The Role of Virtual Time in Real-time Scheduling

Andy Bavier and Larry Peterson
Department of Computer Science
Princeton University

Tech Report TR-611-99

March 3, 2000

Abstract

The BERT scheduler is designed to schedule a mix of best effort and real-time processes. BERT is based on manipulating how tasks run in the fair queueing fluid model and tracking the changes using virtual time. This paper makes two contributions. First, it builds on results in real-time scheduling to show two properties of BERT: (1) important real-time tasks are schedulable with regard to their finish times in the fluid model, and (2) unimportant tasks have bounded workahead. BERT relies on these two properties when making scheduling decisions. Second, it draws attention to the thread that connects the fluid model and the BERT implementation: *virtual time*. Virtual time is a multifaceted and powerful abstraction, and we use our results for BERT to argue that it can form the basis for designing, implementing, and analyzing new real-time scheduling algorithms.

1 Introduction

The BERT scheduler is designed to schedule a mix of best effort and real-time processes [1]. BERT consists of two layers: (1) a theory-based core, and (2) a number of optimizations and approximations for a more efficient implementation. The work in [1] empirically demonstrates the real-time capabilities of BERT. In this paper, we strengthen the foundation of BERT by focusing on its theoretical core, which we call WFQ-Stealing.

The work presented here builds on the results of Figueira and Pasquale [6]. In that paper, the authors propose a general analytic framework for the *schedulability* problem of a real-time scheduling discipline—that is, can the scheduler run all tasks before their deadlines. We use their approach to analyze WFQ-Stealing.

This paper makes two contributions. The first is specific to BERT: we show schedulability results for WFQ-Stealing and bound its workahead. These results support our claims about the real-time behavior of the BERT algorithm. The second is more general: we try to convince the reader that there is more to the idea of *virtual time* than meets the eye. Virtual time is in fact a stepping-stone between a mathematical description of the system, and the real-time scheduling algorithm that approximates it. As such, it provides a framework for creating, implementing, and analyzing new real-time schedulers. The results we establish for WFQ-Stealing will provide one example of the power of virtual time.

The paper is laid out as follows. We begin by presenting background on Weighted Fair Queueing and WFQ-Stealing. Section 3 establishes a schedulability result for WFQ-Stealing, demonstrating its real-time behavior. Section 4 derives a bound on the amount of time a task in WFQ-Stealing can work ahead (i.e.,

receive service in advance of when it should according to the ideal system). The BERT algorithm relies on knowing this bound. Section 5 shows how we can add constraints to WFQ-Stealing to give it better real-time behavior. Section 6 ties all these results together, showing how to create a hybrid system that merges bounded workahead and better schedulability. The BERT algorithm runs in this hybrid system, and Section 7 argues that the enhancements which BERT adds to WFQ-Stealing preserve the system properties. Finally, Section 8 explores the role of virtual time in the creation and analysis of WFQ-Stealing.

2 Background

This section defines terminology used throughout the paper. The terms we use are similar to those found in [6], except they have been re-shaped to apply to scheduling the CPU rather than the network link. We also review Weighted Fair Queueing and WFQ-Stealing.

2.1 Definitions

Assume that we have a system with a CPU of rate R_{CPU}, running a set of processes which may change over time. Each process generates a sequence of *tasks*. The system assigns individual tasks (1) an eligibility time, and (2) an execution deadline. The eligibility time (also called a request time) is the earliest the task can start executing, while the deadline is the maximum allowable completion time for the task. The system runs an Earliest Deadline First (EDF) scheduler, which selects the piece of eligible task with the lowest deadline to run.

Tasks are numbered in the order that they are submitted—for example, the ith task belonging to process P would be $t_{i,P}$. Let $E_{i,P}$ be the execution time (i.e., in cycles) of task $t_{i,P}$. The eligibility time of this task is $S_{i,P}$ and the deadline $F_{i,P}$ (think: start and finish times).

The system is *preemptive* if a new task can be selected to run as soon as it becomes eligible. In a *nonpreemptive* system, tasks runs to completion and the system can schedule a new task only after the current one finishes.

The *eligible task sequence* is the sequence of tasks that become eligible for execution over time. The sequence is *schedulable* if the system can complete all tasks before their deadlines. The sequence is δ-*schedulable* if the system can guarantee that the amount by which any task's completion time exceeds its deadline is bounded by a constant δ; that is, if $\hat{F}_{i,P}$ is the *actual* finish time of a task $t_{i,P}$, then the system is δ-schedulable if $\hat{F}_{i,P} \leq F_{i,P} + \delta$ for all tasks. It may seem that a δ-schedulable system is not that useful since there is no guarantee that it can meet all task deadlines. However, note that such a system still has a real-time property, and later we will discuss how to build a real-time scheduler from such a system.

Table 1 summarizes these definitions, as well as additional notation that will be introduced in later sections.

2.2 Weighted Fair Queueing

Weighted Fair Queueing [4] (also known as Packetized General Processor Sharing (PGPS) [9]) is an algorithm originally proposed to allocate link bandwidth in a packet network; subsequent work extended WFQ to scheduling the CPU [7, 10]. The idea of WFQ is that each process P makes a reservation R_P of some slice of the CPU. As a process generates individual tasks, they are given *timestamps* based on the process's

R_{CPU}	rate of the CPU
R_P	reserved rate of process P
C_P	cycles received by process P
$t_{i,P}$	ith task generated by process P
$F_{i,P}$	execution deadline (i.e., fluid model finish time) of task $t_{i,P}$
$\hat{F}_{i,P}$	*actual* finish time of task $t_{i,P}$
$S_{i,P}$	eligibility time (i.e., fluid model start time) of task $t_{i,P}$
$E_{i,P}$	execution time of task $t_{i,P}$
$VF_{i,P}$	virtual finish time of $t_{i,P}$
$VS_{i,P}$	virtual start time of $t_{i,P}$
$D_{i,P}$	*actual* deadline/timing constraint of $t_{i,P}$
$W_{i,P}$	workahead of task $t_{i,P}$
ϵ	duration (in virtual time) of a bout of stealing
VR_P	the virtual restart time of process P (i.e., when stealing ends)

Table 1: Summary of Notation

reservation and its previous workload. WFQ then executes the tasks in order of increasing task timestamps. The result is that each process receives the CPU share it has reserved.

The behavior of WFQ is defined with reference to a fair queueing *fluid model* (i.e., GPS). In the fluid model, each process receives its reservation continuously, down to an infinitely fine level. A process may actually receive service faster than its reservation, because the fluid model distributes the reservations of idle processes to active ones in proportion to the latter's reservations (e.g., an active process with twice the reservation of another will get twice the unused cycles).

At the heart of WFQ is the concept of *virtual time*. If a process P makes a reservation of R_P, the virtual finish time of task $t_{i,P}$ is the real time it would finish if the task had a dedicated CPU of rate R_P. That is, virtual time on a real CPU can be thought of as real time on a virtual CPU. WFQ gives each task $t_{i,P}$ a timestamp of its virtual finish time $VF_{i,P}$; the ready queue is then ordered by these virtual timestamps. An important feature of virtual time is that executing tasks in order of virtual timestamps makes them finish in the same order as in the fluid model. One of our goals in this paper is to demonstrate the power of virtual time for real-time scheduling algorithms.

There can be discrepancies between the service that a process receives in the fluid model versus a WFQ system: at any point in time a process may have actually received slightly more or less service in WFQ than it has in the fluid model. In a preemptive WFQ system, all tasks complete by the time they would in the fluid model—i.e., it is schedulable according to fluid model deadlines (this follows from the discussion in [6]). A nonpreemptive WFQ system cannot guarantee that a task completes by its fluid model finish time; however, it has been shown that the system is δ-schedulable where δ is equal to the duration of the longest piece of work allowable in the system [9].

2.3 WFQ-Stealing and BERT

The BERT scheduler was proposed in [1] to schedule a mix of best effort and real-time processes. BERT groups processes along two dimensions: real-time vs. best effort, and important vs. unimportant. BERT

uses an innovation called *stealing* to take cycles from unimportant processes and give them to an important real-time process whose reservation is too small to meet a deadline.

WFQ-Stealing forms the theoretic core of the BERT scheduler. WFQ-Stealing was developed within the context of the fair queueing fluid model: stealing is defined as pausing one process in the fluid model and diverting the cycles it would have received to another process (this will be described in more detail in Section 3). WFQ-Stealing works by tracking the modifications in the fluid model using virtual time, and changing the virtual timestamps in the system accordingly. We will show that this approach is sufficient to guarantee that WFQ-Stealing has provable real-time properties.

BERT layers optimizations on top of WFQ-Stealing to make the implementation more efficient. First, BERT uses the WF^{2}Q+ scheduling algorithm to approximate WFQ, since WF^{2}Q+ is much simpler. Second, BERT implements optimizations to allow an important real-time process to steal fairly and efficiently from *all* unimportant processes at once. Section 7 lists the conditions met by these enhancements which ensure that our analysis of WFQ-Stealing remains valid for BERT.

As originally described, the BERT scheduler is nonpreemptive. However, there is no requirement that this be the case. In this paper we will analyze nonpreemptive WFQ-Stealing, but a similar analysis of a preemptive version is straightforward using the results of [6].

3 Schedulability

This section establishes a schedulability result for a WFQ-Stealing scheduler. The intuition behind our result is simple: WFQ is much like EDF, and WFQ-Stealing is much like WFQ except with slightly different deadlines. Therefore, we can analyze WFQ-Stealing from the standpoint of EDF.

Figueira and Pasquale establish a very powerful result in [6]: If the eligible task sequence is schedulable under some (preemptive or nonpreemptive) policy, then it is δ-schedulable under nonpreemptive deadline-oriented scheduling. The term "deadline-ordered" means that the system executes the eligible task set Earliest Deadline First (EDF). In this result, the value of δ is the longest task run-time allowable in the system (i.e., the longest interval between scheduling decisions). What this means is that if there is some way to schedule the eligible task set, then nonpreemptive EDF will finish each task no later than the maximum task run-time after its deadline.

Of course, in this system it is possible for tasks to miss their deadlines. However, it is important to note that this system still exhibits quantifiable real-time behavior. Furthermore we can establish the δ-schedulability property for *any* scheduler if we can show two things:

1. There exists a way to schedule eligible tasks to meet all deadlines.
2. The eligible task set is executed using nonpreemptive EDF.

The rest of this section demonstrates that WFQ-Stealing satisfies these two conditions, and is therefore δ-schedulable.

3.1 Mathematical Foundation

In Section 2 we summarized the fair queueing fluid model and virtual time. The two are intertwined, and the manner in which they are related is important for establishing the δ-schedulability of WFQ-Stealing. Next we define them mathematically in order to tease out this relationship.

First consider the definition of how a task receives service in the fluid model. Let R_P be the reserved rate of process P, and $t_{i,P}$ be a task generated by the process. Also, let A be the set of all active processes, that is, all those processes with outstanding tasks. If C_P is the total amount of cycles received by process P so far, then the rate at which its current task runs is:

$$\frac{dC_P}{dt} = \frac{R_P}{\sum_{q \in A} R_q} R_{CPU} \tag{1}$$

This simply states that a task gets a rate proportional to its reserved rate divided by the rates of all active tasks. Since the fluid model assumes that $\sum_{q \in A} R_q \leq R_{CPU}$ (i.e., the sum of all reservations is less than the CPU rate), task $t_{i,P}$ is guaranteed to receive a rate of at least R_P.

The actual rate received by a task in the fluid model can vary, depending on the set of active processes. One advantage of virtual time is that it can simplify the fluid model. With respect to virtual time, task $t_{i,P}$'s rate is always equal to R_P. This means that virtual time itself speeds up and slows down as the set of active processes changes. Let v be the current virtual time; v changes at the rate:

$$\frac{dv}{dt} = \frac{R_{CPU}}{\sum_{q \in A} R_q} \tag{2}$$

Since $\sum_{q \in A} R_q \leq R_{CPU}$, this means that $dv/dt \geq 1$; in other words, virtual time flows at least as quickly as real time. Also, combining Eqs. 1 and 2, it is easy to see that $dC_P/dv = R_P$. Since task $t_{i,P}$ receives service at a constant virtual rate, we know the virtual time at which it will finish. If $E_{i,P}$ is the execution time of the task and it begins to run in the fluid model at virtual time $VS_{i,P}$, then the virtual finish time of the task, $VF_{i,P}$, is:

$$VF_{i,P} = VS_{i,P} + \frac{E_{i,P}}{R_P} \tag{3}$$

3.2 WFQ-Stealing Analysis

In [6], the authors reproduce the well-known result that WFQ (called PGPS in [6]) is δ-schedulable using their analytic framework. We use a similar approach to show the same result for WFQ-Stealing.

WFQ-Stealing, like WFQ, assigns a virtual timestamp to each task representing its virtual finish time in the fluid model. It then executes the tasks EDF. The difference is in the timestamps assigned to tasks. WFQ-Stealing modifies the timestamps of WFQ to give some tasks more cycles than they have reserved in order to meet their real-time deadlines. However, the critical point is that both WFQ-Stealing and WFQ are tracking the flow of service in the fluid model using virtual time. Next we describe WFQ-Stealing mathematically, and then we show that simulating the fluid model with virtual timestamps is sufficient to establish the δ-schedulability of WFQ-Stealing.

WFQ-Stealing is defined in the fluid model itself. Consider two processes, a high-priority process H and a low-priority one L. Normally, the processes run as in the WFQ fluid model, according to their reserved rates. When process H steals from process L, the cycles that L would receive in the fluid model are diverted to process H. If L was idle when the stealing began, L is considered active (i.e., $L \in A$) for the duration of stealing. Formally, when H is stealing from L,

$$\frac{dC_L}{dt} = 0$$

$$\frac{dC_H}{dt} = \frac{R_H + R_L}{\sum_{q \in A} R_q} R_{CPU}$$

Stealing changes the way that cycles are distributed in the fluid model, which in turn affects the finish times of tasks. In order for WFQ-Stealing to track the fluid model, it must know the new virtual finishing times of tasks belonging to H and L. First we consider the task belonging to H. Applying the definition of virtual time, we see that during stealing:

$$\frac{dC_H}{dv} = R_H + R_L$$

During stealing, process H receives a constant virtual rate equal to the combined rates of both H and L. If process H steals from process L for a period of ϵ units of virtual time, it can be calculated that task $t_{i,H}$ has the new virtual finish time $VF'_{i,H}$:

$$VF'_{i,H} = VS_{i,H} + \frac{E_{i,H}}{R_H} - \frac{\epsilon R_L}{R_H} \tag{4}$$

That is, the effect of stealing for a virtual time period of ϵ is to decrease the virtual finish time of the task by $\epsilon R_L / R_H$. Since virtual time flows faster than real time, this means the real finish time in the fluid model decreases by at least this much; the BERT algorithm relies on this behavior.

Next we turn attention to process L. Assume L is active and a task $t_{j,L}$ is running when stealing starts. The task is paused for ϵ units of virtual time, and then continues to run at its normal rate. It is easy to see that:

$$VF'_{j,L} = VS_{j,L} + \frac{E_{j,L}}{R_L} + \epsilon \tag{5}$$

Eqs. 4 and 5 summarize the results presented in [1]. That work also deals with special cases, such as L is idle but submits a task during ϵ, which we omit here.

WFQ-Stealing's fluid model is more complex than that of WFQ. However, the fact that we can calculate virtual finishing times for all work is enough to establish that a nonpreemptive WFQ-Stealing scheduler is δ-schedulable. To explain why, we note two facts. First, the fluid model represents a way to schedule all tasks to meet a certain set of deadlines, namely the task finish times in the fluid model. Second, ordering tasks by virtual finish times is equivalent to ordering them by their real finish times in the fluid model (this follows from $dv/dt > 0$). Therefore, nonpreemptive WFQ-Stealing is a deadline-oriented service discipline with a schedulability test, and so it is δ-schedulable with regard to fluid model finish times. Figueira and Pasquale use a similar argument for WFQ in [6]. In fact, the above reasoning applies to *any* scheduler that assigns virtual timestamps to track a fluid model (this is also implied in [6]). This is a powerful result for designing new real-time scheduling algorithms, as we will discuss later.

4 Bounding Workahead

The previous section demonstrated that a task in a system using WFQ-Stealing will finish no later than δ after its finish time in the fluid model. However, work may actually finish before its fluid model finish time too. In the WFQ literature, this is referred to as *workahead*—the task receives a chunk of service in the WFQ system before it would be received in the fluid model (see [3] for a full discussion of the differences

between the fluid model and WFQ). Workahead presents a small problem for WFQ-Stealing, since it is not possible to steal cycles from a process that has already used them. We briefly discuss workahead next.

WFQ-Stealing takes cycles from process L and gives them to process H, thereby allowing a task belonging to H to meet a real-time deadline. If the stealing interval lasts for ϵ ticks of virtual time, the result is that ϵR_L cycles are stolen from L. However, it may be the case that other processes are stealing from L. We account for the cumulative effects of stealing with the concept of the *virtual restart time*, which is the virtual time at which a process once more receives cycles in the fluid model. Let VR_L be the virtual restart time of L. In order for stealing to work, it must be the case that, if the current virtual time in the fluid model is v, then:

$$VF'_{i,H} \geq max(v, VR_L) + \epsilon \tag{6}$$

This means that process L must have at least ϵ of virtual run-time available to it before the virtual finish time of task $t_{i,H}$. WFQ-Stealing performs this test before stealing from a task.

The variable VR_L above represents one source of cycles unavailable for stealing—those that have already been stolen by other tasks. Workahead represents another. Workahead is different in that it represents a discrepancy between the fluid model and the algorithm that tracks it. In the fluid model, tasks receive their reservations over any interval, no matter how small. In nonpreemptive WFQ-Stealing, one task runs—has exclusive use of the CPU—and then another, and so on. So at any time, the amount of cycles that a task has received in the fluid model may be more or less than in the WFQ-Stealing discipline.

We can think of working ahead as "borrowing" cycles (in contrast to stealing them). When a task is running, it receives an effective rate of R_{CPU}; when the task is sitting on the ready queue, its effective rate is 0. Therefore, a process borrows cycles belonging to other processes when one of its tasks is using the CPU; when it is not using the CPU other tasks borrow from it. In the long haul, it all evens out.

WFQ-Stealing must deal with workahead since it is not possible to steal these borrowed cycles. One approach is to look at the actual workahead of the task to be stolen from when determining whether or not to steal. Another is to *bound* the possible workahead of the task, and then to account for these cycles in the test of Eq. 6; that is, not steal cycles that might be "owed" to other tasks. BERT takes this latter approach.

More formally, workahead is defined as the difference between the virtual time when a task actually completes and its virtual finish time in the fluid model. So for task $t_{i,P}$, its workahead $W_{i,P}$ when it completes at virtual time v' would be:

$$W_{i,P} = VF_{i,P} - v'$$

Recall that the Figueira and Pasquale result referred to the *eligible* task sequence; the key to bounding workahead is incorporating an eligibility test for each task based on when it starts to run in the fluid model. Let $VS_{i,P}$ be the virtual start time of task $t_{i,P}$. At virtual time v, the task is eligible to run if $VS_{i,P} \leq v$; this simply means that the task is running in the fluid model.

The workahead bound falls out directly from the mathematical definitions of workahead and eligibility. A task $t_{i,P}$ cannot start to run until it is eligible, so the virtual time v' that it can finish is:

$$v' \geq VS_{i,P} + E_{i,P}/R_{CPU}$$

Substituting into the definition of workahead,

$$W_{i,P} \leq VF_{i,P} - VS_{i,P} - E_{i,P}/R_{CPU} \tag{7}$$

Stealing from a task changes its virtual finish time, as we saw in Eq. 5. If task $t_{i,P}$ has been paused for ϵ virtual time units, substituting for $VF_{i,P} - VS_{i,P}$ in the above, we get:

$$W_{i,P} \leq E_{i,P}/R_P - E_{i,P}/R_{CPU} + \epsilon$$

$$W_{i,P} \leq E_{i,P}(1 - R_P/R_{CPU}) + \epsilon$$

The possible workahead is related to the task run time. So, we can bound the maximum possible workahead for a process if we know the maximum task run-time for that process. If $E_{max,P}$ is the maximum run time of a task for process P, then the bound $W_{max,P}$ becomes:

$$W_{max,P} \leq E_{max,P}(1 - R_P/R_{CPU}) + \epsilon$$

We have bounded the workahead for WFQ-Stealing—in fact, it is the known workahead for WFQ [3] with the addition of ϵ. This result is not surprising, since the bound follows simply from the mathematical definitions of workahead and eligibility. We wanted to quantify the workahead of a task to ensure that we did not try to steal unavailable cycles. The ϵ represents time stolen from the task, and our check of Eq. 6 already subtracts this from the available capacity (using the virtual restart time). Therefore, we only need to account for the WFQ workahead; this means that, before stealing cycles from H to give to L, WFQ-Stealing must verify that:

$$VF'_{i,H} \geq max(v, VR_L) + E_{max,L}(1 - R_L/R_{CPU}) + \epsilon \tag{8}$$

This test is in fact used by the BERT algorithm when determining the amount of cycles that are available for stealing. It is valid because of the workahead bound for WFQ-Stealing established above.

5 Doing Better than δ-Schedulable

Section 3 established that a nonpreemptive WFQ-Stealing system is δ-schedulable. In this section we present our motivation for getting rid of δ—making the system strictly schedulable—and then explain how to do so.

We use WFQ-Stealing as the foundation of BERT, a real-time scheduler. To explain how, we distinguish between three different quantities for a task $t_{i,P}$: its fluid model finish time, $F_{i,P}$; the virtual finish time of the task, $VF_{i,P}$; and its actual deadline, $D_{i,P}$. The virtual timestamp determines the order of execution of tasks, and the task may finish up to δ after $F_{i,P}$. However, as long as the task finishes execution before $D_{i,P}$, it meets its deadline.

The fact that WFQ-Stealing is δ-schedulable is annoying in one respect: the δ. For a particular task, we can place an upper bound on its finish time in the fluid model. In order for the task to be assured of meeting its deadline in WFQ-Stealing, it must be the case that its actual deadline is at least δ *after* its maximum fluid model finish time (i.e., $D_{i,P} \geq F_{i,P} + \delta$). This means that the process may have to make a conservative reservation, but this can lead to underutilization of the system.

For example, consider a system with a 1GHz CPU in which the longest that a task can run is $\delta = 10ms$. Suppose a process in this system generates tasks of run-time 10 million cycles with deadlines $33ms$ apart, and suppose each task becomes eligible $33ms$ before its deadline. This process uses 300 million cycles per second (Mcps), so assume that the process makes this reservation. Based on the process reservation and δ, each task may take up to $43ms$ to complete after it becomes eligible (i.e., $33ms + \delta$), and thus can miss

its deadline. However, the process can get the service it needs if it makes a reservation of 435Mcps. This corresponds to a reserved rate of 10 million cycles every $23ms$, and so a task of $10ms$ duration will finish within $33ms$ ($23ms + \delta$) after it becomes eligible. Therefore, a process in a δ-schedulable system can still get the service it requires at a fine granularity through making a conservative reservation, but we want to avoid this if we can.

To allow better system utilization, we would like to set $\delta = 0$. Figueira and Pasquale provide a result that we can apply here too: in a nonpreemptive system where it is *not possible* for a task to become eligible while another one is running, the system is strictly schedulable. An example of such a system would be one in which (1) new tasks can enter the system only at a scheduling point between tasks, and (2) all tasks become eligible upon entering the system. Note that the original WFQ algorithm already conforms to condition (2), and condition (1) simply changes the arrival times of tasks. The lack of δ in the schedulability result for such a system does not mean that tasks finish earlier in this constrained system, it simply means that there is a closer correlation between fluid model and actual finish times.

Referring back to our example, in the constrained system the process can make a reservation of 300Mcps and meet all task deadlines as long as the tasks are generated at scheduling points. Chances are that the process does this already; that is, the process generates the next task when the last one completes. In this system, all tasks finish by their fluid model finish times, and there is no need to make a conservative reservation.

6 A Hybrid System

BERT schedules both real-time and best effort processes, and each process can be either important or unimportant. Important real-time processes are allowed to steal from unimportant processes in order to meet their real-time deadlines. This means that our main concern with the important real-time processes is schedulability (i.e., getting rid of δ) while for the unimportant processes it is limiting workahead (so we can steal from them).

In Section 4, we saw that we could limit the workahead of WFQ-Stealing by introducing an eligibility test. Section 5 discussed how we could constrain the system to make it strictly schedulable. In this section, we describe a hybrid system, one in which *some* processes (i.e., important real-time) are strictly schedulable, while *others* (i.e., unimportant processes) have their workahead limited. The BERT scheduler runs in this hybrid system.

The idea here is simple: we start with the constrained system of Section 5. To it, we add the eligibility test of Section 4, but only for the unimportant tasks. To demonstrate that this does what we want, we need to establish three points:

1. The eligible task sequence is schedulable
2. The workahead of the unimportant real-time work is bounded
3. The important real-time work strictly is schedulable

We argue each point in turn.

First, we have seen that the fluid model provides a schedulability test, showing how it is possible to schedule the eligible task sequence to meet deadlines. Since we define "deadlines" in this case to mean task finish times in the fluid model, this is almost trivially true. The only instance in which it would *not* be true

is if an *ineligible* task could run in the fluid model. By definition, a task is eligible when it has started to run in the fluid model and so this cannot happen.

Second, Section 4 demonstrates that the workahead bound of a task follows directly from the mathematical definitions of workahead and eligibility. The intuition is that if a task cannot be scheduled until the virtual time reaches some value, then it cannot work ahead more than a certain amount. This behavior is independent of other tasks. Incidentally, the tasks to which we apply the schedulability test are only δ-schedulable, since they can become eligible when another task is running.

Third, the schedulability of the tasks for which the eligibility test is *not* performed is not affected by performing the test for other tasks. The effect of performing the eligibility test on a task is simply that this task may execute *later* than it would if it had been instantly eligible. This means that a task which is always eligible can only execute *earlier* if other tasks are tested for eligibility. Since the system in which *all* tasks are immediately eligible is strictly schedulable, then so are the important real-time tasks in the hybrid system.

7 BERT Optimizations

At this point, we have a theoretical foundation for BERT. A nonpreemptive WFQ-Stealing system in which tasks are admitted only at scheduling points, and eligibility testing is done for unimportant tasks, has two provable properties: (1) the tasks for which we perform the eligibility test have limited workahead, and (2) the others actually finish by the times they would in the fluid model. BERT assumes that these two properties are true when it makes a decision to steal. However, as mentioned in Section 2, BERT contains some enhancements to make WFQ-Stealing more efficient. As a final step, we argue that the differences between WFQ-Stealing and the BERT algorithm preserve these system properties.

First, BERT does not implement WFQ, but rather uses the WF^{2}Q+ scheduling algorithm [2] to approximate it. The main difference between the two algorithms is that WF^{2}Q+ estimates the virtual time of the fluid model, whereas WFQ actually simulates the fluid model and so knows the exact virtual time. WF^{2}Q+ is much less complicated to implement than WFQ, and has less computational overhead. As shown in [2], however, WF^{2}Q+ has the same δ-schedulability and workahead bound as WFQ. Most important with regard to our analysis of WFQ-Stealing, WF^{2}Q+ is defined such that $dv/dt \geq 1$, yet the virtual time estimated by WF^{2}Q+ is never greater than the actual WFQ virtual time. WF^{2}Q+ has similar properties to WFQ and so can be substituted with no ill results.

Second, BERT's optimizations allow an important real-time process to steal from a *set* of unimportant processes without having to change the timestamps of each individual task. The optimizations work by associating a set of variables with the group of unimportant processes; stealing changes the values of these variables. The actual timestamps attached to unimportant tasks are not changed by stealing, but the new virtual finish time of an unimportant task can be calculated from its timestamp and the new variables. The point is that these optimizations maintain the same virtual finish times for tasks as WFQ-Stealing does. They allow BERT to track the fluid model in the same way as WFQ-Stealing, and so the results we have established for WFQ-Stealing apply to BERT.

8 The Power of Virtual Time

What *is* virtual time? Is it (1) an implementation device, (2) a mathematical abstraction, or (3) something more fundamental? The answer is all of the above. This section uses our analysis of WFQ-Stealing to illustrate the power of virtual time for the design, implementation, and analysis of new real-time scheduling algorithms.

As mentioned in Section 2, the idea of virtual time is closely tied to the abstract representation of a resource. Consider a process P with a reservation R_P. The virtual finishing time of its task $t_{i,P}$ is the *real* time at which the task would finish if the process had sole possession of a CPU of rate R_P. That is, the flow of virtual time for the process corresponds to real time on a virtual resource. Now here is a key insight: when stealing from process P, WFQ-Stealing *multiplexes* other processes onto P's virtual CPU. Once we have removed process P's exclusive access to the virtual CPU, the possibilities for multiplexing processes onto it are vast—stealing to meet important real-time deadlines, as BERT does, represents only one. The point is, virtual time represents a virtual resource that we can manipulate.

The fluid model gives us a context for reasoning about exactly *how* this manipulation is to take place. In it, we can mathematically describe how service should be redistributed, and which tasks are affected. Changes in the fluid model then ripple over to influence the virtual finish times of tasks. The fluid model allows us to describe how we want to multiplex processes, and virtual time lets us quantify the effects.

Virtual time also gives us the ability to implement the system that we have mathematically described via the fluid model. We have shown in Section 3 that ordering tasks by virtual finishing times is equivalent to assigning each task its real finish time in the fluid model as a deadline, and then running the system EDF. So virtual time is the key to making a real system that approximates the mathematical model in its behavior. As other sections show, virtual time is also useful for quantifying the differences between the real system and the fluid model representation.

Finally, virtual time can be a tool in analyzing the behavior of other proposed algorithms—for example, the SMART [8] and Borrowed-Virtual-Time [5] schedulers. Both of these algorithms are based on virtual time and involve mechanisms that change the timestamps of tasks. As should be clear from our analysis of WFQ-Stealing, such manipulations affect the fluid model, which in turn can influence the resulting algorithm's ability to meet real-time deadlines. Elaboration of this analytical framework is intended for future work.

In summary, virtual time is a multifaceted and powerful abstraction. It allows us to represent a virtual resource that we can manipulate in any number of ways. Together with the fluid model, it gives us the ability to describe how we want to use the resource and to quantify the effects. Then it provides the basis for constructing a real-time algorithm from this description. Finally, it is an analytic tool with which we can figure out what some complex scheduling algorithms actually do.

9 Conclusions

This paper has shown two results specific to the BERT scheduler. First, a nonpreemptive WFQ-Stealing scheduler is δ-schedulable with respect to tasks finish times in the fluid model. This makes it a strong foundation for building a real-time scheduler such as BERT. Second, it is possible to create a system in which some (important real-time) processes are strictly schedulable, while other (unimportant) processes have bounded workahead. BERT runs in such a system, and uses these schedulability and workahead results

when stealing from tasks to meet real-time deadlines.

However, the results of this paper reach beyond BERT. Our analysis of WFQ-Stealing has demonstrated the relationship between the fluid model, virtual time, and implementing a real-time algorithm with provable schedulability. It is conceivable that this relationship can anchor a methodical approach to designing real-time scheduling algorithms such as BERT, as well as provide an analytic framework for understanding some current virtual-time-based algorithms. We hope to expand on both method and framework in future work.

References

[1] A. Bavier, L. Peterson, and D. Mosberger. BERT: A scheduler for best effort and realtime tasks. Technical Report TR-602-99, Department of Computer Science, Princeton University, Mar. 1999.

[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] J. C. R. Bennett and H. Zhang. WF^2Q: worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM'96*, pages 120–128, San Francisco, CA, Mar. 1996.

[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] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, Dec. 1999.

[6] N. R. Figueira and J. Paquale. A schedulability condition for deadline-ordered service disciplines. *ACM Transactions on Networking*, 5(2):232–244, Apr. 1997.

[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] 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.

[9] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *ACM Transactions on Networking*, 1(3):344–357, June 1993.

[10] 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.