorithm was just described in generic terms to detach it from packet scheduling. In the form stated above, VC can manage other resources, such as the CPU, as long as a mapping can be found between elements of the algorithm and components of the problem space. Table 1 illustrates the mappings used to apply the VC algorithm to its original purpose of packet switching, as well as to CPU scheduling in Scout.

Generic Concept	**Packet Switching**	**CPU Scheduling in Scout**
resource	link bandwidth	CPU bandwidth
task	flow	path
work unit	transmitting a packet	one path invocation (or timeslice)
duration of work	packet transmit time	path execution time

Table 1: Mapping Virtual Clock onto Scout Paths

Two things should be pointed out about mapping from VC to a particular problem space. First, the resource to be managed must have a time component; e.g., bits or cycles *per second*. Second, the work durations must be known in advance to calculate the Vtick_i variable. This is trivial in the case of packet switching—it is the number of bits in the packet divided by the link speed. When doing CPU scheduling, it is also trivial for best effort tasks that can be scheduled to run for a fixed-length timeslice. Realtime work with a deadline, however, must not be preempted before it completes. Therefore, we must try to predict the execution time of realtime work; in Scout, this means we need to know how long it will take to execute a path. Fortunately, we can gather this information, as described in the previous subsection.

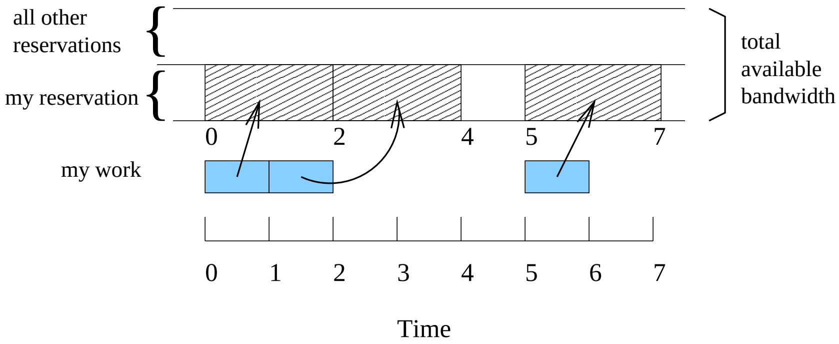

Figure 3: Intuition behind Virtual Clock

Intuitively, virtual clock works by assigning slices of reserved bandwidth to individual quanta of work. Figure 3 shows an example. In the figure, time flows from left to right, with vertical space representing bandwidth (e.g., 300Mcps). Suppose one task has reserved reserves half the CPU bandwidth on a particular machine—the wide bar at the top of the picture represents the available system bandwidth, and the task's reservation filling the bottom half. Work for the task arrives at times 0, 1, and 5; each piece of work has an execution time of 1. The task assigns slices of its reserved CPU bandwidth to each piece of work by assigning the work a timestamp and updating the task's virtual clock. So, the first two pieces of work receive timestamps of 2 and 4, corresponding to the shaded areas of the reservation. The third piece of work arrives at time 5. At this time, the task's virtual clock, which had a value of 4, is updated to the system time of 5. This piece of work is therefore assigned a timestamp of 7. Note that the task did not get to assign the time it had reserved between 4 and 5 to any task because there was no eligible work to give it to.

3.3 Virtual and Real Time

The virtual clock algorithm requires a task to make an absolute reservation, in our case, some number of cycles per second. It would be easy to implement a proportional share interface on top of this reservation; that is, a task could request 20% of the CPU and be given a reservation of 60M cycles-per-second on a 300MHz machine. In this respect virtual clock is very much like proportional share algorithms. In particular, a best effort task can reserve some fraction of the CPU and be assured it will make progress according to this reservation.

Virtual clock goes beyond proportional share, however, in that it maintains a connection between real time and the virtual timestamp assigned each task. (This connection is implemented in line 2a above.) Specifically, it has been proven that the work will finish by its virtual timestamp, as long as the resource is not over-reserved [14].[2] In other words, the timestamp is a *share deadline*, or the time by which the work should be done if the task is receiving its reserved rate. The work is actually done by its share deadline. Simply put, proportional share ensures a given task will receive the cycles it reserved, while virtual clock also promises that the task will receive these cycles by a particular time.

BERT exploits this relationship between virtual and real time to factor deadlines into the scheduling decision. If the timestamp (the share deadline) of a piece of work falls before its realtime deadline—keep in mind that only realtime tasks have realtime deadlines—then we have a *guarantee* that the work will make its realtime deadline since it will be done by its timestamp; we know this fact from the moment the thread is put on the ready queue. In contrast, if the timestamp falls after the deadline, there is a chance the task will not meet its deadline, and in fact, the difference between the virtual timestamp and the deadline corresponds to the number of cycles the task will be short. If the task is of high priority, then it may make sense to *steal* the missing cycles from some lower priority task. It is by allowing tasks to steal—and be *immune* from stealing—that BERT distinguishes between important and unimportant tasks. Figure 4 summarizes how we use stealing and immunity to represent priorities for realtime and best effort tasks.

Priority \ Type	RT	BE
Hi	steal	immune
Lo	stolen from	stolen from

Figure 4: Priority in BERT

3.4 Stealing Overview

The idea of stealing is intuitively simple. The trick is that just any cycles won't do; the additional cycles must be received before the deadline expires. To complicate matters, due to the variability

[2] Actually, the work will finish by its virtual timestamp plus the longest allowable piece of work. For simplicity, this delta is ignored in our discussion but should be taken into account by the system, for instance, all deadlines could be moved up by this amount.

of realtime tasks like video for which frame decode times are not known *a priori*, how many extra cycles will be needed and when they will be needed is not known until the task is ready to run. To meet this challenge, a dynamic, fine-grain mechanism is needed to give realtime tasks extra cycles to meet specific deadlines. The rest of this section gives an overview of how the BERT algorithm addresses these challenges. We do not give a formal proof, but instead present an intuitive argument. To simplify the description, we assume two priority levels: high and low. We discuss how BERT can be extended to support a more general priority scheme in Section 6.

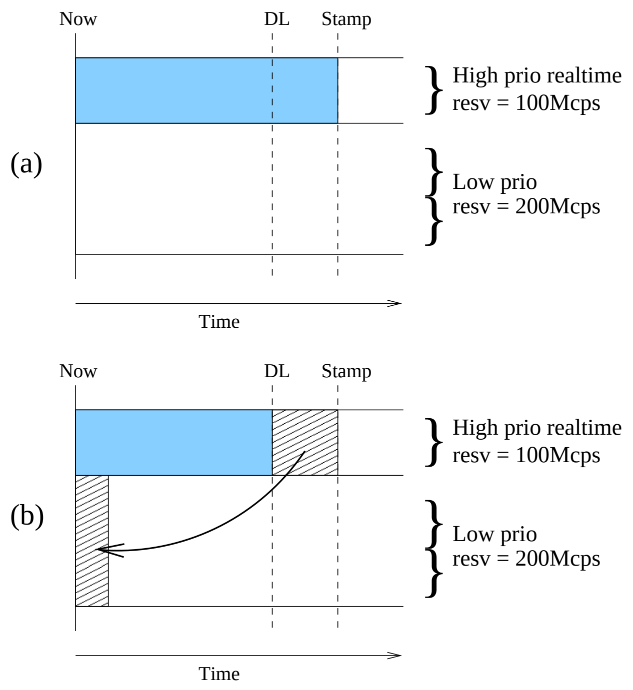

Figure 5: BERT Stealing

To see how stealing works, consider the following example, which is illustrated in Figure 5. Suppose a high priority realtime task has reserved one-third of the available CPU rate and multiple low priority tasks have reserved the remaining two-thirds; these reservations are shown graphically, as in Figure 3. In (a), the high priority task receives a piece of work with a deadline and assigns a VC timestamp to it—that is, the task assigns its reserved cycles between now and the timestamp to the work. The stamp of the work is after the deadline, however, so the deadline may not be met. In (b), the realtime task steals from the best effort tasks. Instead of using its own reserved bandwidth *after* the deadline, it uses a corresponding amount of the low priority tasks' allocation *before* the deadline. Note that both diagonally-shaded boxes have the same area—they correspond to the same amount of cycles. The high priority task can then set the timestamp of the work to its deadline, and the deadline is met. BERT accomplishes this reallocation of bandwidth by delaying the low priority tasks for a certain period of *virtual* time. Applying this *virtual delay* to the low priority tasks frees up their reserved cycles during the period, and the high priority task can use them to make its deadline.

Stealing works because it is careful not to violate virtual clock's relationship between virtual and real time. This relationship holds as long as the resource is not over-reserved—in other words, as long as two tasks don't allocate the same slice of bandwidth. In order to give a high priority realtime task a piece of a low priority task's reservation, we have to ensure that the low priority task does not try to use the same cycles. This is accomplished by delaying the low priority task for the period of virtual time stolen—another way to think of this is that we have effectively incremented the timestamps of all low priority work, bumping them back in the ready queue. The low priority work will no longer finish by its original timestamp but rather its timestamp plus the virtual delay. In this way, stealing preserves virtual clock's realtime aspect, allowing the high priority task to receive the stolen bandwidth in time to make its deadline.

4 Algorithm

This section describes the BERT algorithm in detail; an outline for the algorithm is given in Figure 6. At the core of BERT is the Virtual Clock algorithm. Like Virtual Clock, BERT maintains two variables for every task i: a reserved average rate AR_i and a virtual time counter VC_i. BERT also maintains a third variable—Credit_i—for every low priority task; the meaning of this variable will be explained later in this section.

BERT implements two ready queues, denoted Q_{Hi} and Q_{Lo}, each of which is sorted by timestamp and contains high and low priority tasks, respectively. It also associates three global variables with the low priority queue: Delay_{Lo}, VC_{Lo}, and AR_{Lo}. These three variables play a role in managing of the cycles that have been stolen from the low priority tasks, collectively. The idea is that the algorithm doesn't have to keep track of the cycles stolen from each low-priority task on an individual basis, but can instead account for the total number of cycles stolen and adjust each individual tasks timestamp only when it reaches the head of the ready queue (Q_{Lo}). This is critical for efficiency. Specifically, Delay_{Lo} counts the total number of cycles stolen from the low-priority tasks so far, while VC_{Lo} and AR_{Lo} serve as a composite virtual clock and average rate for all low-priority tasks. Finally, variable $\mathsf{MaxWork}$ is the longest possible work duration for the system.

Although the algorithm presented in Figure 6 deals with both high- and low-priority tasks together, we simplify the explanation by tracing how the algorithm treats each priority separately. We start with the high-priority tasks, and then trace what happens to low-priority tasks. A third subsection discusses step 6, which is an enhancement to the basic virtual clock algorithm to account for tasks that work ahead by using up idle cycles.

4.1 High-Priority Tasks

For high-priority tasks, BERT is equivalent to Virtual Clock in steps 1 and 2(a). Step 3 implements stealing, but only if the high-priority task is also realtime. BERT first computes how many cycles the

1. Upon receiving the first piece of work for task i, $\mathsf{VC}_i \leftarrow \mathsf{REALTIME}$;
   if task i is low priority, $\mathsf{Credit}_i \leftarrow \mathsf{Delay}_{Lo}$
2. Upon receiving each piece of $work$ for $task_i$,
   (a) If task i is high priority:
      i. $\mathsf{VC}_i \leftarrow \mathsf{max}(\mathsf{REALTIME}, \mathsf{VC}_i)$;
      ii. $\mathsf{Vtick}_i \leftarrow \mathsf{duration}(work)\ /\ \mathsf{AR}_i$;
      iii. $\mathsf{VC}_i \leftarrow (\mathsf{VC}_i + \mathsf{Vtick}_i)$;
      iv. Mark $work$ with timestamp VC_i.
   (b) If task i is low priority:
      i. If $\mathsf{REALTIME} > (\mathsf{VC}_i - \mathsf{Credit}_i) + \mathsf{Delay}_{Lo}$,
         $\mathsf{VC}_i \leftarrow \mathsf{REALTIME}$;
         $\mathsf{Credit}_i \leftarrow \mathsf{Delay}_{Lo}$
      ii. $\mathsf{Vtick}_i \leftarrow \mathsf{duration}(work)\ /\ \mathsf{AR}_i$;
      iii. $\mathsf{VC}_i \leftarrow (\mathsf{VC}_i + \mathsf{Vtick}_i)$;
      iv. Mark $work$ with timestamp $(\mathsf{VC}_i - \mathsf{Credit}_i)$.
3. If task i is high priority realtime, and the timestamp of $work$ is after its deadline:
   (a) $\mathsf{need} \leftarrow (\mathsf{VC}_i - \mathsf{deadline}(work)) \times \mathsf{AR}_i$;
   (b) $\mathsf{VC}_{Lo} \leftarrow \mathsf{max}(\mathsf{REALTIME}, \mathsf{VC}_{Lo})$;
   (c) $\mathsf{avail} \leftarrow (\mathsf{deadline}(work) - \mathsf{VC}_{Lo}) \times \mathsf{AR}_{Lo}$;
   (d) If $\mathsf{avail} >= \mathsf{need}$,
      i. $\mathsf{VC}_i \leftarrow \mathsf{deadline}(work)$;
      ii. Mark $work$ with VC_i;
      iii. $\mathsf{VC}_{Lo} \leftarrow \mathsf{VC}_{Lo} + (\mathsf{need}\ /\ \mathsf{AR}_{Lo})$;
      iv. $\mathsf{Delay}_{Lo} \leftarrow \mathsf{Delay}_{Lo} + (\mathsf{need}\ /\ \mathsf{AR}_{Lo})$.
4. Place $work$ in Q_{Hi} or Q_{Lo} as appropriate.
5. Select a task to run: if $\mathsf{head}(\mathsf{Q}_{Hi}) <= \mathsf{head}\ (\mathsf{Q}_{Lo}) + \mathsf{Delay}_{Lo}$, select first $work$ on Q_{Hi};
   otherwise, select first $work$ on Q_{Lo}.
6. Adjust for working ahead:
   (a) $\mathsf{Vtick}_i \leftarrow \mathsf{duration}(work)\ /\ \mathsf{AR}_i$;
   (b) If task i is high priority, $\mathsf{Vstart} \leftarrow (\mathsf{timestamp}(work) - \mathsf{Vtick}_i) - (\mathsf{MaxWork}\ /\ \mathsf{AR}_i)$;
       else, $\mathsf{Vstart} \leftarrow (\mathsf{timestamp}(work) - \mathsf{Vtick}_i + \mathsf{Delay}_{Lo} - \mathsf{Credit}_i) - (\mathsf{MaxWork}\ /\ \mathsf{AR}_i)$;
   (c) If $\mathsf{Vstart} > \mathsf{REALTIME}$,
       $\mathsf{VC}_i \leftarrow \mathsf{VC}_i - (\mathsf{Vstart} - \mathsf{REALTIME})$.

Figure 6: The BERT Scheduling Algorithm

high-priority task needs to steal in 3a. This is the difference between the task's virtual timestamp VC_i and its deadline, times the rate at which the task has been allocated cycles (AR_i). This product corresponds to the upper shaded area shown in Figure 5. The cycles available to be stolen is computed in 3c, using the aggregate virtual clock and rate for all low-priority tasks (VC_{LO} and AR_{LO}). Before computing avail, however, BERT first has to bring VC_{LO} into sync with real time. This is done in step 3b. In Figure 5, the available cycles are represented by all of the low priority reservation between now and the deadline, assuming that no other task has stolen some of them.

Stealing actually happens in step 3d: the task's virtual clock is reset to the deadline (i) and the work is marked with this value (ii). Figure 5 represents this step by showing how the task moves the upper shaded bandwidth to the low priority reservation. The last two steps in 3d update the low-priority variables to reflect the fact that cycles have been stolen; this bookkeeping is discussed in the next subsection. The work is then placed in the high-priority ready queue Q_{HI} (Step 4). Whenever the system is looking for the next thread to run (Step 5), it selects the first thread in Q_{HI} as long as that thread's timestamp is less than or equal to the timestamp first thread in Q_{LO}, adjusted to reflect how long all the low-priority tasks have been pushed into the future by having cycles stolen from them.

4.2 Low-Priority Tasks

Focusing on variable Delay_{Lo} is the key to understanding how low-priority tasks are scheduled. As we have just seen, a high-priority task that risks not making its deadline computes variable need. This variable is used to reset variables VC_{Lo} and Delay_{Lo} in steps 3d(iii–iv). These two variables are similar in that they accumulate the number of cycles stolen from the low-level tasks collectively. The only difference is that VC_{Lo} is kept in sync with realtime in step 3b, and so it reflects *when* the cycles are available; VC_{Lo} is used in step 3c to determine the available cycle bandwidth at a given point in time. In contrast, Delay_{Lo} records the absolute number of cycles stolen so far; it is used in step 5 to adjust the timestamp of the first task on Q_{Lo} to account for stolen cycles. In other words, all low priority tasks are delayed for an amount of virtual time corresponding to the cycles stolen divided by AR_{Lo}, their combined reservations.

The only other complication occurs in steps 1 and 2b when BERT computes the virtual timestamp for a piece of work (thread) being inserted in the ready queue on behalf of low-priority task i. The problem is that stealing punishes idle tasks. As we have seen, the algorithm accounts for cycles with variable Delay_{LO}, but there is no need to delay tasks that were idle during the time in question. We introduce a per-task variable, Credit_i, to represent the portion of the delay that should not be applied to a previously idle low-priority task: Step 1 sets Credit_i of a new task to the current value of Delay_{LO} so as to not penalized a new task right off the bat; Step 2b(i) updates Credit_i for a task that has been idle longer than the applicable delay; and Step 2b(iv) subtracts Credit_i from the task's virtual timestamp to cancel out the inapplicable delay.

4.3 Adjustments for Working Ahead

It is well known that Virtual Clock exhibits throughput unfairness [5]. The reason is that the VC counter of a task that receives bandwidth in excess of its reserved rate runs far ahead of real time. When a new or idle task starts up, its VC is set to the current real time, and as a consequence, it gets to run until its VC catches up with that of the first task. This behavior raises an additional problem for BERT—stealing depends on the relation between virtual and real time. BERT looks at the difference between the realtime deadline and the work's virtual timestamp to know how many cycles to steal; BERT will grossly overestimate the actual amount if the task has previously—when the system had excess cycles—received a lot of cycles.

The solution—implemented in Step 6—is that a task is not charged for excess bandwidth it receives. As we do in stealing, we exploit Virtual Clock's relationship between virtual and real time to determine if this has happened. Stealing is based on the result that a piece of work finishes no later than its timestamp. If task i receives no more than its share, its work will *start* no earlier than VC_i prior to inserting the work on the queue, minus a constant.[3] So, if the work starts before this time, the task is receiving more than its share and we subtract the difference between the expected and actual start times from the task's VC_i. The primary effect is that virtual time is kept in line with real time when the task receives more than its share.

An interesting side-effect of this mechanism is how excess bandwidth is distributed. BERT is not fair in the conventional sense because this bandwidth is not evenly distributed among tasks. A task's virtual clock is corrected only when it runs. When we subtract from a task's VC, however, it receives a slight advantage over other tasks and gets to run again sooner. As a result, the task that runs most often—i.e., has the largest share—gets almost all of the excess capacity. This is not unreasonable behavior for a CPU scheduler; the best-effort task with the largest share is typically the most important, and if a realtime task is using capacity outside its reservation, its reservation is too small. Controlling the way in which excess cycles are distributed is not one of the main goals of BERT.

4.4 Discarding Useless Work

A powerful feature of BERT is its capacity for early drops of realtime work that will miss its deadline, even it it is allowed to steal. Although not shown in Figure 6, there are two places the algorithm could decide to discard a piece of work, depending on how aggressive it wants to be. To support making this decision, Step 2 needs to be augmented to compute the latest acceptable start time (LAST) for the work. It does this by subtracting the number of cycles needed to execute the path ($\mathsf{duration}(work)$) from the deadline. Then, LAST can be checked in Step 5 when the work is actually selected to run; if LAST is already in the past, the deadline is not expected to be met and the work can be discarded. BERT can be even more aggressive, however, by discarding the work in Step 3, before it even places it on the ready list.

[3] The constant is the amount of virtual time, at the task's rate AR, consumed by the longest-running piece of work allowable in the system. This has been experimentally verified but not yet proven. Also, for low priority tasks, we must subtract the credit from the delay.

There is one complication. Observe that there may still value in doing the work, even if the deadline cannot be met. For example, MPEG video consists of different frame types, and some types of frames depend on other frames—for example, a P frame depends on an I frame. If the I frame is discarded, the P frame is useless, so we may want to decode the I frame even if its deadline has passed. This implies the application needs to have some input into the decision. We are currently experimenting with ways of using application knowledge to decide with discarding late work is appropriate.

5 Demonstration

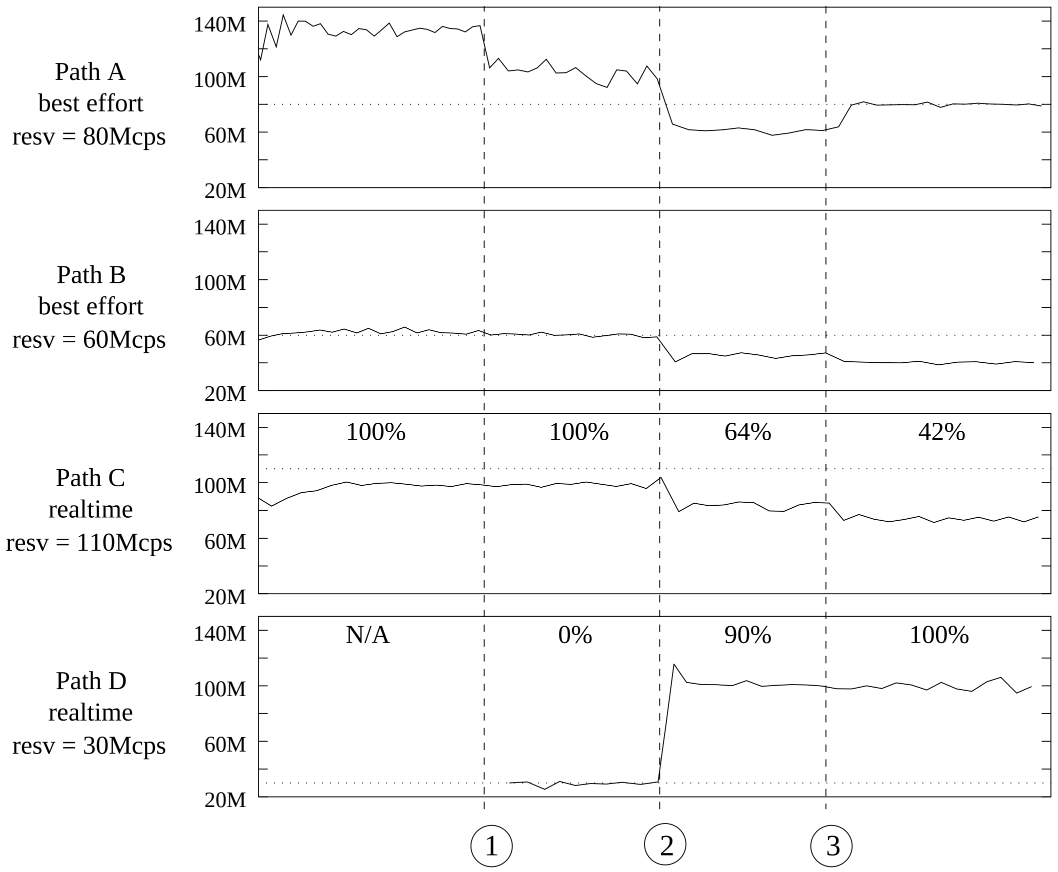

Figure 7: BERT in Action

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

Figure 7 traces the CPU usage of four Scout paths for about a minute on a 300MHz machine. At the start of trace, the top three paths are running. Paths A and B are best effort paths with reservations of 80Mcps and 60Mcps, respectively. Each path's reservation is shown by a dotted line; its actual usage is shown by the solid line. Path C is a realtime video path with a reservation of 110Mcps; this reservation is slightly conservative, as the path requires about 100Mcps. The reservations of all paths totaled 250Mcps, meaning that the system is underloaded.

Between the start and event 1 (marked on the x-axis), both best effort paths receive at least their reserved rates and the realtime video path made all of its deadlines. Path A receives almost all the excess cycles—as discussed in Section 4, our algorithm gives the bulk of extra cycles to the path that has the largest allocation, and is able to consume the cycles. The realtime path C has a higher share, but it does not need any extra cycles to meet its deadlines. Thus, in underload, BERT meets the requirements of all three paths: the best effort paths receive their allocated share, and the realtime path makes its deadlines.

At event 1, admission control allowed a new video stream—path D—to enter the system with a reservation of 30Mcps. This reservation corresponds to all the remaining cycles since Scout reserves the other 20Mcps for system events, such as moving the mouse. This reservation is too small to allow path D to meet its realtime deadlines. At this point, the system has entered an overload situation—there are not enough cycles to satisfy all four paths. Between events 1 and 2, nearly all of path D's deadlines are unmet. (Missed deadlines are not shown on the trace.) Notice, however, that admitting the fourth path has little effect on the other three paths: the other video path does not miss any deadlines in this interval, and the best effort paths receive their reservations (albeit path A received fewer extra cycles).

At event 2, the user clicks a "high priority" button on the window that displays path D's video. This causes path D to start stealing cycles from the other three (unimportant) tasks to meet its realtime deadlines. Immediately, D's utilization jumps from 30 to 100Mcps, even though its reservation remains at 30Mcps. Path D misses a few more deadlines as it parses through the MPEG stream throwing away old frames; it achieves 90% of its deadlines between events 2 and 3, with all of the misses at the beginning of the interval. After catching up, path D does not miss a single deadline for the rest of the experiment—it is able to meet them all through stealing. The effect of D's stealing is visible on paths A, B, and C. Between events 2 and 3, the best effort paths receive about three-quarters of their reserved rates. Realtime path C's rate drops as well, with the consequence that in this interval it makes only 64% of its deadlines.

The user then decides that best effort path A is important, and so clicks its "high priority" button at event 3. This makes the path immune from stealing. After event 3, path A's actual rate rises to its reserved rate of 80Mcps. Path D continues to steal to make its deadlines, but now it was taking cycles from paths B and C only. The rate of these paths drops further; from this point on, path B receives about two-thirds of its reservation and path C makes only 42% of its deadlines.

The previous experiment illustrates, at a coarse-grained level, how BERT distributes bandwidth among paths. When the system is underloaded, Virtual Clock ensured that the best effort paths received their reserved shares, and since its rate is large enough, the realtime path also meets its deadlines. In overload, stealing allows a realtime video path whose reservation was much too small to meet its deadlines, and granting immunity from stealing to a best effort path allowed it receive its fair share in overload conditions.

6 Discussion

This section discusses several issues related to how BERT is used and behaves in practice. One of the key points we make is that it is possible to shield users from having to make reservations.

6.1 Stealing and Realtime Reservations

Allowing realtime tasks to steal the cycles they need to make their deadlines is the primary advantage of BERT over conventional proportional share algorithms. For example, Figure 8 shows the CPU bandwidth used by a realtime task fluctuating over time, as compared to that task's reservation (the dotted line). If stealing is enabled for this task—and there exist other tasks with adequate cycles that are not immune from stealing—then the realtime task is able to steal the cycles it needs to meet its deadlines. (Of course, even without stealing, the task would be able to consume unallocated cycles if any are available.)

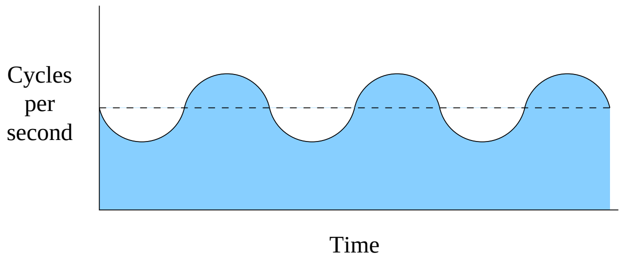

Figure 8: CPU used by a realtime task

In effect, the reservation line in Figure 8 denotes a transition between two different scheduling regimes. When operating below the line, the task is scheduled according to its VC timestamp; above the line, it is scheduled according to its deadline. In other words, BERT is able to slide between proportional share and an algorithm very much like EDF; best effort tasks are always scheduled by the VC half of the algorithm. The relationship between BERT and EDF is discussed in Section 6.3.

Given that realtime tasks can exceed their reserved rates by stealing, one might question the relevance of the reservation; why not have each realtime task make a reservation of 0, and simply steal all the cycles it needs to meet its deadlines? The reason is that we want to support admission control. That is, we want tasks to make at least a "best guess" reservation so we know whether or not to admit them to the system.

Scout currently implements a very simple, but effective, mechanism. When the user wants to start up a realtime application, the application queries Scout to learn the available CPU bandwidth. If the the available rate is above an application-specific threshold, the task is allowed to start with an initial reservation equal to all the unallocated cycles. If not, the task is rejected. As the task runs, Scout monitors its actual CPU usage, and adjusts the reservation accordingly, possibly giving bandwidth back to the unallocated pool. The key is that this feedback mechanism is extremely course-grained; on the order of several seconds. This is sufficient, however, since BERT is able to steal to make up any short-term inadequacies in the reservation.

Note that users are also allowed to manually decrease a task's allocation, so as to free up resources so a more important task can be started. There is no need to allow users to increase a realtime task's reservation, however. Instead, the user would enable stealing for the task and the usage monitor would eventually settle on the right reservation.

6.2 Best Effort Reservations

The preceding discussion focused on realtime tasks. The next question is how to make reservations for best effort tasks. The first thing to keep in mind is that reservations are absolute—they are expressed in terms of cycles per second. It is easy, however, to implement various relative policies on top of this interface. For example, by keeping track of the sum of the cycle rates reserved by all best effort tasks, along with the number of best effort tasks, one could implement a policy that gives each task a relative share of the CPU. Alternatively, one could map weights or priorities (in the traditional sense) onto absolute reservations.

It is our experience, though, that most best effort tasks are I/O-bound, and therefore it is possible to determine the rate at which the task consumes cycles in order to keep some device fully utilized. For such tasks, reservations are handled in much the same way as for realtime tasks: an initial "best guess" allocation is made, the task is monitored for actual usage, and the reservation is adjusted to match this usage rate.

6.3 BERT and EDF

The BERT algorithm can be viewed as a hybrid of Virtual Clock and Earliest Deadline First (EDF). Through stealing, BERT mixes share deadlines and realtime deadlines on the run queue, while providing the guarantees of Virtual Clock to important tasks.

In Section 3.3, we discussed that work scheduled by virtual clock finishes by its timestamp. The timestamp is a *share deadline*, meaning the time by which the work will be done according to the task's share. Virtual clock executes work according to increasing share deadlines—in fact, it is a variant of EDF, with deadlines assigned according to a task's share.

When BERT steals to make a deadline, the timestamp of the work is set to the work's realtime deadline. That is, when a task's share is insufficient to allow it to meet its deadline, BERT schedules the work EDF. The main difference between BERT and EDF is that EDF performs poorly in overloaded conditions, whereas BERT does not. BERT guarantees that if the task can steal successfully, the deadline is met regardless of the overall load on the system. BERT also ensures the task does not interfere with the guarantees made to other high priority tasks. These are properties that EDF does not have.

Knowing the execution time of realtime tasks gives BERT another advantage over EDF. As mentioned in Section 4, BERT can discard realtime work whose latest acceptable start time has already passed before it even runs. BERT can even aggressively throw away realtime work that is not expected to make deadline because its timestamp is too late, thereby allowing the task to spend its cycles on deadlines that are guaranteed to be met. In contrast, EDF is prone to scheduling work that is of no value by the time it completes, thus behaving very poorly under load.

6.4 Generalizing Priorities

The version of BERT presented in Section 4 was simplified to include only two levels of priorities: high-level realtime tasks are allowed to steal from low-priority realtime and best effort task, and high-level best effort tasks are immune from stealing. The cycles a high-priority realtime task steals from a give low-level task is proportional to the latter task's reservation; i.e., twice as many cycles will be stolen from a task with a reservation of 20Mcps than from a task with a reservation of 10Mcps.

In general, it would be easy to extend BERT to support multiple priority levels, in which case a realtime task at priority i is allowed to steal from all tasks at priority $< i$, and a best effort task at priority i is immune from stealing by any task at priority i or lower. The real question is the extent to which a high-level task should be allowed to steal from tasks at any given priority level. This question is relevant to our simple two-level scheme—should high-level realtime tasks be able to starve low-level tasks? Scout answers that question in the affirmative, but we can easily imagine other strategies. For instance, we could introduce a "middle" priority level, with its own queue. If we divide the combined rates of all mid-priority tasks by two when calculating cycles available for stealing, high priority tasks would be allowed to steal only half of the mid-priority bandwidth. We are currently exploring such alternative designs.

While generalizing BERT to support multiple priority levels has a certain appeal, our experience is that the current two-level scheme is sufficient for the kinds of multimedia applications we envision. In Scout, each window has a single on/off button that sets the application's priority level. Applications start in low priority by default; users select the button for important applications. (Users can later unselect the button if the application becomes less interesting.) It's not clear that users will make sense of a more sophisticated mechanism.

6.5 Sensitivity to Accurate Predictions

A path in Scout connects specific modules which perform the same function each time the path is executed. As a result, we are able to predict the duration of a path execution to a high degree of accuracy, even when the path's function is complex. For example, a path decoding MPEG video takes about 8 to 10ms to run on a 300MHz machine, but Scout can predict the runtime of the path to less than a millisecond over 90% of the time [1]. Our experience with BERT indicates that it is operates robustly with this level of error. Future research will try to quantify the sensitivity of BERT to prediction error; in the meantime, we can offer these remarks.

The first thing to note is, prediction error does not accumulate during the life of a path. Though the BERT algorithm increments the path's VC counter using its predicted runtime (steps 2a and 2b in Figure 6), after the path has run the VC can be adjusted by the difference between the actual and predicted times. When each new timestamp is assigned, the VC counter accurately reflects the path's CPU usage up until that time. In other words, the total accumulated error from all paths is no more than the sum of the errors from when each path last ran.

Prediction error affects realtime paths—it usually does not matter if a best effort path receives its share a few milliseconds late. The scenario that primarily concerns us is, a realtime path has stolen cycles to move its timestamp up to its deadline; due to errors in its own or other paths' prediction, it does not finish by its timestamp and so misses the deadline. However, realtime work can only miss deadline due to prediction error if someone's prediction was too small. If all paths give conservative predictions, no work will be delayed by error; the worst that can happen is, a path will run sooner than it should. Conservative predictions can hurt the path that makes them, though, because the path may not receive its full share due to the way that Virtual Clock works. Also, a high priority path may steal more cycles than it actually needs to make deadlines, impacting the low priority paths more than necessary. Ideally, all paths give accurate predictions; but failing this, erring on the conservative side will minimize the effects on other paths.

Finally, the granularity of the timestamp in Scout is about a millisecond, and the runtime of most paths is a few milliseconds or less. Therefore, the prediction error must be very large—50% or more—in order to result in a reordering of paths in the ready queue. Our predictions almost always are more accurate than this.

Overall, in our experience using BERT in Scout, we have observed few effects of prediction error. We believe there are two reasons for this: the locality of the effects of errors, and the accuracy of our path runtime predictions in Scout. We continue to study the effects of error, as well as to refine our predictions.

6.6 BERT in Other Systems

Although BERT was designed to take advantage of Scout paths, we believe it could be implemented in other operating systems as well. First, best effort tasks need not be represented as paths—they could instead be regular processes that are allowed to run for some time slice. The duration of this time slice would then correspond to $\mathsf{duration}(work)$ in the algorithm. Second, realtime tasks would need to behave more like Scout paths, but this would be possible without supporting the full path infrastructure. The keys would be (1) early demultiplexing, (2) preempting the process once per iteration (e.g., once per inputed packet or frame), and (3) predicting how long a given iteration will take. Finally, the OS interface would need to be extended to allow applications to specify reservations and priorities. As argued earlier in this section, though, a policy that sets reservations could be included with the scheduler; there is no reason for users to have to specify them directly.

7 Conclusion

BERT is a new scheduling algorithm for multimedia workstations that allows best effort tasks make progress and realtime tasks make their deadlines. Moreover, by allowing high-priority realtime tasks to steal cycles and high-priority best effort tasks to be immune from stealing, BERT gives the user the opportunity to discriminate between important and unimportant work. 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 (i.e., it decouples realtime from high-priority). 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] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] G. G. Xie and S. S. Lam. Delay guarantee of a virtual clock server. *ACM Transactions on Networking*, 3(6):683–689, Dec. 1995.

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