Creating New CPU Schedulers with Virtual Time

Andy Bavier
Department of Computer Science, Princeton University
Princeton, NJ 08544
acb@cs.princeton.edu

Abstract

We propose a design methodology for producing CPU schedulers with provable real-time behaviors. Our approach is grounded in the well-known technique of using virtual time to track a fluid model representation of the system. However, our work aims to go beyond traditional fair sharing schedulers like Weighted Fair Queueing by laying bare the mathematical foundations of virtual time. We show how arbitrary changes described mathematically to the fair sharing fluid model can be manifested in real time in the running system. The BERT scheduler is presented as an example of how to use our framework. We hope that this work will enable the creation of new and interesting real-time scheduling algorithms.

1 Introduction

We propose a methodology for creating complex and dynamic schedulers with provable real-time properties. The key to our framework is using *virtual time* to track a fluid model representation of the system. Though this technique has clearly been used to design schedulers (e.g., WFQ in 1989), we believe that a general theory of virtual time has not been elaborated before. In fact, the BERT algorithm [2], 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. Since modifying virtual-time schedulers to change their behaviors has been the subject of recent work [4, 6], we hope that others will find our approach useful.

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 paper summarizes 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.

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. 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. When these requirements conflict, as they are bound to do in an overloaded system, the system should use the importance of processes *to the user* to resolve conflicts in an intuitive way.

BERT comprises a virtual-time-based scheduling algorithm, a simple policy framework, and a minimal user interface. The scheduling algorithm combines the WF^{2}Q+ 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 on BERT's scheduling algorithm, and particularly on the stealing mechanism.

BERT exploits the relationship between virtual and real time implied by the bounded lag of the WF^{2}Q+ algorithm.

BERT notes that if a task's deadline falls after the lag bound of the algorithm, then the deadline will be met because the 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. 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 the virtual multiplexing of tasks 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.

It is crucial that stealing preserves the relationship between virtual and real time on which BERT depends. In the rest of this paper, we demonstrate why stealing works in real time, and in the process outline a framework for deriving new scheduling mechanisms based on virtual time.

3 Methodology

The Fair Queueing Fluid Model (FQFM for short) forms the foundation of fair sharing algorithms like Weighted Fair Queueing. 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 that 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. 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. We will return to this later.

3.1 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.

3.2 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 queue 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 3.1. 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 [1] we show that 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.

3.3 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 [5]. 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 increas-

ing 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, preemptive 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 task, it first assumes that the task's process will receive no more than its reserved rate in the fluid model and calculates a conservative fluid model finish time for the task. 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 Future Directions

Our methodology provides scheduler designers with two additional degrees of freedom over traditional fair sharing. First, the dynamic behavior of the system can be modified in specific and controlled ways on the fly. The designer describes changes to a fluid model of execution, and uses virtual time to manifest the changes in real time in the running system. BERT is a prime example of this. Second, the mathematical basis of the FQFM and virtual time appears to leave room for a process to request *any* cycle function as its reservation—for example, it could reserve a sine function or a square wave. We must simply do the math to calculate virtual timestamps for tasks according to the reservation function, and theory takes care of the rest. A non-constant rate function may allow a process to describe its resource needs more precisely than a simple "slice of the CPU". We intend to investigate both of these directions more fully in future work.

References

[1] A. Bavier and L. Peterson. The power of virtual time for multimedia scheduling. In *Proceedings of the Tenth International Workshop for Network and Operating System Support for Digital Audio and Video*, pages 65–74, June 2000.

[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] 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.

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

[6] 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.