Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler

Kenneth J. Duda and David R. Cheriton

Computer Science Department
Stanford University

kjd,cheriton@cs.stanford.edu

Abstract

Systems need to run a larger and more diverse set of applications, from real-time to interactive to batch, on uniprocessor and multiprocessor platforms. However, most schedulers either do not address latency requirements or are specialized to complex real-time paradigms, limiting their applicability to general-purpose systems.

In this paper, we present Borrowed-Virtual-Time (BVT) Scheduling, *showing that it provides low-latency for real-time and interactive applications yet weighted sharing of the CPU across applications according to system policy, even with thread failure at the real-time level, all with a low-overhead implementation on multiprocessors as well as uniprocessors. It makes minimal demands on application developers, and can be used with a reservation or admission control module for hard real-time applications.*

1 Introduction

With modern processor speeds and memory capacities, systems can now run a wide diversity of application tasks, *and they need to* in order to meet user/customer expectations. For example, a software engineer can reasonably watch and listen to a training video on his or her PC while editing and (re)compiling software, and receive a Voice-over-IP call in the midst of this activity, with software performing packet reception, decompression and timed playback as well as sampling, compression and transmission. Further, in embedded systems such as an IP router, multiple command line interpreters and network management tasks can run concurrently with real-time tasks performing routing protocols, packet forwarding and signaling protocols.

Most general-purpose operating system processor schedulers just provide fair sharing of the CPU among competing tasks, with limited support for different latency-sensitivity among competing threads and no guarantees. Thus, in the above example, the scheduler may allow a frame display by the video player to be delayed by the concurrent compilation process when a disk I/O completes shortly before this frame time, degrading the video playback with no real benefit to the compilation. Similar delay would also degrade the voice quality. For hard real-time tasks, excessive delay can cause outright failure.

In contrast, specialized real-time operating system schedulers handle latency sensitivity by allowing *and requiring* application threads to specify their future processing needs in some detail. In particular, with a deadline-based scheduler, a thread is required to specify in advance by what *deadline* it next needs to complete its processing, how many *cycles* it requires to complete the processing, and the earliest *starttime* it is prepared to initiate this processing. However, this complex scheduling model imposes extra overhead on the application developer and the scheduler itself, and makes the scheduler unsuitable for use with unpredictable real-time threads and general-purpose single- and multi-user timesharing systems. Conventional wisdom holds that these costs and special mechanisms are necessary to meet real-time response requirements. We believe they are not.

In this paper, we present *borrowed-virtual-time (BVT)*, a general-purpose scheduling algorithm that allows a single operating system kernel to support the diverse range of applications outlined above, and thus a candidate "universal" processor scheduler. We show that BVT scheduling allows a simple low-overhead implementation, requires little or no change to applications, and provides comparable, if not superior, response behavior to specialized real-time schedulers.

2 BVT scheduling

With BVT scheduling, thread execution time is monitored in terms of *virtual time*, dispatching the runnable thread with the earliest *effective virtual time (EVT)*. However, a latency-sensitive thread is allowed to *warp* back in virtual time to make it appear earlier and thereby gain dispatch preference. It then effectively *borrows* virtual time from its future CPU allocation and thus does not disrupt long-term CPU sharing. Hence the name, *borrowed virtual time* scheduling. This algorithm is described in detail in the rest of this section.

Each BVT thread includes the state variables E_i, its effective virtual time (EVT); A_i, its actual virtual time (AVT); W_i, its virtual time warp; and $warpBack_i$, set if warp is enabled. When a thread unblocks or the currently executing thread blocks, the scheduler runs thread i if it has the minimum E_i of all the runnable threads.

The EVT for the thread is computed as:

$$E_i \leftarrow A_i - (warp?W_i : 0)$$

where $warp$ is determined as described later.

The scheduler accounts for running time in units of *minimum charging unit (MCU)* or mcu, typically the frequency of clock interrupts. That is, a thread that runs for $k * mcu - \epsilon$ time is charged for running for $k * mcu$ time. A thread that runs for t microseconds has this amount rounded up to the next multiple k of mcu and then charged for k time units. If $mcu/2$ is approximately the context switch cost, the rounding up on average charges the current process for the context switch.

The scheduler is configured with a *context switch allowance* C, which is the real time by which the current thread is allowed to advance beyond another runnable thread with equal claim on the CPU. C is typically larger than, and a multiple of, mcu, preventing two compute-bound threads at same AVT from thrashing by switching on every timer interrupt. For example, a system scheduler could use $C = 10$ milliseconds and $mcu = 100$ microseconds. C is thus similar to the *quantum* in conventional timesharing.

2.1 Weighted fair sharing

Each runnable thread receives a share of the processor in proportion to its *weight* w_i over a scheduling window of some number of mcu (see Section 6). To achieve this, the AVT A_i of the currently running thread i is incremented by its running time divided by w_i. In implementation, the scheduler stores for each thread i an mcu advance variable, m_i, that is set proportional to $1/w_i$. The scheduler increments A_i by $k * m_i$ on a context switch when thread i has run for t microseconds, as above. On each AVT update, the scheduler switches from current thread i to runnable thread j if

$$A_j \le A_i - C/w_i$$

to factor C into the context switch decision.

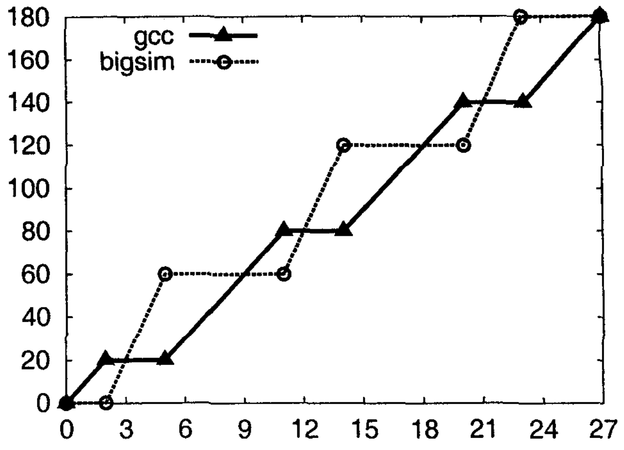

Figure 1. Weighted fair sharing of the CPU. The X axis is real time (in mcu) and the Y axis is virtual time so a running thread appears as a diagonal line and a waiting (runnable) thread appears as a flat line. Gcc has twice the weight of bigsim so it receives 2/3 of the CPU while bigsim receives 1/3. The scheduler gives all runnable threads equal amounts of virtual time, and gcc consumes its virtual time more slowly because of its greater weight.

This sharing of the CPU is illustrated in Figure 1 where the vertical axis indicates virtual time and the horizontal axis indicates physical time (in mcu). Context switches occur when the running thread passes the waiting thread by 2 mcu (the context switch allowance). Over a *scheduling window* of 9 mcu, each thread receives its fair share. In general, the error between fair share and actual allocation is never greater than the context switch allowance plus one mcu if the threads are all runnable. The lines on the graph intersect where precise sharing is achieved.

When thread i becomes runnable after sleeping

$$A_i \leftarrow \max(A_i, SVT)$$

where *scheduler virtual time (SVT)* is a scheduler variable indicating the minimum A_i of any runnable thread[1]. This adjustment prevents a thread from claiming an excessive share of the CPU after sleeping for a long time as might happen if there was no adjustment, as illustrated in Figure 2. With this adjustment, a thread gets the same share of the CPU on wakeup as if it has been runnable but not dispatched to this point because it is given the same AVT in both cases.

The scheduler can consider the AVT of threads blocked by *involuntary* sleep, such as a page fault, as part of the minimum computation to avoid the problem of a runnable batch

[1]SVT captures the notion of the current virtual time of the scheduler's threads, similar to the role of global virtual time (GVT) with optimistic parallel simulation. Any thread that is not runnable is effectively blocked on an event in the future and so does not hold back SVT or GVT.

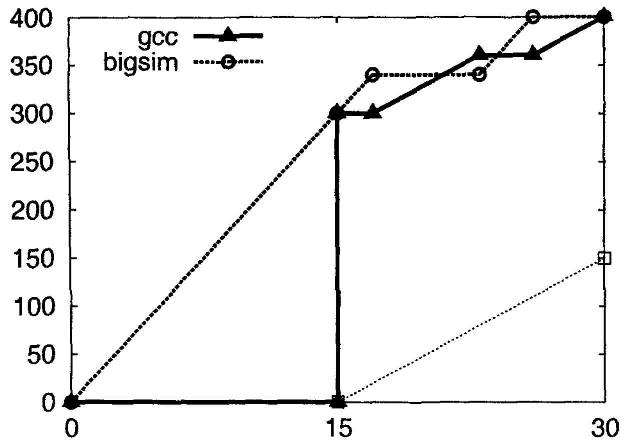

Figure 2. Adjusting AVT for a thread after a long sleep. gcc sleeps for 15 mcu from time 0 to time 15 while bigsim executes. When gcc wakes up, A_{gcc} is brought up to SVT, so it fairly shares the CPU with bigsim over the next several mcu. Without adjusting its AVT, gcc would starve bigsim, as indicated by the lower thin line starting at real time 15 and virtual time 0.

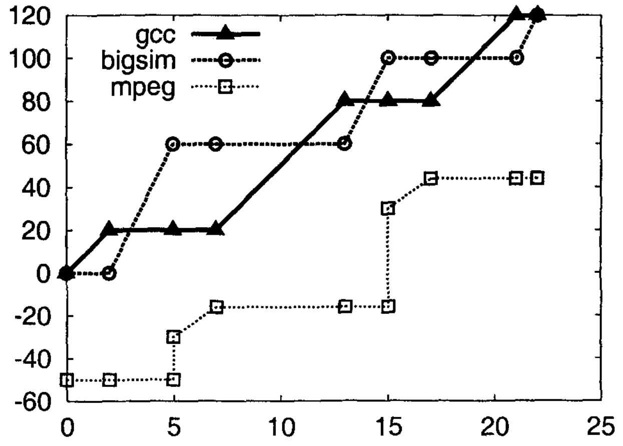

Figure 3. Low latency dispatch using warp, showing effective virtual time on the Y-axis: The mpeg player wakes up every 10 mcu (time 5 and time 15), uses 5 mcu, and goes back to sleep. It runs first because it is warped back 50 virtual units, giving it the earliest EVT. Like any thread, when it wakes up, its AVT (not shown) is advanced to SVT (causing the short vertical segments in its EVT).

thread effectively losing a portion of its share by taking a page fault immediately after being (finally) dispatched. It also fits the logical model because the thread is not logically waiting for an event, just delayed by implementation artifact, namely the virtual memory system. (This is equivalent to only doing this AVT adjustment to a thread if its sleep was voluntary, such as waiting for user input.)

This scheduling behavior is similar to weighted fair queueing [2] and start-time fair queueing (SFQ) [5].

2.2 Low latency dispatch

A thread is created with a non-zero warp W_i to give it dispatch preference. Larger warp values provide lower latency dispatch than smaller values. The $warpBack_i$ flag can be set directly by a system call, causing the thread to run warped normally, or it may be enabled on signal invocation by passing a SA_BVT_WARP flag to `sigaction()`, causing the thread's signal handler to run warped.

Figure 3 illustrates an MPEG player using a warp value of 50 to achieve low latency dispatch in competition with `gcc` and `bigsim` sharing the same processor. When the mpeg player wakes up to generate the next frame, it immediately preempts the other applications because its EVT is $SVT - 50$ because of its warp whereas the other programs are at SVT or later, by definition. However, the MPEG player's long-term usage is still constrained by the weighted fair sharing of BVT, similar to SFQ, because its A_i is advanced based on its actual CPU usage. Without warp, the MPEG player could be delayed at frame time by other interactive and batch application threads with the same AVT, time-slicing with these same threads to completion, similar to the behavior illustrated in Figure 1. Warping thus reduces the jitter from the 10's of milliseconds to microseconds or less, depending on the time for a context switch.

Each thread also has a *warp time limit* L_i and an *unwarp time requirement* U_i, both in real time units, e.g. microseconds. Thread i is allowed to run warped for at most L_i time and, if it attempts to run warped longer, it is then unwarped by the scheduler, which dispatches the new lowest EVT thread[2]. Similarly, if thread i attempts to warp after having previously warped within U_i, the scheduler runs it unwarped until at least time U_i has passed. The unwarped time is measured from when a thread explicitly unwarps or blocks.

Relating back to our earlier setting of E_i,

$$E_i \leftarrow A_i - (warp?W_i : 0)$$

thread i is run warped (i.e the *warp* variable being true) if $warpBack_i$ is true and the L_i and U_i limits are satisfied. With L_i non-zero and $U_i = 0$, a thread is automatically unwarped after running warped for L_i time and remains unwarped until it explicitly sets its state to warped again. With $L_i = 0$, a thread has no time limit on how long it can run warped.

These warp parameters can be set to limit the short-term CPU consumption of a high priority (i.e. high warp) thread more strictly than the thread weight does and thus limit the latency it can add to other threads, as illustrated in Figure 4. In contrast, a strict priority scheduler would allow a high

[2] Ideally, the hardware provides an accurate interval timer or cycle counter that detects the case of a thread running for longer than its warp time limit, allowing this limit and U_i to be accurate to microseconds.

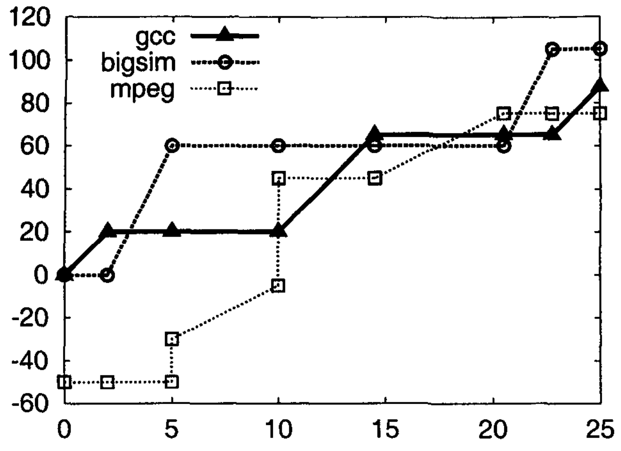

Figure 4. Infinite loop in a low latency thread: The MPEG player wakes up at time 5 and goes into an infinite loop. By time 10, it has exceeded L_i causing it to be unwarped at which time it reverts to its A_i, allowing other threads to run, preserving weighted fair sharing.

priority thread in an infinite loop to starve lower priority threads.

The U_i parameter prevents a periodic task from using excessive CPU in the short-term by waking up too frequently. For example, a periodic device task that requires low latency dispatch every 100 milliseconds and consumes well under 1 millisecond of CPU each time can use a large W_i to get immediate dispatch when it unblocks, with a L_i warp time limit of 1 milliseconds, and unwarp time U_i of say 99 milliseconds. If the device fails and signals the thread every 2 milliseconds, the scheduler prevents the thread from warping and using excessive CPU cycles during the application scheduling window. Without these warp limit parameters, a failed device thread with a large warp could run for 200 milliseconds or more before being preempted because of exceeding its CPU share.

The default value of 0 for L_i and U_i is suitable for most threads because their warp values and CPU share are too low to significantly interfere with other threads. Thus, these parameters need not complicate the system configuration except for time-critical real-time threads.

The BVT scheduler implementation includes a logging facility we have used to debug the scheduling behavior of programs, including detecting threads running into their warp time limit and unwarp time requirement. Alternatively, the system could generate a signal or delete a thread when it hits one of these limits.

2.3 Interrupt service routines

An interrupt routine can be handled as the signal handler of a BVT thread, providing it with a weight, warp and warp limits. In this case, the running time of the interrupt routine is accounted for as cycles used by this thread in its signal handler. Then, an interrupt thread that attempts to use an excessive share of the CPU is blocked from running until SVT had advanced sufficiently, just as with other threads and their signal handlers. This avoids the problem of interrupt routines blocking out other processing, as Mogul and Ramikrishnan [9] identified as a problem with network protocol processing.

2.4 Multiprocessor scheduling

Using BVT on a shared memory multiprocessor, each processor runs the earliest EVT thread of all the runnable threads, but adds a *migration penalty* M for each thread that ran most recently on another processor. Thus, if thread i most recently ran elsewhere, its EVT for dispatch locally is

$$E_i = A_i - (warp?W_i : 0) + M$$

This computation favors migrating a latency-sensitive thread to an available processor to achieve lower latency because of its higher warp value. The value M is set small on machines where fast response is critical and larger when throughput is the primary purpose of the multiple CPUs.

2.5 Multi-level scheduling

BVT can be used in a multi-level or hierarchical scheduling structure, similar to hierarchical SFQ [5], to allow a set of threads to be treated as an aggregate with respect to lower-level scheduler. For instance, the first-level scheduler can be configured to run a set of real-time threads as a *closed* system[3] with a known set of threads and requirements and admission control. The first-level allocates a fixed CPU share to the second-level (by the weight of the second-level scheduler) which runs a set of timesharing threads as an *open* system. These threads simply degrade in performance as their demands exceed the second-level CPU share as appropriate for timesharing. A group of real-time threads can be aggregated similarly, such as the set of threads handling network protocol processing.

The two-level scheduler can run in *threshold* mode in which case the second-level thread effectively runs with the second-level scheduler's warp value W_i if its warp is over a specified threshold. This mode is intended for a second-level scheduler running timesharing or best-effort threads where warp values below its threshold just give a thread response preference relative to other second-level threads. Warp values above the threshold are used for threads, such the MPEG player, that benefit from low-latency response even relative to other threads at the first level. Setting a larger threshold value effectively conserves the limited warp time of the second-level scheduler for its higher priority threads.

In *direct* mode, a second-level scheduler runs its threads with the warp value of the thread it is executing. This mode is used when the second-level scheduler is grouping a set of real-time threads to allocate a CPU share to the aggregate

[3] A *closed* system refers to one where an admission control module knows all the threads and ensures that their processing demands can be met, in contrast to an *open* system in which all threads degrade as new threads and load are "openly" added without control.

rather than individual threads. For example, a system may allocate 30 percent of the CPU capacity to network protocol processing, spread across many threads. The second-level scheduler executes with the warp value and warp time limit of the network processing thread it is executing, so the dispatch latency is not affected by running the network processing in this aggregate, assuming the aggregate CPU share is adequate.

A third-level BVT scheduler could provide virtual time-based scheduling within a simulation framework, where AVT corresponds to simulation time and is advanced by simulation events, not weights. Then, the warp mechanism can be used to allow a simulation process to compute ahead in actual virtual time, ahead of the GVT process, as appropriate with optimistic parallel simulation, but limited in compute-ahead by the warp value. We have not, however, explored BVT beyond two levels to date.

This describes the complete system interface provided to the user for BVT scheduling except for administrative controls for setting the weights w_i and warp parameters W_i, L_i and U_i. Setting these parameters and further details on using multi-level scheduling are described in Section 6.

3 BVT implementation

We implemented BVT scheduling on Linux as follows. We use an *mcu* of 10 milliseconds (the timer interrupt period). A thread corresponds to a Linux process. The thread descriptor holds the thread's *MCU advance* $m_i = 1/w_i$ (which must be an integer) rather than its weight.

On a context switch, besides finding the thread with lowest EVT, the scheduler identifies the thread that it would run next assuming no other threads wake up, computes the number of *mcu*'s until the context switch should take place (taking the context switch allowance into account), and stores that value in a *context-switch countdown* in the thread descriptor. Then, on every timer interrupt, the active thread's *mcu* counter is incremented. When it reaches the context-switch countdown, the timer interrupt handler invokes the scheduler.

Whenever the scheduler is invoked (due to a timer expiration, a wakeup, the running thread blocking, or the running thread reaching its context-switch countdown), it first advances the AVT of the running thread i by m_i times its MCU counter, sets the thread's MCU counter to 0, and updates scheduler virtual time. Then it picks a best thread and a second-best thread as described above. Thus, whenever a thread wakes up, we dispatch it *immediately* if its EVT is less than that of the running thread. Coarse-grained switching between CPU-bound threads still occurs because of the context switch allowance.

The standard Linux thread descriptor is augmented with integers representing timer interrupt advance, warp, warp time limit, unwarp time limit, actual virtual time, and the scheduler ID in which the thread runs. To implement runtime limits, we also include a record of how much warped runtime the thread has left, a flag if it is waiting to warp again, and a timestamp of when it is next allowed to run warped again. A new thread inherits those quantities from its parent thread. The initial thread has an MCU advance of 10, a warp of 200, and 0 for the remaining fields.

We implemented a hierarchy of two schedulers, scheduler 0 (reserved effort) and scheduler 1 (best effort). A per-scheduler structure holds the scheduler's scheduler virtual time, context switch allowance, and warp threshold. Because the best-effort scheduler acts like a thread in the reserved-effort scheduler, it also contains the per-thread fields described above. Each level runs the same BVT scheduling algorithm but differs in its parameters and admission policy. Only root can add threads to the first level, while the child scheduler provides the conventional open system, allowing threads to be added dynamically without restriction.

The additional storage our BVT implementation adds to the Linux kernel is 92 bytes of global variables (mainly 36 bytes per scheduler times 2 schedulers), 52 bytes per thread, plus a 64k debugging log.

To avoid integer overflow problems, when system virtual time exceeds `0x70000000`, the system subtracts `0x60000000` from all virtual time quantities in the system. With expected parameter values, this happens about once every 18 days. Runnable threads are kept in an unsorted linked list, so the cost of selecting a thread for dispatch grows linearly with the number of runnable threads.

We also implemented a `bvtctl()` system call to set scheduler parameters, a user-level `bvtctl` program to provide a shell interface to the system call, and a modest amount of kernel instrumentation monitoring code to help gather experimental results.

The most significant change is in the scheduler main loop, where 89 lines of code update the thread descriptor of the thread that just ran, and 50 lines select the next thread to run. Another 165 lines implement a new `bvtctl(2)` system call used to manage BVT scheduling from user space (specify warp runtime limits, set per-scheduler parameters, move threads between schedulers, *etc.*). 22 new lines in the routine to add a thread to the run queue check to see if the new thread has an earlier effective virtual time than the current one and invoke the scheduler if so. Six new lines in signal delivery warp on behalf of threads that have set the SA_BVT_WARP flag in the thread's signal's `sigaction` `sa_flags` field. The total kernel support for BVT scheduling comes to 447 lines, including comments and whitespace, out of a total kernel size of approximately 750,000 lines.

Our total modifications to the Linux system (including instrumentation code, header files, and the user-level shell interface to BVT scheduling) totals 996 lines of code, and took one engineer two weeks to write and debug. The simplicity of BVT scheduling in lines of code is a significant advantage over other approaches, favoring its use even in relatively small embedded systems.

4 Experiments

The Linux BVT implementation was evaluated by measuring the behavior of several applications, both real and test, namely:

mpeg_play — the Berkeley mpeg_play MPEG-1 video decoder [12], playing back MPEG video files at 60 frame per second, with a warp = 50000. It was modified to record whether the frame got displayed on time by reading the system clock after receiving the X Shared Memory Operation Complete event; we define "on time" as within 30 millisecond of the ideal time based on the previous frame[4].

int — run in 125 millisecond bursts with a warp of 500, using 30 percent of the CPU, modeling a heavy interactive process.

grep — unmodified Linux grep/find.

cont — run continuously with warp 0, modeling a CPU-intensive process.

rt — runs for 5 milliseconds with a warp of 100000 and then sleeps for 95 milliseconds with a corresponding warp time limit and unwarp time requirement, modeling a periodic hard real-time task.

All experiments were run on a Pentium III 500 Mhz system with 384 mb RAM running Linux 2.3.17 modified to include our implementation of BVT, which can be enabled or disabled using a runtime switch. The kernel was also modified to generate an in-memory log of wakeups, context switches, and warping events, and to measure the overhead of our BVT scheduling implementation. Timestamps are based on the Pentium cycle counter, accurate to 1 microsecond, as provided to the machine-independent layer. Unless otherwise noted, all numbers below were obtained from the logs.

In these experiments, we set the X server's warp to the same as mpeg_play, given they are equally latency-sensitive.

In all experiments, the context switch allowance is set to 200 milliseconds, resulting in a context switch rate between CPU-bound jobs comparable to that of other system such as Linux or Solaris. However, this allowance only affects time-slicing threads; a thread is immediately dispatched on wakeup if its EVT is less than or equal to that of the current thread.

Our measurements indicate that the scheduler overhead is less than 0.3% for all runs, even with two-level scheduling, The overhead includes costs of log generation and the overhead measurement itself (making it a slight overestimate) but not the context switch itself or the indirect costs of reduced cache and TLB performance, because all scheduling algorithms incur these costs. The 0.3% scheduler overhead indicates that this cost is not significant with BVT, so it is not considered further.

Also, our experiments indicate that each thread gets within a few percent of its weighted fair share of the CPU when the thread is runnable over a significant period of time. Moreover, as argued in Section 6, this should always occur over a suitable scheduling window of time.

Thus, the rest of this section focuses on dispatch latency and response time.

[4]In the measurements of this section, a stricter notion of "on time" such as 1 ms. would have made BVT look even better and Linux appear worse, while not reflecting what was actually visually noticeable.

Measure	BVT	Linux
Frames	553	284
frame rate	29.78	14.91
late	8	113

Table 1. Video Player frame performance when competing with a large-scale text search. A frame is on-time if within 30 milliseconds of the frame time.

4.1 MPEG player and grep

The first measurement captures a variant of the scenario described in the introduction, namely a software engineer watching a training video running mpeg_play on his or her Linux PC while running a 4-way parallel "grep" over a large number of files, such as a product source tree. Table 1 characterizes MPEG performance for two configurations: 1) BVT scheduling with mpeg_play and the X server warped by 50000 and other programs warped by 100, and 2) standard Linux scheduling with default parameter settings and the same programs.

Subjectively, with BVT, the video is basically glitch-free while with standard Linux, the video is painful to watch. This assessment is supported by the measurements. As the first line of the table indicates, Linux produces only roughly 51% of the frames produced under BVT over the same time interval, leading to roughly half the frame rate. With the mpeg_play implementation, the video playback is simply slowed down by that amount, but if it was held strictly to real time, it would drop about half frames, often multiple frames at a time. Moreover, 113 frames or almost 50% of the delivered frames were late under Linux, several late by over 100 ms. The net effect is an unacceptably poor quality of video playback under standard Linux.

With BVT, only a little over 1 percent of the frames were late, likely due to I/O conflicts, so the video performance was fine. The large 4-way grep receives roughly 20% less CPU than under Linux, but that seems like a reasonable price to pay for high-quality video back.

We also performed experiments running mpeg_play concurrently with cont, a strictly compute-bound test program. However, the standard Linux scheduler correctly gives compute-bound processes lower priority so the differences are not as significant. Strictly compute-bound applications are less realistic compared to a real text search, which makes extensive use of disk. Nevertheless, the standard Linux scheduler did surprisingly well running mpeg_play against various batch applications, suggesting that a more sophisticated scheduler is most compelling with really latency-critical workloads, such as (hard) real-time.

Measure	BVT	Linux
Frames	1198	904
Very Late	0	273
Late	0	41
Max. int. latency	15.7	13.0

Table 2. Performance of video player with int: very late is beyond the time of the test and late is more than 30 milliseconds from frame time.

Measure	BVT
Frames	662
Late	1
Max. int. latency	68.0
Mean int. latency	19.0

Table 3. Performance of 2 video players with int

4.2 MPEG player with interactive task

Table 2 shows the results of running the MPEG player with one instance of `int`. BVT produces all frames on time whereas Linux delays more than 23% of the frames outside the test run time of 20 seconds and delivers another 3% later than 30 milliseconds. The maximum interactive latency with BVT is larger as one would expect in giving preference to MPEG, but only by 2 milliseconds, an unnoticeable increase for interactive applications.

Table 3 shows the results of running two MPEG players with one instance of `int`, just for BVT. This test suggests that BVT is able to handle multiple simultaneous latency sensitive tasks. Running two MPEG players does increase the maximum interactive response to 68 milliseconds in this test, but that is still quite small, and the mean at 19.0 milliseconds remains similar to that of the first test.

4.3 Hard real-time thread performance

Table 4 shows the performance of three test programs, `rt`, `int` and `cont`, running under BVT. As expected, `rt` has dispatch latency comparable to the Linux context switch time and a CPU share according to its CPU consumption per period. `int` has longer response time than `rt`, but it is still acceptable for interactive threads and far superior to the batch response time of `cont`.

Table 5 shows the performance of the same three test programs, but with `rt` having failed into an infinite loop. Here, `int` and cont continue to receive a similar share of the CPU and comparable response time as they do without the failure. So, for example, if `int` is a command line interpreter, it can allow the user to restart the failed `rt` thread to recover. In contrast, with a strict priority scheduler, `rt` as a much higher priority thread than `int` would completely starve it, making it impossible for the user to regain control of the system. The dispatch latency of int at 10 ms is actually the worst that can arise because, with *mcu* as 10 ms and checking warp time limits at the granularity of *mcu*, we always detect exceeding L_i on 10 ms boundaries.

Measure	rt	int	cont
CPU share	5%	29.5%	65.5%
disp. latency	0.005 ms	5.02 ms	265.1 ms

Table 4. Real-time thread with interactive and batch

Measure	rt	int	cont
CPU share	6.1%	30.0%	63.9%
disp. latency	540.0 ms	10.0 ms	269.9 ms

Table 5. Real-time thread failure with interactive and batch

Overall, these experiments show that BVT scheduling provides low latency response for both real-time and interactive tasks competing with each other and batch processes in a general-purpose operating system, even dealing with infinite loop failures by high priority tasks. Linux does substantially worse, even with a quite successful heuristic for identifying latency-sensitive processes. We expect other general-purpose schedulers to perform the same as, or worse than, Linux.

5 Deadlines

To compare the effectiveness of BVT scheduling to achieve low-latency with a specialized real-time scheduler, we implemented a deadline-based scheduler (DBS) and ran a number of experiments, comparing it with BVT.

Under DBS, a thread requests a reservation in the form (s, t, d), requesting t units of CPU between times s and d. The scheduler accepts the request if and only if it is feasible to satisfy it and all previous requests accepted at that point. It then runs the accepted reservations, earliest-deadline-first, distributing any leftover cycles round-robin to threads without reservations.

5.1 Test programs and configurations

A test program `randres` simulates a real-time task scheduling based on deadline reservations for use with DBS. `randres` requests reservations $(s_1, t_1, d_1), (s_2, t_2, d_2), \ldots$ as it runs. When a request is denied, it counts the deadline as missed and does not run until its next reservation. If `randres` reaches its deadline time d_i but has not received the desired quanta, it counts the deadline as missed and does not run again until its next reservation. This behavior favors it making the next deadline compared to just running timesliced at this stage.

`randres` picks reservation parameters as follows. It picks d_i so that $d_i - d_{i-1}$ (the time between deadlines) is uniformly distributed between 26 and 34 *mcu*'s. It calculates t_i (the requested amount of CPU) as $a_i + e_i$, where a_i is its *actual need,* uniformly distributed between 6 and 12 *mcu*'s, and e_i is the *prediction error*, calculated as specified below. Finally, s_i is calculated as $(1-f)d_{i-1}+fd_i$ where f is a test parameter specifying how "fussy" the thread is about when it runs. That is, if f is 0, then the thread is willing to run at any point between its deadlines, so it is easy to schedule, but as f approaches 1, s_i moves closer to d_i, resulting in a constraint that is harder to satisfy.

To approximate error in prediction of CPU need, `randres` introduces this error in one of three modes: overpredict, underpredict, and best guess, corresponding to positive, negative and mixed e_i respectively. The average magnitude of e_i is a test parameter in the range of 0 to 3. Overpredicting models an application that reserves extra CPU to ensure it makes its deadlines at the risk of having its requests denied or causing the requests of others to be denied unnecessarily.

A test workload was configured consisting of two instances of `randres` and one instance of `cont`, run on each of BVT, DBS and (for further comparison) a fixed-priority scheduler (FPS) and a weighted round robin scheduler (WRR) (BVT with warp = 0 for all threads), using various values of f. When running on DBS, `cont` receives only unreserved quanta. When running on FPS, the `randres` instances have priority 1 (highest) and 2, and `cont` has priority 3. When running on BVT, the quantum advances are set to be 100 for each `randres` and 3500 for `cont`, dividing the CPU 49.25%, 49.25%, 1.5%, so that neither `randres` instance ever exceeds its fair share. The `randres` instances warp 40000 and 20000 respectively, modeling two threads with different latency requirements.

5.2 Deadlines: no prediction error

Figure 5 shows the deadlines made with these various test configurations, using $e_i = 0$, i.e. completely (and unrealistically) accurate CPU predictions. The "DBS" column represents optimal performance with FCFS handling of reservation requests, given that the predictions are exact and DBS only runs a task if the task can make its deadline. That is, it does not waste resources or deny requests that it could satisfy in favor of ones it cannot, as can occur with inaccurate estimates.

These measurements show that BVT is within 10% of DBS and has the same behavior as fixed priority (in the absence of failure). WRR is uncompetitive as one would expect. BVT (diff) with different warp values performs better than BVT (same) where the two instances of `randres` use the same warp values. In reality, different tasks have different latency requirements, so different warp values make sense in practice. Moreover, with DBS, two threads that attempt to schedule at the same time run the risk of one being refused. If they do not conflict, they also work just fine with BVT and the same warp value.

5.3 Deadlines: prediction error

Figures 6, 7, and 8 show the effect of prediction error on DBS.

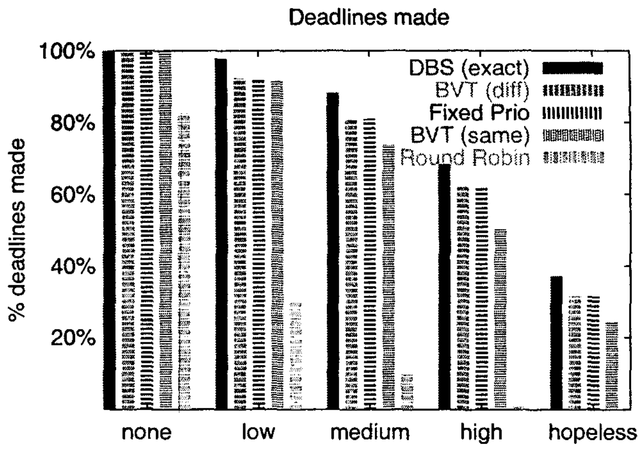

Figure 5. Percentage of deadlines made with different schedulers, relative to different levels of contention, indicated by X axis, assuming accurate CPU need predictions. Levels of contention correspond to $f = 0, 0.4, 0.54, 0.66$, and 0.75 for none, low, medium, high, and hopeless.

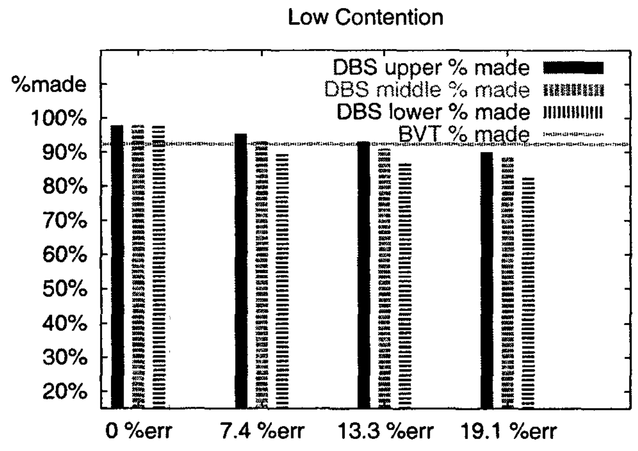

Figure 6. Deadlines made relative to prediction error with low contention (most deadlines feasible).

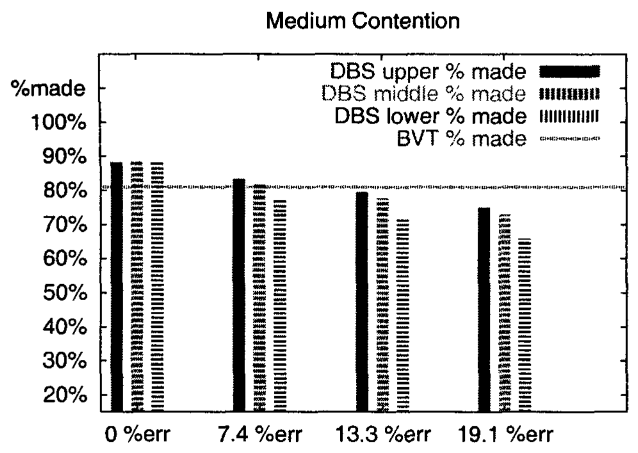

Figure 7. Deadlines made relative to prediction error with medium contention (90% deadlines feasible).

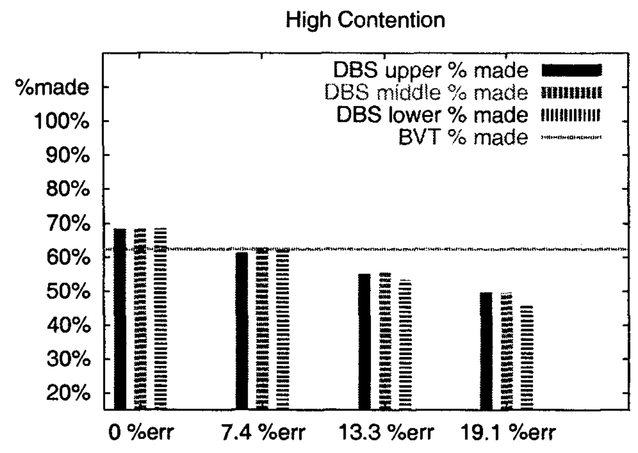

Figure 8. Deadlines made relative to prediction error with high contention (69% of deadlines feasible).

In these graphs, for each degree of error, there is a vertical bar for each of three runs: the first bar, "DBS upper", is for positive error (*i.e.*, the prediction is higher than the actual need); the second bar "DBS middle", is for positive and negative error; and the third bar, "DBS lower", is for negative error. The horizontal line on the graph is the fraction of deadlines made by BVT for the given level of contention. (BVT is not sensitive to the magnitude or sign of error because BVT does not rely on a thread's prediction of its future CPU needs.) When the horizontal line is above the vertical bars, BVT is making a larger fraction of deadlines than DBS.

Considering these graphs, BVT performs as well as DBS under high contention with CPU predictions in error by less than 7.4%, is competitive with DBS at medium contention with error between 7.4% and 13.3%. and performs just as well as DBS under low contention once the prediction error reaches 13.3%. Thus, error in prediction degrades DBS significantly, especially under high load, which is where scheduling matters the most. Therefore, these experiments suggest that, unless the application developers can program so that the CPU predictions are within 15%, using BVT is better than using DBS, given the other advantages of BVT over DBS such as simplicity, efficiency and greater generality. Predicting future CPU needs within 15% is difficult because of uncertainties about workload, cache behavior, TLB behavior, and interrupts. For example, the CPU cycles to decompress and blit a frame can vary by more than an order of magnitude in the MPEG player used in Section 4. Further, even when predictions are exact, only about 10% more deadlines are made using DBS than using BVT.

This basic result should apply to other deadline-based scheduling approaches, such as SMART [11] and EEVDF [4]. These systems differ from DBS primarily in their algorithm for deciding whether to accept a reservation request, adding requirements beyond mere feasibility (e.g. to improve fairness). We do not expect these differences to affect the result.

5.4 Utility of deadline scheduling

The utility of deadlines as the basis for an operating system scheduler seems limited. Many tasks, such as network input processes, do not have specific deadlines by which to accomplish their processing, so the deadline notion does not apply even though these tasks are latency sensitive.

Also, many tasks cannot predict their processing requirements well in advance and thus risk having deadline requests being refused at an intermediate point in the execution of the system. For example, the network input process cannot predict that it needs to process a packet until that packet arrives, and cannot predict how many cycles will be required to process it when it does arrive. The request for a deadline for this processing right when the packet arrives can be refused, leaving the thread to time slice with other low priority threads. On the other hand, threads that *are* suitable for deadlines (because they have predictable processing requirements so they request reservations well in advance) are typically periodic tasks whose requirements are easily expressible as BVT parameters[5] and thus can be handled by a simpler, more general scheduler such as BVT.

Jitter control is also difficult with deadline schedulers. A thread that allows only a short time between the requested starttime and deadline risks the request being refused (because the scheduler has less latitude to satisfy it) causing it to run time-sliced with unpredictable delay and jitter. A thread that allows a long time instead risks significant jitter in its periodic execution (because the scheduler is free to dispatch it any time after the start time as long as it is sufficiently before the deadline).

Finally, deadlines introduce extra complexity into the scheduler as well as into writing the task code that specifies the deadlines and handles their refusal.

A claimed benefit of deadline-based scheduling is that it ensures that (at least) some threads make their deadlines in an open system under overload, as opposed to all threads being degraded and missing their deadlines. However, in practice, a system needs to ensure that *particular* threads make their deadlines over others, not just some random subset. Thus, the system needs to partition the threads into the critical and non-critical, and limit the threads that are allowed to make reservations to the critical set, sized so that their deadline requests can in fact be satisfied. Consequently, the system is necessarily closed for those that can make deadlines and only open for those that cannot, the best efforts threads. Given this partitioning, it is feasible to instead use multilevel BVT and place the critical threads in a first-level scheduler, and the non-critical threads as the open portion in a second-level scheduler. With this structure, another claimed benefit of deadlines becomes less relevant, namely avoiding the "priority inflation" of priority-based scheduling, where designers pick increasing priority levels in an attempt to ensure they meet response requirements. With a closed real-time system, the set of threads and their requirements can be known and the warp parameters can be algorithmically set. (And the warp parameters of the best-effort threads can be controlled by the system, as described in the next section.)

Overall, deadline-based scheduling appears to be of limited utility for applications that we can identify, given the ability of BVT support real-time scheduling and its ability to address, at lower cost and complexity, a much wider range of system requirements and configurations, as described in the next section.

6 Configuring BVT scheduling

Configuring BVT to meet application response and CPU sharing requirements requires careful selection of parameters and levels, especially for hard real-time systems. This section describes how to configure BVT scheduling for these systems.

There are three key dimensions to configure for each thread: CPU share, warp limits and response time. The *CPU share* is the portion of the CPU allocated to this thread over an extended period of time, either absolutely or relative to other threads. From the standpoint of CPU share, there are two types of threads:

- **Reserved Effort (RE)**: A thread for which a specified percentage of CPU cycles are reserved.
- **Best-Efforts (BE)**: A thread that shares the CPU cycles left over from those used by the RE threads, claiming its share relative to the demands of other threads.

Systems using BVT may be entirely RE, entirely BE or a combination of the two, as described below.

The *warp limits* of a thread specify limits on the CPU dispatch preference the thread can use, limiting the amount it can temporarily *warp* the scheduling from its weighted fair sharing.

The *response time* of a thread is the real time from when a signaling event occurs for that thread until it has dispatched and handled that event.

The key BVT parameters per thread, weight, warp, warp time limit and unwarp time requirement, are set to achieve the desired behavior for the application, as described in the following subsections.

6.1 Hard real-time systems

A hard real-time system reserves CPU share and response time for some threads so they can be guaranteed to respond to events within specific real-time limits.

6.1.1 CPU share

Given a static set of RE threads, the system designer or admission control module selects the MCU advances to obtain the desired sharing and to determine the *scheduling window*, the time interval over which the desired sharing is guaranteed to occur. To illustrate the calculation, consider three threads that are to receive 10%, 30%, and 60% of the CPU respectively and assume (for now) that the threads do not warp.

The system designer first calculates the MCU advance for the threads by taking the reciprocals of the weights and then scaling so they are small integers. In the example, the weights are 0.1, 0.3, and 0.6; the reciprocals are 10, 10/3, and 5/3; normalizing (multiplying by 3/5) yields the MCU advances of 6, 2, and 1. The system designer then calculates the *virtual window* (V), which is the smallest amount of virtual time with the property that if the system starts in a state with the threads at virtual times v_1, v_2, and v_3, and all threads remain runnable, then at some later point, the system will reach a state where the threads are at virtual times v_1+V, v_2+V, and v_3+V. To calculate V, the designer computes the least common multiple of the MCU advances times C/mcu. In the example, suppose that $C = 2mcu$. Then the virtual window V is $2*\mathrm{lcm}(6,2,1) = 12$. That is, if we start all threads at virtual time 0, then at some later point, they all reach virtual time 12. Finally, the system designer calculates the *physical window* (W) which is the amount of real time

[5]Conversely, the warp time limit and unwarp time requirement might be regarded as an efficient way to specify deadline-like requirements for periodic tasks. However, these parameters are useful in another cases as well.

it takes for all threads to consume V units of virtual time. In the example, it takes the threads 2 mcu, 6 mcu, and 12 mcu respectively to consume 12 virtual time units; thus, the physical window is 20. The system designer now knows that out of any 20-mcu real-time window, if all threads remain runnable, then each thread will receive exactly its weighted share.

Now, let's extend this analysis to threads that warp.

With high-priority warping threads, the warp time limits are typically set to limit the CPU share they can use within the scheduling window. Moreover, these limits may restrict the thread to running far less than C or even an mcu. For example, in the above example, there could be two hard real-time threads, each allocated 5 percent of the CPU, but limited by the warp time limits to using no more than this amount over a scheduling window. In this case, the CPU share specified for these threads need not be accurate as long as it corresponds to more than the share provided by the warp limits. To handle these threads in the calculations of weights, their CPU share is deducted from the total available, and the weights of the remaining threads are determined as above. For example, to allow for the 10 percent required by these two threads, we reduce the CPU share of the last thread in the original example to 10% and perform the same calculation, but with the shares of these threads adding up to 90%, not 100%.

For warping threads with no unwarp time limit, the MCU advance should accurately specify the CPU share, changing the choice of V if necessary to get this accuracy. For example, if one of the above real-time threads had no unwarp time limit and a 5 percent share on the CPU, its mcu advance m_i can be set to 20. In the worst case, one of these threads could unblock at SVT, warp and run for m of its CPU shares, where $m = W_i/V$, i.e. the number of scheduling windows its warp allows it to advance. For example, if $m = 4$, it could run for 4 of its CPU shares in the current scheduling window. In this case, we can allow $m * w_i$ as its worst-case share of the CPU in computing that available to other threads within this scheduling window. For example, we would have to allow for 20% in the above example relative to other threads if the actual CPU share allocated was 5 percent. Alternatively, we can increase the application notion of scheduling window from the minimum window we have been considering so far, and use a lower multiple for the CPU share. For example, with an application scheduling window of 4 times the minimal or 80 mcu, the above thread can use at most 7 times its CPU share over 4 scheduling windows or less than twice its share in the worst-case. This approach is attractive because many applications do not generally need a very fine-grain guarantee on sharing. For instance, with 10 millisecond mcu, a thread allocated 10 percent of the CPU would receive its share over 800 milliseconds without warping, even if another thread was sleeping and then warped aggressively to run as much as possible[6].

An RE system requires an admission control module that checks on each attempt to create a new RE thread that the system is able to accommodate the new thread. For example, a new thread requiring 10 percent of the CPU when only 5 percent is unallocated should be refused or cause the removal or reduction in shares of existing threads, depending on system policy. If acceptable, the system needs to readjust the weights of the threads as new threads are created and others terminate. This situation can arise at system design time, system configuration time, or even during system execution for systems supporting dynamic configuration.

This admission control module is explicitly *not* part of BVT so different systems can use different modules and policies, depending on requirements while all using the same BVT scheduler. For example, in a statically configured RE system, the admission control can be performed by a module that checks the allocation at configuration or system initialization time and is not even present during normal system execution. A system with a dynamic set of threads may need this module present during execution.

6.1.2 Warp limits

Setting the L_i and U_i values as accurately as possible minimizes the negative impact of a thread failure on the dispatch latency and response time of other threads. The values are selected by measuring the time of the thread's longest processing and adding some safety margin, plus determining the minimum reasonable time between when it needs to run warped. Here, we focus on high priority periodic threads with reasonable predicting processing requirements and behavior. These parameters work less well for unpredictable non-periodic threads.

Accurately setting these parameters is particularly important for very high priority periodic threads. For lower priority threads, the need to set these parameters is less critical. For example, the network input thread may use 0 for its unwarp time requirement, relying on the CPU share to limit its impact on the system on overload (but allowing for it using a multiple of its CPU share within the scheduling window in the worst case, as discussed earlier). However, it may even be attractive to set a non-zero unwarp time limit for this thread if the input packet buffer is large enough, even though packet inter-arrival times are unpredictable. With a non-zero unwarp time requirement, this thread is delayed in dispatch to handle incoming packets when the traffic level is high, causing it to batch-process several packets at a time rather than waking up for each individual packet. The result is more efficient packet handling and more cycles for other threads, at the cost of slightly delaying input packet handling.

The warp value a thread may use is fixed at its creation in general, thereby putting a limit on this dimension of warping as well[7]. The above admission control mechanism must also

[6]This behavior again reflects that fact that a thread, by warping, is borrowing from its future allocation which it has to pay back, and is not gaining a long-term advantage.

[7]An earlier formulation of BVT provided a separate warp limit per thread, allowing the thread to use a warp value up to this limit. We simplified the scheme after observing that threads were, in practice, either warped or not. However, a thread could be allowed multiple levels of warp to deal with, for instance, priority inversion-like issues.

limit the warp values to allow other threads to achieve their response requirements, as developed further below.

6.1.3 *Response time*

Our approach to response time is to pick warp values to mimic the behavior of a conventional priority scheduler, assuming that the application or system threads have already been assigned priorities according to their latency sensitivity. Thus, a thread i that is more latency-sensitive than another thread j is classified as higher *priority*, meaning it gets higher priority to dispatch and run when the two (or more) threads are competing for the CPU.

For RE threads, the priority is mapped to a warp value per thread using the following algorithm:

1. Set the current warp value to 0 and consider the lowest priority level p.
2. Set the warp value W_i for all threads i at priority p to the current warp value.
3. Go to the next priority level, $p-1$. Increment the current warp value by L_i/w_i, where L_i/w_i is the maximum value across all threads at priority $p-1$. (L_i/w_i is the amount the thread's AVT increases by running for the full L_i time period.)
4. If more threads, go to step 2, else terminate.

Low priority interactive threads may operate with a warp time limit of 0 (i.e. no limit) so they may have fairly long execution times occasionally without losing their warp. In these cases, we assume a value of $L_i = C$ for the purposes of the above calculation, taking the view that threads at this level should timeslice when they run for longer than C.

Let's consider the dispatch latency and response time for an RE thread with this assignment, assuming the thread is using the CPU within its share limit, as specified in the weight, w_i and within its warp limits L_i and U_i. (This assumption is reasonable because an RE thread is within these limits unless it has failed or the parameters have been set incorrectly. And, when a thread fails, we are concerned about containment of damage, not response time.)

The highest priority or most latency-sensitive thread i is dispatched immediately after being signaled, executed with a warp value W_i that ensures that it runs for up to its warp time limit L_i before being preempted by the second most latency-sensitive thread, unless it blocks first. If this thread requires $t < L_i$ microseconds of processing time to respond to the event, its response time is $t + c$ where c is the context switch time, including any interrupt disable time latency. c was 5 microseconds in Section 4.

This response time assumes there are no other threads of the same priority that are dispatched during the same time. If there are other such threads, the response time is increased in the worst-case by the sum of the response times of all these other threads[8]. In the typical case, each thread's response time is significantly less than C so each equal priority thread is executed sequentially, the same as would occur with a conventional priority scheduler. The context switch allowance C ensures the second competing thread at the same logical priority level does not run until the first has executed for at least C real time.

[8]However, some threads explicitly schedule themselves at non-conflicting times so do not conflict based on application-level schedules.

For lower priority threads, the worst-case dispatch latency and response time is as above plus the worst-case times for all higher or equal priority threads, the same as for a conventional priority scheduler, in the absence of failures. If a higher priority thread i fails by going into an infinite loop[9], its response time processing from the standpoint of lower priority threads and their response time calculation is L_i, after which it is unwarped and presumably preempted by other well-behaved threads, as was demonstrated in Section 4. Thus, assuming the unwarp time requirements are such that the higher priority threads can only be dispatched once within the application scheduling window, the worst case is the sum of all the L_i's for all higher or equal priority threads. However, this case requires all of these threads to unblock at the same time and fail in infinite loops, presumably extremely improbable.

With a warping thread that has a zero unwarp time requirement, we assume that, if it fails in an infinite loop, it exceeds its L_i causing it to get unwarped and run timesliced with the lowest priority threads until it is restarted. Even lower priority threads may run with a zero L_i indicating no warp time limit. Here, we assume the warp is small, so by running one or a few C times, the thread's EVT is comparable to SVT so it begins to timeslice with other lower priority threads. Also, such a thread has a response time requirement that is necessarily comparable C because it is effectively relying on timeslicing to control excessive use of CPU at its priority level by itself and other threads. This is the case for interactive applications.

The equivalence of the above scheme (in the absence of failure) to a conventional priority scheduler with timeslicing allows BVT to be used directly with systems and applications designed for priorities with little or no modification[10] In this sense, BVT can be regarded as mapping the notion of "priority" into virtual time so that both share and latency requirements can be effectively evaluated by scheduler in a single, simple consideration of thread EVT. A more complex model of managing warp values is possible but has not been necessary for the applications we have considered to date.

The above scheme provides the basis for an admission control module for a hard real-time system.

[9]It makes sense for the scheduler to focus on this particular thread failure because sandboxing techniques and memory protection can catch other forms of failure that might harm the system but cannot solve this "halting problem". Application-level checks can catch higher-level forms of failure, such as performing the wrong action, even if within the right time.

[10]To handle failures, a watchdog thread can run periodically to detect and restart failed threads. More sophisticated failure detection and handling are also an option.

6.1.4 *Example: flight control system*

A life-critical flight control system would use an RE scheduler where the critical flight control threads are guaranteed shares of the CPU and response times, but with strict limits on the maximum they can use, both over the short term using the warp limit parameters and over the long term, using weight. These tasks can be determined, characterized and fixed at system design time, allowing these parameters to be set accurately and system performance guarantees to be verified.

6.2 Best-efforts systems

In an entirely BE system, the threads can be allocated one of a fixed range of weights, corresponding to different allocation preferences. For example, interactive tasks may have a weight of 10% while batch threads may have a weight of 5%, giving them half as many cycles as interactive threads in the fully loaded case. As the number of threads in the system increases, the share of the cycles that each thread receives decreases accordingly, but the relative weights between threads are preserved.

6.2.1 *Warp limits*

Warp limits are set the same as for reserved effort systems with non-zero limits primarily for the higher priority threads, such as our MPEG player. However, the warp limits can be used in conjunction with the scheduler logging facility to determine whether the application threads are in fact executing within the execution parameters that the developer is expecting.

6.2.2 *Response time*

The RE warp assignment scheme can also be used in a BE system where the response time analysis assumes the BE share of the CPU available to the thread is adequate to meet its processing requirements. The response time degrades proportional to the thread's CPU share when the scheduler experiences overload. To provide a simple API to conventional applications, the system could provide a single priority parameter to specify at thread creation time for the thread and/or its signal handler. The system then maps this priority value to a fixed set of values of weight, warp, warp time limit and unwarp time requirement, according to system administrator-selected values. For example, a user priority of 1 might be used for latency-sensitive user threads such as an MPEG display thread, providing it with a weight of 10 (versus 5 for normal threads), a warp time limit of 20 milliseconds, and a warp computable as above, relative to priority 2 interactive threads.

Most BE threads have no warp or a small warp, such as interactive threads. The latter normally execute with $L_i = 0$ but with a presumption of executing warped less than C normally, so their warp is computed assuming $L_i = C$ (as described earlier). If these threads are the lowest priority ones with a non-zero warp, the warp is such that an interactive thread i's EVT E_i is the same as SVT after executing C time in any case, and so will start to timeslice with other interactive threads after executing for at most $C + mcu$.

6.2.3 *Example: large-scale web server*

A large-scale web server such as the Google search engine [1] receives a large number of competing search requests. Most requests are simple and can be handled within a small number quanta, but less frequent complex queries take 150 times as long to process on the average. Using BVT, the system designer can guarantee fair share for long queries while providing good response time to short queries in most cases. A fixed number of threads (say, 10) are assigned to handling requests. Each thread repeatedly dequeues a request from the request queue, processes it, and repeats, sleeping if there are no requests on the request queue. When a new request arrives, it is given to the sleeping thread that has slept the longest, or added to the request queue if all threads are busy. Each thread is assigned the same weight and a warp sufficient to handle a typical simple request (say, 50).

To illustrate the dynamics that arise from this setup, consider the scenario in which the system begins idle and then receives a complex query. Thread 1 dequeues the query and begins working on it. SVT advances along with thread 1's EVT. Then a simple query arrives and thread 2 wakes up to process it, with AVT advanced to SVT as part of wakeup. With 50 units of warp, thread 2 may run for up to five quanta in a row while thread 1 waits. Thus, short queries complete in half the time compared with Unix-style round robin. Next, a third complex request arrives, awakening thread 3, which runs for five quanta and then time slices with thread 1. Subsequent short requests now complete in one-third the time compared with round robin.

The behavior above continues until the system becomes overwhelmed by short queries for a sustained time period. When a simple query is running, borrowing time from a complex query, the AVT of the thread handling the simple query is advancing while SVT is not. SVT only advances past a given point when all threads' EVT (including the complex query thread) have passed that point. Thus, once the other threads have consumed their warp on short queries, the complex queries get a chance to run; no matter what the offered load, each complex query receives its weighted fair share of the CPU in the long run.

6.3 Combined real-time and best-efforts systems

A system supporting a mixture of BE and RE threads is configured with a two-level BVT scheduler, the first level handling RE threads and the second-level handling BE threads, dividing the CPU cycles available to it among these BE threads according to their weights. The second-level scheduler or BE scheduler uses threshold mode, so threads are only run warped relative to the first level when their warp is over a specified threshold. Thus, the first-level or RE scheduler corresponds to an RE system, with some CPU share assigned to BE threads through the BE scheduler, so the RE techniques above are applied to the first level and BE techniques are applied to the second.

The BE scheduler can have its warp value set to effectively give it higher priority than some or all of the other RE threads, if desired. The impact on the response time of other RE threads of giving this BE scheduler priority is limited by the scheduler's warp time limit and unwarp time requirement, the same as other threads. For example, the BE scheduler might have a warp value that is greater than that of an RE thread that handles the disk subsystem. Then, an MPEG player frame update (for example) can be dispatched with lower latency providing the BE scheduler thread is within its CPU share and L_i without significantly increasing the file system response time, given the latter is dominated by physical disk latencies. When the BE scheduler thread exceeds its share and/or warp limits, the behavior of the BE threads degrade accordingly, as appropriate for best-efforts scheduling. With this configuration, the user sees high-quality video play under normal (non-overloaded) conditions, with disk actions occasionally being delayed in favor of displaying a new video frame, but the delay is no more than that occurring from rotational latency or a longer seek latency.

The RE scheduler can also handle non-latency-sensitive threads that simply need a reserved CPU share. For instance, a periodic review or housekeeping task can be executed by the RE scheduler with a warp of 0 and a weight equivalent to 10 percent of the CPU, ensuring the review is performed no matter want the demand is from other threads.

This two-level BVT scheduler approach supports RE and BE threads using the same BVT scheduling algorithm at each level, but with a clean interface and separation between the two levels. It also allows the system configuration to specify the aggregate CPU share reserved for BE threads. Finally, it minimizes the need to modify the weight and warp values of threads at both the RE and BE levels. Supporting RE and BE threads in a single scheduler incurs the overhead of revising the weights and warp values of all threads every time a new thread was created or deleted. A single scheduler would also not allow full control of the aggregate CPU share available to BE threads.

6.3.1 *Example: router software*

A modern router includes a performance-critical real-time component that deals with packet forwarding plus real-time routing protocol tasks and timesharing-like command-line interpreter (CLI) tasks for managing the router. Packet forwarding can use BVT-scheduled RE threads (including interrupt service routines as their signal handlers), providing it with low latency dispatch and a guaranteed share of the CPU. The routing protocol tasks can run at the RE level but with lower priority. Moreover, a thread implementing the signal processing of A-to-D and D-to-A processing for a "soft" modem would execute under the RE scheduler. It can then be guaranteed low latency dispatch, yet limited to a specified long-term share of the CPU plus a limited number of cycles per C units of time.

A second-level scheduler runs the CLI threads, which can vary in number, depending on the users and administrators accessing the router. The BVT guarantee of a specified CPU share to the routing protocol tasks and the CLI threads is critically important to ensure that packet forwarding overload cannot disable either of the former tasks. For instance, a network administrator must be able to use the CLI to correct a misconfiguration of the router that could be causing an excessive level of packet forwarding. BVT insures that packet forwarding receives low-latency service as long as the system is not overloaded, while routing protocol processing is assured of receiving CPU in accordance with its weight in the long term. Using BVT for all scheduling in a router rather than various specialized and *ad hoc* throttling mechanisms keeps the router implementation and behavior simpler and thus less error-prone.

6.3.2 *Example: multi-user timesharing*

A conventional timesharing system such as Linux uses various heuristics in the scheduler to guess which programs are latency-sensitive (i.e. interactive) rather than requiring explicit setting of a parameter such as warp. For example, Linux uses a *counter* parameter that is decremented as a process runs a full quantum and when the process is not dispatched when it unblocks, all heuristic indications of it not being an interactive latency-sensitive program. It is incremented when the process blocks before the end of a quantum.

This heuristic learning mechanism can be readily adapted to BVT.

A warp value W_{int} is selected that is less than the warp of real-time applications. Then, the Linux *counter* mechanism is modified to update the warp value between 0 and W_{int}, depending on how interactive this process behaves. Consequently, a process that behaves "batch-like" ends up with a 0 warp while one that acts interactive retains the full warp value of W_{int}. Thus, when an interactive task unblocks, it typically has an EVT of $SVT - W_{int}$, causing it to run immediately unless delayed by a real-time application such as an MPEG player, just as latency requirements would dictate. The interactive thread's warp is relatively small because it is meant to enable rapid completion of a small amount of work, rather than to enable the thread to monopolize the CPU for an extended time period.

This warping provides the same behavior as well-proven systems such as Unix, where the scheduler grants a modest preference to a thread that has recently awoken in order to improve the response time of interactive tasks. However, it is provided as part of a more general scheduling mechanism that also accommodates real-time tasks.

To prevent undue interference between users, a process can inherit its weight and warp parameters from its parent process. A user process can effectively "nice" his or her processes by reducing their warp parameters and weight in the same spirit as the UNIX `nice` command, but cannot increase them.

7 Related work

Various scheduling algorithms use a number of different bases for selecting the next thread to dispatch.

Several previous systems [14, 5, 4, 11] use the same virtual time basis as BVT for measuring and controlling long-term sharing of the CPU, or something that behaves similarly. However, most do not provide application control over low-latency dispatch and those that do introduce extra mechanisms such as deadlines. Start-time fair queueing (SFQ) [5] also describes the use of a hierarchy of schedulers, similar to our multi-level scheduling with BVT.

In contrast to virtual time-based BVT, priority-based schedulers base dispatch decisions on the priority value associated with each thread, running the ready thread with the highest priority, often represented as the lowest numeric value. These schedulers have the problem that a high priority thread going into an infinite loop permanently starves the lower priority threads. For example, in POSIX real-time scheduling, the user may assign a fixed high priority to latency sensitive threads, thereby assuring low-latency dispatch to at least the highest-priority thread. Nieh et al.[10] show that this mechanism can significantly skew long-term CPU allocation to the point of total starvation of all but the single highest priority thread. Priority scheduling can be extended by providing time slicing of threads of equal priority and also having the system modify the priority of timesharing threads based on their CPU consumption, as commonly done in Unix implementations. However, this requires a separate accounting of CPU usage and mechanism to modify the priority as part of thread execution.

In deadline-based scheduling, such as in Spring [13], SMART [11] and Rialto [7], threads declare future CPU needs to the system. A periodic thread may express a sequence of similar reservations as a single period length and need per period. The system either accepts the request, in which case the thread is guaranteed to be dispatched according to its predeclared need, or the system rejects the request, in which case the thread receives no preferential dispatch.

Using BVT, this reservation capability is modularly separated from the scheduler and provided as a higher-level system function in a reservation or admission control module, as described in Section 6. For instance, it appears that the Rialto "system resource planner" [6] could perform this function on top of BVT. This separate facility also allows the unit of CPU resource accounting and planning to be separate from abstractions such as threads or processes, as done in Mercer *et al.*'s processor capacity reserves [8] and in Rialto.

The SMART scheduler [11] adds deadline-based scheduling for real-time threads to the basic virtual time mechanism, notifying applications that it determines cannot make their deadlines. SMART adds a bias to the virtual times of non-real-time threads, thereby preferentially dispatching the real-time threads, but does not provide application control of this bias. We believe that BVT scheduling performs comparably to SMART for the video player application without incurring the complexity of deadlines. Furthermore, in SMART as in other deadline-based systems, there is some risk that the system predicts that some deadline cannot be met when in fact they could be (due to other threads not using their full reservation), and there is some risk of the converse as well (in the case of sudden network interrupt overload, for example).

8 Conclusions

BVT scheduling in a multi-level implementation provides a universal scheduler, configurable to support applications from hard real-time tasks to interactive applications to batch jobs while ensuring weighted fair sharing among competing threads and protecting against low-latency threads using excessive processing cycles. The two-level BVT scheduler allows hard real-time threads to run in a strictly admission controlled regime while so-called soft real-time, interactive and batch thread run in a second level scheduler that gracefully degrades under increasing load. A Linux implementation demonstrates that this scheduler is simple to implement and provides good performance while incurring a low time and space overhead, all within the context of a general-purpose operating system kernel.

BVT, as its primary innovation, extends virtual time-based scheduling with the ability for a thread to *warp* back in virtual time so that its *effective* virtual time for scheduling is earlier, causing it to be dispatched earlier. Using this warping mechanism with its associated warp limits, our results show that BVT provides real-time performance comparable to specialized deadline schedulers while still supporting general-purpose task scheduling. Also, the standard technique of giving unblocking interactive threads a slight priority boost can be realized in BVT using a corresponding warp value, unifying this proven heuristic into a general scheduler framework.

The virtual time base of BVT provides a single simple measure that handles both CPU share and latency requirements, superior to using priority or deadlines as the basis. In particular warping effectively encodes priority into the virtual time paradigm using a simple algorithm (described in the paper) yet without disrupting the fair sharing mechanism. We have used a *context switch allowance* and *migration penalty* to factor the costs of these operations into this same virtual time measure to prevent thrashing of between threads, on the same processor and between processors.

BVT supports unpredictable threads with a response time commitment within their CPU share without extra mechanism in the scheduler. It also supports best-efforts scheduling together with reserved-effort threads, allowing even best-effort threads to request low latency dispatch. Moreover, these requests are honored to the limit of the availability of CPU resources *at the time of execution* rather than being accepted or denied based on worst-case estimates of future CPU demands, as occurs with deadline-based scheduling. This advantage is analogous to that for a best-efforts datagram networks where bandwidth is shared more efficiently than circuit-switched approaches while freeing applications from the requirement of fitting all communication mechanisms into small fixed-size pipes.

Contrasting with the merits of BVT, our measurements show that deadline-based scheduling has a significant vulnerability to inaccurate prediction of processing requirements, causing this approach to underperform BVT when the predictions are off by as little as 15 percent. This prediction problem is significant, if not insurmountable, in many practical situations because the processing requirements of even a strictly periodic task such as MPEG playback can vary by almost an order of magnitude. Also, interrupt service routines can effectively steal cycles, making the number of cycles available in some time period uncertain. We also point out the difficult trade-off in sizing the scheduling window of a deadline reservation to minimize jitter. These issues and the implementation complexity of the deadline-based approaches further support our conclusion that virtual time-based scheduling extended as in BVT is the superior solution, even for hard real-time.

BVT demonstrates a key principle of operating system design: It is better to provide to a single simple measure, such as virtual time, that provides well-specified behavior relative to other tasks and have applications map their requirements onto this measure than to require applications to specify their requirements in detail. With the single measure approach, applications and system policy modules contain their policies in the mapping to the measure whereas otherwise, the operating system itself performs a complex and fixed mapping that effectively incorporates fixed policies and makes it more difficult for applications to achieve their desired results. Moreover, the simple measure approach, besides fitting with the principle of separating policy from mechanism, allows some systems to determine the mapping at design or configuration time, rather than forcing it to always take place at run-time with the attendant overhead.

Overall, we see BVT as an important step to achieving a fully general yet efficient and stable operating system kernel that can simultaneously execute a wide range of applications with different requirements, behaviors and failure modes. We hope in on-going work [3] to see it tested and deployed across this spectrum to further support our results to date.

Acknowledgements

This work was initially supported by Cisco Systems, Inc. in a project on high-speed switch and router software. Thanks to our "shepherd" Jack Stankovic and the anonymous reviewers plus Andy Valencia, Dan Li and Fusun Ertemalp for their help in improving the content and presentation of the paper.

References

[1] S. Brin. Personal communication. `www.google.com`, 1999.

[2] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Proceedings ACM SIGCOMM*, pages 1–12, September 1989.

[3] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling. `http://www-dsg.stanford.edu/pub/bvt.h` 2000.

[4] I. Stoika *et al.*. A proportional-share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 51–59, December 1996.

[5] P. Goyal, X. Guo, and H. Vin. A hierarchical CPU scheduler for multimedia operating systems. *Proceedings of the Usenix Symposium on Operating System Design and Implementation '96*, pages 107–122, October 1996.

[6] M. Jones, J. Barrera, A. Forin, P. Leach, D. Rosu, and M. Rosu. An overview of the Rialto real-time architecture. *Microsoft Research Technical Report MSR-TR-96-13*, July 1996. Microsoft Inc., Redmond, WA.

[7] M. Jones, D. Rosu, and M. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 198–211, October 1997.

[8] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.

[9] J. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. on Computer Systems*, 15(3):217–252, August 1997.

[10] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 35–48, November 1993.

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

[12] L. Rowe, K. Patel, and B. Smith. Performance of a software MPEG video decoder. *Proc. ACM Multimedia '93*, pages 31–39, August 1993.

[13] J. Stankovic and K. Ramamritham. The Spring Kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–73, 1991.

[14] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible proportional-share resource mangement. *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–11, November 1994.