The Power of Virtual Time for Multimedia Scheduling

Andy Bavier and Larry Peterson
Department of Computer Science, Princeton University
Princeton, NJ 08544
{acb,llp}@cs.princeton.edu

Abstract

Many multimedia scheduling algorithms implement fair sharing of the CPU among processes. However, often a share of the CPU does not adequately satisfy the timing constraints of applications such as MPEG video. Several schedulers have been proposed to address this problem; each provides CPU shares but also features innovative uses of virtual time to better support multimedia applications. To give the reader a flavor of the work in this area, we first compare the mechanisms by which the SMART, BERT, and BVT algorithms provide better multimedia performance. Second, and more significantly, we propose a design methodology for producing multimedia schedulers with provable real-time behaviors using virtual time. Virtual time abstracts critical information from a complex mathematical description of the ideal system. This information is then used to schedule tasks so that the system conforms to its ideal description in real time. Virtual time is a bridge between theory and code, and this is its power.

1 Introduction

Many multimedia scheduling algorithms implement *fair sharing* of the CPU among processes. [1] Recently, several scheduling algorithms have been proposed which share a common premise: while fair sharing produces desirable overall system behavior—namely, guaranteed execution rates and isolation between processes—it does not adequately satisfy the timing constraints of real multimedia applications like MPEG video decoding. Each of these new schedulers begins with an algorithm based on *virtual time* to provide fair sharing of the CPU, and then tweaks the algorithm so that it better supports multimedia applications. In each case, the result is a system that sometimes diverges from strict fair sharing, but in which multimedia applications enjoy improved performance. In this paper, we will consider three such multimedia scheduling algorithms: SMART [9], BERT [2], and BVT [6].

[1] Also referred to as *proportional sharing*.

Our paper is written in two parts. First, we summarize the goals of each algorithm and describe how it uses virtual time to meet them. In the process, we hope to convince the reader that virtual time is a powerful and flexible abstraction for multimedia scheduling. Our intention is not to perform a comprehensive survey of scheduling algorithms, but rather to provide some perspective on this area and to motivate the problem of designing novel virtual time schedulers. Second, we propose a design methodology for creating new multimedia scheduling algorithms; this is the main contribution of the paper. Algorithms produced by our methodology *manipulate virtual time* in order to diverge from fair sharing in controlled and quantifiable ways. Designers following our method begin with a mathematical description of the system (based on modifying the fair queueing fluid model), apply a simplifying abstraction (virtual time), and then implement a timestamp-based scheduling algorithm that conforms to the mathematical description in real time. We believe this framework gives rise to a family of interesting and powerful multimedia scheduling algorithms.

2 Background

First, we introduce some language that we will use in the rest of this paper. A *process* is an application in the system. Each process reserves a *share* or slice of the CPU expressed in absolute terms, for instance some number of cycles per second; the period of the reservation is insignificant, as we will explain later. A process generates a sequence of *tasks*, which represent individual chunks of work of known duration (e.g., a timeslice, or the amount of cycles required to decode a particular video frame). [2] A task may or may not have a *deadline* that represents a soft timing constraint on the completion of the task. Each process has at most one task on the ready queue at any time. For simplicity, we assume that there are no synchronization issues between tasks.

Multimedia applications such as video and audio decoders are often called soft real-time. Scheduling soft real-

[2] This definition of "task", though awkward for those who equate tasks and processes, is standard in the real-time community.

time applications presents some challenging problems, perhaps as difficult as those facing hard real-time schedulers. Soft real-time applications are tolerant of missing some timing constraints, but often exhibit less deterministic behavior than hard real-time applications. For example, an MPEG video decoder requires a variable amount of cycles to meet a frame's deadline, depending on whether it is an I, P, or B frame. Moreover, the longer-term processing requirements of the decoder may change too as scenes change in the video; more detail or action typically requires more cycles to decode. Add to this the fact that soft real-time applications are expected to run on overloaded systems and even to adapt their behaviors to use fewer resources. The combination of overload and dynamic unpredictability gives multimedia scheduling its own unique flavor.

The benefits of CPU fair sharing for multimedia scheduling are widely recognized. The essential characteristics of a fair sharing scheduling algorithm are:

- Each process reserves a cycle rate—e.g., 1 million cycles-per-second (Mcps)—and is guaranteed to receive at least this rate when it has work to do.
- Unused and unallocated capacity is fairly distributed to active processes in proportion to each process's reservation. An active process that receives extra cycles beyond its reservation is not charged for them.
- An idle process cannot "save credits" to use when it becomes active. Unused share is simply lost.
- The guarantees made to processes provide *isolation* between them—each process gets its rate no matter what any other process does.

The guarantees and isolation of fair sharing can lead to desirable behaviors in multimedia systems. For the most part, a share of the CPU is what an MPEG video decoder wants—typically it requires roughly the same number of cycles to decode its 30 frames each second. Fair sharing isolates processes so that if the requirements of the MPEG video decoder increase, it will receive extra cycles only if it can do so without impacting another process. If the cycles are not available (i.e., the CPU is overloaded), then the user has the opportunity to redistribute shares to applications in order to achieve the overall system behavior that she wants. This combination of power and flexibility is not available with real-time scheduling algorithms such as EDF or rate monotonic.

Fair sharing has its drawbacks for scheduling multimedia applications as well. An MPEG decoder process can require five times as many cycles to decode an I frame as a B frame [1]. If the video makes a reservation based on its peak requirement, such a conservative share may force the admission controller to reject processes even though there are usually plenty of cycles available. Conversely, if the video reserves its average rate, it may miss deadlines on expensive I and P frames. These frames cannot be discarded and must be decoded in a timely fashion, because later frames reference them. The system can delay playback to smooth the variability in frame processing requirements (it closely resembles network jitter), but this can introduce too much latency for interactive video. In summary, fair sharing is very good for scheduling multimedia, but not perfect.

Many fair sharing algorithms have the concept of *virtual time* at their heart [4, 5, 10, 11]. Algorithms that implement fair sharing using virtual time are based on a theoretical fluid model that describes the ideal behavior of the system. From this model, an algorithm is constructed that schedules individual tasks. The key to tying together the theoretical model and the actual implementation is virtual time; it provides the means of creating a scheduler to run the system in a way that conforms to the ideal model. Next we give some background on the fluid model, a fair sharing algorithm implementation, and virtual time.

The fair queueing fluid model [10] provides a mathematical description of the instantaneous execution rates of processes, based on their reserved shares, in an ideal system. One way to think of the model is as a cycle-by-cycle weighted round robin scheduling of processes. This means that over any interval, no matter how small, a process with N times the share of another receives exactly N times the cycles. The fluid model gives us a powerful tool for describing fair sharing. In the fluid model, a running process always receives at least its reserved share; since the fluid model is work-conserving, the process additionally receives a proportion of the excess system share due to unreserved capacity or idle processes. Therefore, the fluid model describes the ideal real-time behavior of a fair sharing system.

Weighted Fair Queueing (WFQ) [5] is the canonical implementation of a fair sharing algorithm using virtual time. The goal of WFQ is to approximate the behavior described by the fluid model in the running system. WFQ assigns *virtual timestamps* to the individual tasks of each process, based on the process's reserved CPU share and previous activity, and then executes the task set in order of increasing timestamps. The result is a system in which processes fairly share the CPU within certain limits. Algorithms created by our scheduler design framework of Section 4 operate in just this way. However, as we aim to show, virtual time is much more than simply a mechanism for implementing fair sharing.

In a system running WFQ, over any interval of time, a process may actually receive slightly more or less service in WFQ than it does in the ideal fluid model. The service that a process receives in the real system before it is due in the model is called *lead*, and service received after it is due is known as *lag*. [3] A powerful result that has been established for WFQ is that its lag is bounded—by zero for a preemptive scheduler, and by the duration of the longest allowable task for a nonpreemptive one. The same result has been shown for many other virtual-time-based fair sharing algorithms. Its importance is that it quantifies how a process receives its share in real time, making virtual time algorithms attractive for real-time scheduling.

Finally, virtual time is the abstraction that ties together the ideal fluid model description of the system and the algorithm that approximates it. The intuition behind virtual time is that it represents a virtual resource. Consider a process P that reserves a rate R on the CPU. As the process submits work, each piece of work receives a virtual timestamp which represents its *actual* finish time if the process had a dedicated CPU of rate R. Recall that one feature of a fair sharing algorithm is that an idle process loses its share and cannot save it to use later. If we think of a process's share as a virtual CPU, this makes sense—leaving your CPU idle for a while does not make it faster. We will define virtual time formally in Section 4.2, and show how it abstracts information critical for real-time scheduling from the fluid model. Fully explaining the relationship between the fluid model, virtual time, and the real-time behavior of the resulting scheduling algorithm is one of the main contributions of this paper.

3 Comparison

In this section we compare three multimedia schedulers: SMART, BERT, and BVT. Each of these algorithms uses virtual time to provide processes with CPU shares, yet each departs from strict fair sharing in an innovative way. The discussion has a dual purpose. The first is to highlight how the three algorithms use virtual time, in order to give a flavor of the approaches that have been tried. The second is, through examining the successes and limitations of the algorithms, to motivate the problem: how do you design a scheduler to provide particular kinds of service in real time to multimedia applications?

3.1 SMART

SMART (a Scheduler for Multimedia And Real-Time applications) was historically the first of the three schedulers, and no doubt it influenced the design of the other two. The SMART scheduler provides CPU shares to processes while striving to meet as many real-time deadlines as possible in a general-purpose OS. SMART stems from the observation that, while a CPU share is fine for most conventional processes, it is not exactly what a multimedia process like an MPEG video decoder needs from the system; the decoder would like to have the system satisfy its deadlines, not simply give it a share. The goal of SMART is to satisfy the deadlines of multimedia processes in the general context of fair sharing.

SMART contains a rich mixture of features, but we will focus on one in particular: the Earliest Deadline First (EDF) reordering of a set of real-time tasks. All processes in SMART have both a priority and a share, and individual real-time tasks also have deadlines. Individual tasks are assigned virtual timestamps based on the process share in the usual way. A task's priority (from its process) and virtual timestamp together form a *value tuple* which is used to rank tasks; the task with the higher priority is said to have a higher value tuple, and at the same priority the task with the lower timestamp has a higher tuple. The core of the SMART task selection algorithm is as follows:

1. If the task with the highest value tuple is a conventional task, run that.

2. If the task with the highest value tuple is a real-time task, create a candidate set of all real-time tasks with a higher value tuple than the highest-ranked conventional task.

3. From the candidate set, create a work schedule as follows. In order of decreasing value tuples, insert each task into an EDF schedule only if doing so preserves the feasibility of the schedule. That is, test to make sure that inserting a task into the schedule will not cause a previously inserted tasks to miss its deadline.

4. Execute the resulting work schedule EDF.

For our purposes, the most interesting feature of SMART is the EDF reordering of the candidate set at steps (3) and (4)—this is SMART's mechanism for better supporting multimedia applications in the context of fair sharing. [4] SMART performs this reordering because EDF is an optimal scheduling algorithm: it will meet all deadlines if it is possible to meet them. In order to make sure that it *is* possible, SMART performs a *feasibility test* prior to inserting each task from the candidate set into the work schedule. A successful insertion means that the task is guaranteed to meet its deadline regardless of its share or timestamp.

SMART inserts tasks into the work schedule in order of decreasing value tuple, which among tasks of the same priority means in order of increasing virtual timestamps. When a task T is inserted into the schedule, only tasks with higher

[3] Lag can also be described in relation to a process's virtual rate; our usage is slightly stronger, as we will show later.

[4] A multimedia process could be run at high priority, but that provides no isolation between it and most other processes. For this reason, the designers of SMART envisioned that high priority would be used sparingly.

value tuples have already been inserted—these tasks were ahead of T in the ready queue to begin with. Therefore, task T is given an opportunity to move up in the queue if it can do so without consequences; likewise, lower-ranking tasks are permitted to cut in front of T only if T will still meet its deadline. Note that if T cannot be inserted, then it would not have met its deadline without reordering, because it would have executed after all of the tasks already in the work schedule. The important point is that no task in the candidate set is worse off than if reordering had not been done.

The use of virtual timestamps in the creation of an EDF work schedule allows SMART to improve on the separation between processes provided by fair sharing. Conventional tasks are not affected, while real-time tasks stand an equal or better chance of meeting their deadlines. Though it does not explicitly manipulate virtual time, SMART uses it creatively as a tool in multimedia scheduling.

3.2 BERT

The BERT (Best Effort and Real-Time) scheduler is designed to schedule a mix of real-time and best effort (i.e., conventional) processes in Scout, a communication-oriented OS. BERT's focus is on producing good system behavior despite the problems of overload and changing application requirements that are widespread in multimedia systems. Like SMART, BERT reasons that a best effort process wants a CPU share; on the other hand, a multimedia process would like its deadlines met, regardless of the share needed to do so. The system should try to satisfy the requirements of both kinds of processes. When these requirements conflict, as they are bound to do in an overloaded system, the importance of processes to the user should be used to resolve conflicts in an intuitive way. The idea that the user should explicitly yet easily specify how the system distributes resources is central to BERT.

BERT comprises a virtual-time-based scheduling algorithm, a simple policy framework, and a minimal user interface. The scheduling algorithm combines the WF^2Q+ fair sharing algorithm [3] and a mechanism called *stealing*. The policy framework divides all processes into two priority levels, important and unimportant, and defines how processes in each class interact with those in other classes; its main feature is that an important real-time process can steal cycles from unimportant processes to meet its deadlines. The user interface includes a button on the frame of each application window that the user clicks to indicate that she considers the application important. In this discussion we will focus primarily on BERT's scheduling algorithm, and particularly on the stealing mechanism.

The goal of the BERT scheduler is to provide multimedia processes with a deadline-oriented QoS while giving conventional processes fair shares of the CPU. To accomplish this, BERT exploits the relationship between virtual and real time implied by the bounded lag of WFQ: if a task's deadline falls after the lag bound of the WFQ algorithm, then the deadline will be met because task will have completed running by then. Furthermore, BERT uses stealing to give an important real-time task extra cycles to meet its deadline when its share is too small. Stealing manipulates the fluid model and virtual time to explicitly redistribute the reserved service of unimportant tasks to an important real-time task. BERT leverages the theoretical underpinnings of WFQ to satisfy the requirements of both real-time and conventional processes.

Stealing is based on the insight that a virtual CPU can be manipulated like a real one. Recall that virtual time itself is a mechanism for multiplexing processes onto the real CPU by providing each with the abstraction of a virtual processor to which it has exclusive access. BERT simply takes away a process's exclusive right to this virtual CPU. Through stealing, BERT multiplexes a real-time task that needs extra cycles to meet its deadline onto the virtual CPU of a less important process. Stealing introduces a dynamic dimension into static fair sharing of the CPU.

The stealing mechanism is spread across several levels of fair sharing theory and implementation. First, it mathematically describes how the virtual multiplexing takes place within the context of the fluid model. Second, the stealing mechanism calculates how virtual time flows for the affected processes in the modified fluid model—one process gets delayed a little (in virtual time) while the other speeds up. Third, the timestamps of tasks belonging to the processes are modified to reflect the changes in virtual time. Stealing uses virtual time to track changes in the fluid model, resulting in tasks receiving new timestamps. This theme will be expanded upon in Section 4.

It may appear that BERT, through stealing, violates the isolation between processes provided by fair sharing; in fact it does not. BERT *preserves* isolation, since a task can steal a specific amount of cycles from another only when explicitly allowed to do so. Instead, BERT redefines the description itself of how processes are isolated from one another by dynamically altering the fluid model.

It is crucial that stealing preserves the relationship between virtual and real time on which BERT depends. We give an informal proof that it does so in Section 4, when we present our scheduler design framework. The important point is that dynamically modifying the ideal fluid model description of the system, as BERT does, and tracking the changes using virtual time, can control the real-time behavior of processes. Stealing forms a prime example of the flexibility and power of virtual time.

3.3 BVT

The Borrowed Virtual Time (BVT) scheduler is targeted for a diverse range of applications (e.g., multimedia, interactive, batch) on a general-purpose OS. BVT's insight is that both interactive and multimedia applications are *latency sensitive*—dispatching their tasks earlier rather than later can improve the overall system performance. BVT does not take multimedia tasks' deadlines into account when scheduling them. BVT focuses on long-term CPU sharing while supporting the varying latency requirements of applications.

BVT supports latency sensitive processes with a mechanism called *warping*. Some processes provide a *warp factor*, which represents a constant that is subtracted from the virtual timestamps of its tasks. When warping is activated for a process (via a syscall), the effective timestamp of its task is lowered by the warp factor, causing the task to move up in the ready queue and run sooner than it otherwise would have. Larger warp factors correspond to lower-latency dispatches than smaller values.

Warping a task can introduce latency for other tasks (including warped ones) and so the BVT scheduler provides additional parameters to warping: a *warp time limit* and an *unwarp time requirement* for each thread. The warp time limit governs the maximum amount of time that a thread can run warped; the unwarp time requirement is the time the thread must then wait before warping again. Warping a process within these constraints forms the means by which BVT provides better performance to multimedia applications. The authors of [6] show experimentally that BVT can be effective in reducing task dispatch latency while generally providing fair sharing of the CPU.

It is difficult to reconcile warping with the notion of virtual time as representing a virtual CPU. Subtracting a warp factor from a task's timestamp seems to be like saying, *do this yesterday*—it has no coherent meaning. Instead, BVT uses virtual time as a simple mechanism for ordering tasks: warping a task moves it up in the ready queue, and this reduces its dispatch latency. As a result, it is not clear exactly what kinds of behaviors BVT can provide. For instance, how do multiple warped tasks interact with each other? How does a user set the various warp parameters for all applications in order to produce a desired overall system behavior? More research on BVT may be necessary to answer these questions.

The BVT mechanism of warping is simple and intuitive. Warping a task in virtual time subtracts a constant from its timestamp, which causes it to run sooner within a framework of long-term fair sharing. It seems likely that with judicious choice of the warp parameters, BVT can provide tasks with shares of the CPU while reducing latency for multimedia and interactive applications.

3.4 Contributions

To conclude our discussion of these three schedulers, we compare the contributions each has made to the idea of creatively using virtual time in multimedia scheduling.

First, the idea at the heart of all three schedulers—to begin with fair sharing and virtual time, and then try to better support multimedia—derives from SMART. SMART has also been influential in framing the discussion of other aspects of multimedia scheduling not discussed here, such as application feedback and user interfaces. However, it appears that SMART's EDF reordering of the candidate set offers only minor benefits in many cases. Since the candidate set is composed of real-time tasks that are adjacent in the ready queue, a system with many conventional tasks will probably have very small sets and thus not be able to benefit a great deal from reordering. Still, SMART should be recognized as laying the groundwork in this area.

Second, BERT's use of virtual time departs most radically from fair sharing. BERT makes no effort to return "stolen" cycles to processes, reasoning that since a more important process has taken them from a less important one, this is the right thing to do. The point is that BERT demonstrates the flexibility of virtual time by using it to go beyond providing shares of the CPU, while at the same time maintaining provable real-time behavior. We will say more about this in the next section.

Third, the warping mechanism of BVT is in some ways the most attractive of the three. It is simple to grasp and straightforward to implement in any WFQ-like scheduler. However, because warping one process affects the latency of others in ways that have not yet been quantified, our understanding of what BVT actually offers is weak.

4 Design Methodology

The three schedulers just described provide examples of the diverse ways that virtual time is being stretched in multimedia schedulers. Next we propose a methodology for creating future virtual time schedulers. Virtual time can form the basis of complex and dynamic schedulers with provable real-time properties. We combine mathematical description, theory, and virtual time to provide a framework for implementing such schedulers.

The key to our framework is tracking a fluid model representation of the system using virtual time. Though this technique has clearly been used to design schedulers (e.g., WFQ), we believe that a general theory of virtual time has not been elaborated before. In fact, the BERT algorithm, which is the prime example of a scheduler designed in accordance with our theory, actually preceded it. First we created and implemented BERT and convinced ourselves that it worked; the insight about *why* it worked came later, and led to our methodology. We realized that the steps we took to create BERT could be used to produce any number of real-time scheduling algorithms.

To create a new scheduling algorithm using our method, the designer follows four steps:

1. Mathematically describe changes to how processes execute in the fair queueing fluid model
2. Track the fluid model changes using virtual time
3. Modify the virtual timestamps of affected tasks
4. Execute the task set in order of increasing stamps

The rest of this section explores what is involved at each step. We use the BERT algorithm as an example of how to use our method, and we intersperse our description with the explanation of why it works. We conclude with some ideas for new algorithms that could be developed using our framework.

4.1 Fluid Model

The Fair Queueing Fluid Model (FQFM for short) forms the foundation of fair sharing algorithms like Weighted Fair Queueing. As outlined in Section 2, the model describes the real-time behavior of an ideal, fluid system in which each process receives at least its reserved rate whenever it is active. The FQFM can be given a concise mathematical definition as follows. Let the n processes in the system be indexed from 1 to n. Each process generates a sequence of tasks, which represent chunks of work of known duration. Let P_i be the ith process and $T_{i,m}$ be the mth task it generates. P_i reserves a cycle rate R_i that can be expressed in any units, for example, cycles per second. Let C_i be the total cycles that process P_i has received so far. Also, let R_{CPU} be the actual processor rate, and let A be the set containing the indices of all currently active processes. At all times t, the fluid model defines the instantaneous execution rates of the current task belonging to P_i:

$$\frac{dC_i}{dt} = \frac{R_i}{\sum_{k \in A} R_k} R_{CPU} \tag{1}$$

The above simply states that the instantaneous execution rate of a process is the proportion of the CPU equal to its reserved rate over the sum of the rates of all active processes. Since admission control ensures that the sum of all rates never exceeds the CPU rate, each running process will always receive at least its reserved rate in the model. Note that the units of the reservation (e.g., cycles per second) do not matter since the model describes an *instantaneous* execution rate.

BERT provides an example of how to dynamically modify the fluid model description of the system. The FQFM provides the base of the BERT algorithm, but BERT departs from the FQFM when one process steals from another. BERT describes stealing at the lowest level in terms of modifying the flow of the fluid model: conceptually, stealing pauses one process in the fluid model and gives its allocation to another for a predefined interval. Formally, this is expressed as follows. When process P_i steals from process P_j, the cycles that P_j would receive during the steal are diverted to P_i. If P_j was idle at the start of stealing, it is considered active (i.e., $j \in A$) while the stealing is going on. During the stealing interval:

$$\frac{dC_i}{dt} = \frac{R_i + R_j}{\sum_{k \in A} R_k} R_{CPU} \tag{2}$$

$$\frac{dC_j}{dt} = 0 \tag{3}$$

It is significant that BERT defines stealing in the context of the FQFM. The reason is that a running fluid model provides a *feasibility test* for a particular real-time system, somewhat like SMART performs when it reorders the candidate set. In the model, an individual task completes at a specific time based on its instantaneous execution rate. This means that the fluid model describes a way that the system *could* schedule tasks to meet a certain set of "deadlines", namely their finish times in the fluid model. Modifying the fluid model, as BERT does, changes the finish times of individual tasks while preserving the descriptive power of the model. This may seem almost trivial, but it is an important point for real-time scheduling.

4.2 Virtual Time

The fluid model provides an ideal description of how the system could schedule tasks to meet a set of deadlines using infinitely fine preemption. The problem is, we cannot know what these deadlines are in advance (though we can put an upper bound on a task's finish time by assuming that the task receives no more than its reserved rate). Since the instantaneous cycle rate of a task depends on the set of active processes, we must know what processes will be active during its execution in order to know what rate it will get. However, in a real system the set of active processes changes unpredictably, for instance as processes enter and leave the system or block on I/O events. The value of virtual time is that it abstracts this problem away.

Virtual time itself flows at a rate proportional to the rate of the active processes. This allows the *virtual* rate of a process (i.e., the rate expressed in terms of virtual time) to be constant and equal to the rate the process has reserved. That is, virtual time lets us provide a simplified description of the system in which each process P_i runs on its own CPU of speed R_i. So, if v is the current *virtual* time, then virtual time flows at the rate:

$$\frac{dv}{dt} = \frac{R_{CPU}}{\sum_{k \in A} R_k} \tag{4}$$

We can combine Eqs. 1 and 4 to express the rate of process P_i in virtual time:

$$\frac{dC_i}{dv} = R_i \tag{5}$$

The significance of the virtual time definition in Eq. 4 is that it allows us to abstract away the active process set from the fluid model. Since the virtual rate of a process always equals its reservation, the *virtual* finish times of its tasks depend only on the tasks' durations and so can be known in advance. Furthermore, virtual time maintains a very important feature of the fluid model description. If all processes actually had their own dedicated CPUs, individual tasks might finish at different times than in the fluid model but they would still finish in the same *order*. In other words, if task A has a larger virtual finish time than task B, then A will finish after B in the fluid model as well. The virtual time abstraction simplifies the fluid model while preserving critical information.

The BERT scheduler uses virtual time to track the effects of its modifications to the fluid model. Stealing changes the virtual finish times of tasks in an easily quantifiable way. During an interval when process P_i steals from P_j, the virtual rate of P_i is $R_i + R_j$, while the virtual rate of P_j is 0. From this, it is simple to calculate the new virtual finish times of the current tasks belonging to the two processes. If the stealing interval is of duration ϵ, and $T_{i,m}$ is the current task of P_i, then the virtual finish time of the task moves up by $\epsilon R_j / R_i$. Likewise, if $T_{j,n}$ is the current task of P_j, then its virtual finish time moves back by ϵ. We will see in the next section how this information is used.

4.3 Modifying Virtual Timestamps

Algorithms based on virtual time (such as WFQ) assign a *timestamp* to each task representing its virtual finish time (VFT) in the fluid model. In WFQ, if v_0 is the virtual time that a task $T_{i,m}$ begins executing in the fluid model, and the duration (in cycles) of the task is $d_{i,m}$, then the timestamp assigned to the task is given by:

$$VFT(T_{i,m}) = v_0 + \frac{d_{i,m}}{R_i} \tag{6}$$

If an algorithm dynamically alters the fluid model description, as BERT does, then this can change the virtual finish time of a task that had previously been assigned a timestamp. In this case, it is necessary to change the timestamp of the affected task so that the ready list continues to reflect the fluid model.

When one process steals from another, the virtual finish times of tasks are affected as described at the end of Section 4.2. BERT modifies the timestamps of tasks in the system accordingly—however, care must be taken when doing so. The reason is that some tasks which are still "executing" in the fluid model may in reality have already run, and so are no longer in the system. It is not possible to modify the virtual timestamp of such a task and so it must not be stolen from.

Rather than checking whether or not a task is in the system before stealing from it, BERT's approach is to rely on the known *workahead bound* of a process. The workahead indicates the amount of a process's reservation that can be received in the real system in advance of the fluid model; in Section 5.1, we show why this quantity is bounded for BERT. Prior to stealing, BERT calculates the amount of cycles that can be stolen from a process before a particular deadline. Since the workahead bound represents cycles that a process may have already received, BERT subtracts them from the total. Though conservative, this allows BERT to safely steal from processes without having to track whether particular tasks have already run.

4.4 Execution Order

Virtual time algorithms execute the task set in order of increasing timestamps. We have outlined the progress of a virtual time algorithm through the fluid model definition, tracking the model using virtual time, and assigning timestamps. At this point we tie it all together and show how running tasks by increasing timestamps leads to a real-time algorithm that provably conforms to its fluid model description.

Figueira and Pasquale establish two very powerful results in [7]. First, if the eligible task sequence is *schedulable* under any policy, then it is schedulable under preemptive deadline-ordered scheduling—for our purposes, deadline-oriented scheduling is the same as Earliest Deadline First, or EDF. Second, this same task sequence is δ-schedulable under nonpreemptive deadline-ordered scheduling. Simply stated, these results mean that if it is possible to meet all deadlines using some scheduling discipline, then preemptive EDF will meet them, and nonpreemptive EDF will miss them by no more than a quantity δ, which is the runtime of the longest task in the system.

With these results in hand, the significance of the steps in our method becomes clear. Executing tasks by increasing timestamps runs them in the same order as their fluid model deadlines, and so is equivalent to EDF. By definition, the fluid model itself shows that there exists a method, albeit impractical, of scheduling the tasks to meet these deadlines. Therefore, preemptively scheduling by virtual timestamps meets all fluid model deadlines, and nonpreemptive scheduling misses them by no more than the δ described above. That is, the preemptive algorithm *never* lags its fluid model description, and the nonpreemptive algorithm has its lag bounded by δ. In either case, the actual running system conforms to its ideal fluid model description in real time in a quantifiable way.

The progress of a process in the fluid model never lags the virtual CPU of the process. The reason is that Eq. 4 shows that $dv/dt \geq 1$ when the sum of all reserved rates is less than the rate of the CPU. As long as this is true, then virtual time (showing progress on the virtual CPU) flows faster than real time; this means that, for any interval of time, the cycles received by the process in the fluid model are always at least what it would receive on its dedicated CPU. Therefore, since we have established lag bounds relative to the fluid model, the same lag bounds apply to the virtual CPU description of a process's progress. This result is at least as powerful as those which bound an algorithm's lag relative to virtual time.

BERT depends entirely on this conformity for its effectiveness. As originally described in [2], BERT is a nonpreemptive scheduling algorithm. When BERT needs to meet the time constraint of a video frame, it first assumes that the decoder process will receive no more than its reserved rate in the fluid model and calculates a conservative fluid model finish time for the frame. It then steals enough capacity from less important tasks to ensure that the latest fluid model finish time for the task is at least δ *before* the timing constraint. With this accomplished, BERT can guarantee that the constraint will be met.

4.5 Future Directions

The design method outlined in this section can potentially produce a variety of new real-time scheduling algorithms. Below we briefly discuss a few that we have implemented or will explore as future work.

Latency-sensitive tasks in BERT Our framework allows us to combine different service models together in the same scheduler while preserving real-time behaviors. We can currently support latency-sensitive tasks (e.g., the mouse) by stealing to finish some small distance in the future. This allows us to make no reservation for the mouse, and also to control which processes are affected by moving the mouse—the mouse steals only from unimportant processes, and so for example leaves an important video unaffected.

Many-tiered BERT In its original description, BERT contained only two priority levels: important and unimportant. This could be extended to multiple levels by expanding the fluid model description of how processes on different tiers interact.

No starvation BERT During a stealing interval, BERT can steal 100% of a process's allocation. This means an important process can starve an unimportant one. It is possible to modify the fluid model description of stealing so that only a fraction of a process's allocation can be stolen—say, 80%. This would produce an scheduler in which unimportant processes always make progress even when stolen from.

Alternative fluid models In the fair queueing fluid model, a process reserves a constant rate on the CPU. The key feature of a fluid model is that the sum of instantaneous process execution rates never exceeds the real rate of the CPU—it is not necessary that these rates are always constant. It should be possible to formulate a fluid model in which processes can reserve non-constant functions—e.g., linear functions or even waves. For example, a periodic process that needed low dispatch latency could reserve a square wave that represented its requirements. This has something of the flavor of [8], in which the authors show that packets in a single stream could have different weights and still be serviced in real time. The important point is that a process could reserve a very exotic pattern of CPU service while the system maintained the isolation between processes that is a main feature of fair sharing. There are many details to be worked out; for instance, we may need to use calculus to figure out a task's timestamp, and admission control in such a system could get pretty complicated. On the other hand, there may be alternate formulations of virtual time which can be used to simplify more complex fluid models. At any rate our framework may point the way to entirely new and sophisticated process service models.

5 Implementation

In this section we discuss how two common features implemented by many fair sharing algorithms—fluid model task eligibility and virtual time estimation—can be incorporated into algorithms created using our framework. In addition, we quantify how these mechanisms affect the relationship between the fluid model description and the resulting dynamic system.

5.1 Task Eligibility

It is well known that, in a system running Weighted Fair Queueing, tasks can finish executing long *before* they have completed in the fluid model [4]. When a task receives cycles before they are due in the model, it is called *lead* (in contrast to *lag*) or *workahead*. Some fair sharing algorithms (e.g., WF^2Q+) incorporate a concept of *task eligibility* to reduce the possible lead of tasks and make the resulting system conform more closely to the fluid model. In such an algorithm, only an eligible task may be scheduled to run; a task is eligible if it is currently receiving service in the fluid model. This means that a task with a higher timestamp may run even though an ineligible task with a lower timestamp is present in the system. The result is a system with a provably much smaller bound on the lead of tasks. Task eligibility can be used with an algorithm produced by our framework as well.

The results of Figueira and Pasquale mentioned in Section 4.4 refer to the "eligible task sequence". It must be possible to schedule tasks to meet their deadlines based on the times that they become eligible to run, using whatever definition of eligibility the system desires, and not based simply on their arrival times. In WFQ, all tasks are eligible from the moment they arrive, and the fluid model describes how to schedule them to meet their fluid model deadlines. However, note that the descriptive power of the fluid model applies to an algorithm using the fair sharing notion of task eligibility as well, since these algorithms define an eligible task as one that is receiving service in the fluid model. That is, by definition the fluid model shows how to schedule the eligible task sequence. The only task sequence for which the fluid model would not be valid is one in which an *ineligible* task is executing in the model.

The fair sharing definition of task eligibility can be useful to multimedia scheduling algorithms. For instance, the BERT algorithm uses it to bound a process's workahead, which is then taken into account when stealing. If P_i is a process, $T_{i,m}$ is its current task, $d_{i,m}$ is the duration (in cycles) of the task, and ϵ is the amount of time stolen from the task, then the maximum lead for the task is bounded by:

$$lead(T_{i,m}) \leq d_{i,m}\left(\frac{1}{R_i} - \frac{1}{R_{CPU}}\right) + \epsilon \qquad (7)$$

This is easy to see. A task is not eligible to run until it has begun service in the fluid model. This means that the task will receive its first cycle in the real system no sooner than in the model. In the fluid model, it is possible that the task only receives its reserved rate R_i; at this rate, and if ϵ is the amount of time stolen from the task while it is running, then its last cycle will be received at $t_0 + (d_{i,m}/R_i) + \epsilon$. However, a running task actually receives the rate of the whole CPU. So if the task starts to run in the real system as soon as it is eligible, and it is not preempted, then it will complete at $t_0 + (d_{i,m}/R_{CPU})$. Eq. 7 represents the difference between when the last cycle is received in the fluid model and the real system. It should be straightforward to obtain similar results for other algorithms devised using our framework.

5.2 Estimating Virtual Time

According to Eq. 6, the system must know v_0, the virtual time at which a task begins service (called the virtual start time), in order to assign the correct timestamp to the task. For an active process, this is simply the virtual finish time of the previous task. On the other hand, for an idle process it is the "current" virtual time when the process becomes active. It is necessary to know, what virtual time is it?

The Weighted Fair Queueing algorithm simulates the fluid model so that it can always know the exact virtual time. However, the simulator is complicated to implement and computationally expensive to run. More recent fair sharing algorithms instead *estimate* the current virtual time. For example, WF^2Q+ takes the following approach. Eq. 4 implies that, as long as admission control prevents processes from reserving more than the total CPU rate, then $dv/dt \geq 1$—that is, virtual time flows at least as quickly as clock time. Also, the actual virtual time can be no less than the the smallest virtual start time of a task in the system—the fluid model is work-conserving, so at least one task in the system must be running in the model. Therefore, WF^2Q+ updates its global virtual time by the elapsed clock time, or up to the smallest virtual start time of a task, whichever is greater. This method results in an estimate which is either accurate or slightly conservative.

Virtual time estimation brings up an interesting issue that we plan to investigate as future work. Namely, how can we understand the way that introducing a new mechanism into WFQ (such as estimating virtual time, or perhaps warping in BVT) affects the real-time behavior of the system? A promising approach may be to describe how the mechanism affects the execution of tasks in the fluid model, and then to use our framework to reason about the resulting system.

For example, consider the effect of a conservative virtual time estimate on an idle process that becomes active. If the estimate is δ less than the actual virtual time, then newly active process P_i will receive $R_i\delta$ cycles more than it would have if the virtual time had been accurate. An analogy with BERT's stealing may help us understand what is going on. It seems as if P_i gets to immediately steal from all active processes, for an interval of duration $(R_i\delta)/R_{CPU}$. This pushes back all other task finish times in the fluid model by this amount. However, unlike BERT's stealing, no timestamps are changed; the cycles "stolen" by P_i correspond to a part of its share that was fairly distributed to the active processes when P_i was idle. That is, the effect of underestimating the virtual time is that a newly active process gets to take back some of the cycles that were borrowed from it when it was idle.

Additionally, virtual time estimation in conjunction with task eligibility (see Section 5.1) can affect the lag of tasks relative to the model. Typically, the eligibility check for a task involves comparing its virtual start time with the current virtual time; if the task's virtual start time has already passed, it is eligible. Underestimating the virtual time by δ means that a task may become eligible up to δ later than it should. It seems that delaying the point that a task is eligible to run could push its finish time back by a corresponding amount. Worse yet, the fluid model no longer provides a description of this system, since an ineligible task (from the system's standpoint) is executing in the model. However, the solution may be to formulate an alternative fluid model in which the affected tasks begin running δ later (in virtual time) than in the standard model. This modified fluid model would accurately describe the system, and so we could determine the system's real-time behavior by referring to the model.

For multimedia scheduling, the practical effects of underestimating the virtual time appear to be minimal. In the case of an idle process becoming active, no process actually receives less than its reserved share—the cycles "stolen" from a process P_i by a newly activated process P_j correspond to excess capacity received by P_i when P_j was idle. For example, the real-time behavior of BERT is not affected because it calculates the maximum finish time for a real-time task using the worst-case assumption that it will receive only its reserved rate. Idle processes becoming active can affect the actual, but not the worst-case, finish time of the task. Furthermore, when we combine estimation with eligibility, we appear to get a system in which eligible tasks can steal a few cycles from tasks that are not yet eligible due to the error in our estimation. However, since all tasks are affected by the error, and every task eventually becomes eligible, it seems likely that it may all even out; in fact, WF^2Q+ combines both mechanisms and has been shown to have its lag bounded relative to the virtual CPU [3]. We plan to investigate the relationship between such mechanisms and our design framework in the future.

6 Conclusions

We have proposed a design methodology for creating complex and dynamic multimedia scheduling algorithms which go beyond fair sharing in their behaviors. An algorithm designed using our framework starts with an ideal mathematical description of the system based on modifying the standard fair queueing fluid model, applies the abstraction of virtual time to this description, assigns timestamps to tasks according to their virtual finish times, and runs the resulting system in order of increasing stamps. The system scheduled by such an algorithm conforms to its ideal description in real-time in quantifiable ways. Our framework allows designers to create new and interesting multimedia algorithms that maintain the guarantees and isolation which are hallmarks of fair sharing.

The key to our framework is virtual time. Virtual time simplifies the ideal system into a form that can be tracked using virtual timestamps—it transforms the mathematical description of a complex system into an algorithm. Virtual time is a bridge between theory and code, and this is its power.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.