Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers

David Petrou
Carnegie Mellon University
dpetrou@cs.cmu.edu

John W. Milford
NERSC
jwm@csua.berkeley.edu

Garth A. Gibson
Carnegie Mellon University
garth.gibson@cs.cmu.edu

Abstract

We describe extensions to lottery scheduling, a proportional-share resource management algorithm, to provide the performance assurances present in traditional non-real time process schedulers. Lottery scheduling enables flexible control over relative process execution rates with a *ticket* abstraction and provides load insulation among groups of processes using *currencies*. We first show that a straightforward implementation of lottery scheduling does not provide the responsiveness for a mixed interactive and CPU-bound workload offered by the decay usage priority scheduler of the FreeBSD operating system. Moreover, standard lottery scheduling ignores kernel priorities used in the FreeBSD scheduler to reduce kernel lock contention.

In this paper, we show how to use dynamic ticket adjustments to incorporate into a lottery scheduler the specializations present in the FreeBSD scheduler to improve interactive response time and reduce kernel lock contention. We achieve this while maintaining lottery scheduling's flexible control over relative execution rates and load insulation. In spite of added scheduling overhead, the throughput of CPU-bound workloads under our scheduler is within one percent of the FreeBSD scheduler for all but one test. We describe our design, evaluate our implementation, and relate our experience in deploying our *hybrid lottery scheduler* on production machines.

1 Introduction

Lottery scheduling from Waldspurger *et al.* is a recently introduced proportional-share scheduler that enables flexible control over the relative rates at which CPU-bound workloads consume processor time [21]. With a proportional-share scheduler, a user running several CPU-bound jobs, such as those found in scientific environments, can easily control the share of CPU time that each job receives. In time-sharing systems, proportional-share schedulers control the relative rates at which different users can use the processor, enabling load insulation among users. One possible policy forces users with CPU-bound processes to consume CPU time at an equal rate, regardless of the number of processes they own. Such control is particularly useful for Internet Service Providers (ISPs) which often have hundreds of competing users simultaneously logged into one machine. With conventional processor schedulers, it is simple for one user to monopolize the system with her own processes. Considering desktop workstations, another policy allows the console user to consume CPU time at a faster rate than remote users so that the console user's windowing system is responsive. Naturally, this idea applies recursively so that individual users can control the relative rates at which their own processes consume CPU time.

Although proportional-share schedulers such as lottery scheduling have powerful and desirable features, they are not in wide use. We set out in this research to see if there were any technical obstacles to overcome in an implementation of lottery scheduling on conventional time-sharing systems. We began with a straightforward implementation of lottery scheduling on the FreeBSD 2.2.5R operating system. CPU-bound workloads performed as advertised. However, when running batch and interactive workloads together, we experienced poorer responsiveness, or "choppiness," with the interactive applications compared with the same workload under the stock FreeBSD scheduler.

The FreeBSD scheduler has features to dynamically favor specific processes that lottery scheduling lacks. In this work we show a variety of dynamic ticket adjustment strategies that are harmless to the lottery scheduling goals and that allow us to provide comparable specializations to favor specific processes in specific conditions.

Our *hybrid lottery scheduler* includes *kernel priorities* to reduce kernel lock contention and *abbreviated time quanta* to increase responsiveness by preempting processes before their quanta expire. Our primary contribution to achieve responsiveness comparable to the FreeBSD scheduler is with *windowed ticket boost*, a scheme for dynamically identifying interactive processes and boosting their ticket allocation. We make *tickets*, the priority abstraction in lottery scheduling, adaptively serve a dual purpose based on measured process behavior. If a process is using less CPU than has been granted by its ticket allocation, we identify it as currently interactive and give it an apparent boost in ticket allocation to influence scheduling order. We accomplish this without impacting the ability of lottery scheduling to control the relative CPU time used by CPU-bound processes. Further, while lottery scheduling gives users and administrators flexible resource control, it does not easily offer the UNIX `nice` semantics in which a user (system administrator) can downgrade (upgrade) one of its processes relative to all others without downgrading (upgrading) the rest of its processes. We present an approximate emulation of the `nice` semantics. Our hybrid lottery scheduler, which has been continually running on two production servers and one personal machine for 1.5 years, provides comparable responsiveness and throughput relative to the FreeBSD scheduler under the benchmarks we run.

The rest of the paper is organized as follows. In Section 2 we describe both the FreeBSD and lottery schedulers. Section 3 explains our extensions to the lottery scheduler while Section 4 details our implementation. We evaluate our hybrid lottery scheduler and compare it with the FreeBSD scheduler in Section 5. In Section 6 we present our experience in deploying our scheduler on production machines. Section 7 summarizes related work, while Section 8 discusses work that we leave for the future. Finally, Section 9 concludes this paper.

2 Background

A process scheduler has several conflicting goals. The scheduler should ensure that interactive processes are responsive to user input despite not being able to always distinguish these processes from non-interactive processes. Batch processes should be scheduled to maximize throughput despite potential lock contention between such processes. While addressing these goals, the scheduler must ensure that no process starves. In this paper we are not concerned with real-time schedulers [10].

2.1 Scheduling in FreeBSD

FreeBSD [7] is a UNIX operating system for the Intel x86 platform based on UC Berkeley's 4.4BSD-Lite [14] release. FreeBSD's scheduler is a typical decay usage priority scheduler [4] also used in System V [8] and Mach [2]. The scheduler employs a multi-level feedback queue in which processes with equal priority reside on the same runqueue. The scheduler runs processes round-robin from the highest priority non-empty runqueue. Long (100 ms) time slices make TLB and cache state flushing infrequent, imposing minimal overhead on CPU-bound processes. The scheduler favors interactive processes by lowering the priority of processes as they consume CPU time and by preempting processes before their quanta expire if a higher priority sleeping process wakes up. The scheduler prevents starvation by periodically raising the priority of processes that have not recently run. FreeBSD's scheduler also employs higher priorities for processes holding kernel resources. These kernel priorities cause processes to release high-demand kernel resources quickly, reducing the contention for these resources.

The FreeBSD scheduler has several limitations. Hellerstein demonstrates the difficulty in constructing "fair-share" systems based on decay usage schedulers [9]. These fair-share systems [12, 5] dynamically adjust the priorities of running processes to obtain specific processor consumption rates over the long-term via nontrivial and potentially computationally expensive operations. Further, FreeBSD provides only rudimentary inter-user load insulation by limiting the number of simultaneous processes that one user may run, and by terminating processes that accumulate more than a certain amount of processor time. These mechanisms prevent a user from starting many processes that consume CPU time slowly and also from executing processes that consume a lot of CPU time over a long duration.

2.2 Lottery Scheduling

Recently, proportional-share schedulers such as Waldspurger's lottery scheduling have been introduced which strive for instantaneous fairness; that is, making fair scheduling decisions against only the currently runnable set of processes [21]. In lottery scheduling, each process holds a number of *tickets*. The scheduler selects which process to run by picking a ticket from the runnable processes at random and choosing the process that holds this winning ticket. Users can set the ticket ratios among processes to determine the expected ratios that their processes are selected to run. Only runnable processes (not sleeping, stopped, or swapped out) are eligible for this lottery. Hence, if one process is always runnable while another with equal tickets sleeps periodically, the first will consume more CPU time because it has a greater fraction of the system's tickets during the times that the second was sleeping.

Currencies enable *load insulation* among users, making the rate that a user can consume CPU time independent of the number of processes owned. While processes hold tickets in per-user currencies, users hold tickets in a system-wide base currency. For simplicity, per-user currency tickets will be called "tickets," and base currency tickets will be called "base tickets." The ticket distribution within a user's processes determines the relative rate of execution among these processes. To control the distribution of CPU time among users, the system administrator can vary the number of base tickets held by a user. This varies the total execution rate of all the user's processes with respect to processes owned by other users. The scheduler accomplishes this by first converting the tickets, T_p, belonging a process into a number of base tickets, B_p, and then performing a lottery with base tickets instead of tickets. In detail, $B_p = ET_p$, where E, the exchange rate, is the number of base tickets funding a user's currency divided by the total number of tickets across the user's runnable processes.

Lottery scheduling employs *compensation tickets* to enable a process which goes to sleep before exhausting its time quantum its fair share of CPU time if it becomes runnable before the next scheduling decision is made. Each time a process uses only a fraction, f, of its quantum, the scheduler compensates the process by temporarily increasing its tickets until the next time the it is chosen by the scheduler. When compensated, a process holds an *effective* number of tickets equal to $T_p(1/f)$. For example, a process with 10 tickets that yields the CPU after using $1/2$ of a quantum will hold 20 effective tickets, enabling it to be chosen twice as likely as it would be without compensation tickets. This type of technique—dynamic ticket adjustments to achieve specific behavior without sacrificing the overall goals of proportional-share scheduling—is the basis for the mechanisms described in this paper.

3 Hybrid Lottery Scheduler Design

We extended lottery scheduling to be more responsive to interactive applications and to reduce kernel lock contention using techniques borrowed from the FreeBSD scheduler. Further, user feedback prompted us to provide semantics similar to the UNIX `nice` utility. The challenge was to improve performance without squandering the desired features of proportional-share scheduling. We describe our extensions resulting in a *hybrid lottery scheduler* in the following sections.

3.1 Kernel Priorities

Processes that block in the kernel often hold shared kernel resources locked until they wake and leave the kernel, such as when a process holds a buffer locked while it waits for disk I/O to complete. FreeBSD schedules processes asleep in the kernel with a higher priority than processes which exhaust their quanta at user level so that they will release these resources sooner, reducing the chance that other processes will find these resources in use [14]. The FreeBSD scheduler implements this by temporarily assigning static kernel priorities to processes after blocking in the kernel so that they will be preferentially scheduled upon waking. These kernel priorities are ordered in importance of the resource held. For example, a process holding a vnode locked will have a higher priority than a process holding a buffer waiting for disk I/O because vnodes have been deemed a more contended or important resource.

In lottery scheduling, a blocked process could temporarily transfer its tickets to the process that holds the desired resource, encouraging it to run and release the resource sooner. This technique was originally introduced to solve priority inversion [21]. However, it is complex to retrofit a kernel such as FreeBSD to perform ticket transfers at each point where a process may block on a kernel resource because of the number of places where new code would have to be added and validated. We cannot simply interpose ticket transfers within the kernel sleep and wakeup functions, because the arguments to these functions do not encapsulate enough information to know to which process tickets should be transferred. Further, this approach incurs more overhead than the FreeBSD scheduler. A process that needs a locked kernel resource will find the resource in use by another process, transfer its tickets to that process, block to enable a new lottery, and eventually recover its transferred tickets when it wakes up. Instead, by preferentially scheduling processes that wake up holding kernel resources, kernel priorities reduce outright the chance that resources are found locked. This reduces the number of context switches because processes contending for these resources will block less often.

Rather than introduce in-kernel ticket transfers to address kernel resource contention, we preferentially schedule processes holding kernel resources, similar to the FreeBSD approach. In detail, we maintain a list of processes that wake up after being blocked on a kernel resource. This list is sorted by the kernel priority of the resource that each process was blocked on. When making a scheduling decision, we run the standard lottery scheduler algorithm if the list is empty. If not, we choose the first process on this sorted list, emulating the behavior of the FreeBSD scheduler.

We lose the probabilistic fairness of lottery scheduling by choosing processes outside of the standard lottery scheduling algorithm. We recover this fairness with a variation of compensation tickets (see Section 2.2). We track the total time that each process has run between two successive selections by the standard lottery scheduling algorithm. During this period the process may go to sleep several times as it blocks in the kernel. When the process gets descheduled in user space by the time slice interrupt, the scheduler computes its compensation tickets based on this total time. If the process ran longer than one time quantum, an appropriate number of *negative* compensation tickets is temporarily assigned until the process is chosen to run again. These negative compensation tickets make the process *less* likely to be chosen by the scheduler. For example, consider a process with 10 tickets that overran its time quanta by 25%. The scheduler will negatively compensate the process by 2 tickets so that it will hold only 8 effective tickets until the process is chosen by the lottery. In detail, effective tickets $= T_p(1/f) = 10(1/1.25) = 8$.

Sometimes a process spends very little time in user space because it continually makes blocking system calls, making it unlikely that the time slice interrupt will deschedule it. As described, when the process is eventually descheduled by the time slice interrupt, it will receive a large number of negative compensation tickets, causing it to wait a long

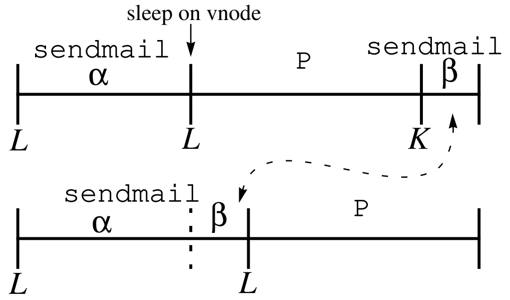

Figure 1: This figure shows processes being scheduled as time moves from left to right. L indicates a scheduling decision by the standard lottery scheduling algorithm while K shows where a process holding a kernel priority runs without having been chosen by the standard lottery scheduling algorithm.

time before running again. To avoid this, we force the process to relinquish the CPU in user level as soon as it has consumed more than one time quantum and is not executing in the kernel. This way the process will incur a small number of negative compensation tickets more frequently.

Figure 1 shows one way to think about kernel priorities in the hybrid lottery scheduler. On the top line `sendmail` is chosen by the lottery scheduler and after α time goes to sleep waiting for locked vnode, perhaps for the mail spool directory. Meanwhile, another process, denoted `P`, gets chosen by the lottery scheduler. When the vnode becomes available, `sendmail` is chosen from the kernel priority list (at K) and runs for some time, β, at which point the time slice interrupt occurs. The hybrid lottery scheduler grants `sendmail` compensation tickets at the end of β exactly as if `sendmail` was able to acquire the vnode lock without blocking and had run contiguously for time $\alpha+\beta$ as shown on the bottom line. The scheduler is in the same state at the end of both lines, showing that kernel priorities only temporarily violate the proportional-share properties of lottery scheduling.

3.2 Abbreviated Quanta

Lottery scheduling succeeds at scheduling processes which never voluntarily relinquish the CPU in proportion to the number of tickets that they hold. Interactive jobs, such as editors and shells, spend most of their time idle, and thus for them we are not concerned with the rate at which they consume CPU time, but instead with how responsive they are to user input. Upon becoming runnable from receiving input, we desire the time it takes for the scheduler to choose the process, known as the *dispatch latency*, to be small. Since lottery scheduling does not distinguish between CPU-bound and interactive jobs, it often fails to schedule interactive jobs first, causing noticeable "choppiness." This and the following sections describe enhancements to lottery scheduling to improve responsiveness.

Under the FreeBSD scheduler, a process waking up after blocking in the kernel (such as when a process waits for an I/O event) will preempt the running process before its quantum expires. The process which woke up has a kernel priority and will be chosen to run next unless multiple such tasks arrive at once. This behavior, which we call *abbreviated quanta*, is desirable if the sleeping process is interactive, for example, an `emacs` process that has just received a keystroke. Further, there are instances in which FreeBSD will *conditionally* preempt the running process. One such instance is when a process receives a signal and has a higher priority than the running process. Lottery scheduling, however, does not preempt the running process when another process wakes up or receives an event[1].

Our hybrid lottery scheduler forces a context switch when a sleeping process wakes up or when a process receives an event such as a signal. To ensure that the preempted process will receive the processor in proportion to the number of tickets that it holds, the scheduler awards the preempted process compensation tickets based on the fraction of the time slice it used (see Section 2.2).

We note a synergy between kernel priorities and abbreviated quanta. Without kernel priorities, abbreviated quanta will preempt running processes when a process wakes up, but will not guarantee that processes that were blocked in the kernel will be chosen ahead of processes executing user-level code. Without abbreviated quanta, kernel priorities will schedule processes blocked on kernel resources ahead of those that are not, but will not preempt processes before their quanta expire.

3.3 Windowed Ticket Boost

Since interactive processes generally use a fraction of their quanta, compensation tickets make them more responsive by temporarily boosting their effective tickets. Unfortunately, compensation tickets cannot be relied on to provide adequate responsiveness. If a CPU-bound job gets preempted by another process due to abbreviated quanta, then it will also receive compensation tickets, putting it on equal footing with interactive processes.

Consider the interactive application `emacs` running in an X window while a CPU-bound job runs in the background. At the outset, X runs while `emacs` sleeps waiting for a keystroke. Earlier, the CPU-bound job was preempted and received compensation tickets. When we press a key, `emacs` preempts X and runs immediately because we have implemented abbreviated quanta and `emacs` holds a kernel priority. After processing the keystroke `emacs` goes back to sleep. Both X and the CPU-bound process are runnable and neither have a kernel priority because they were both preempted running user-level code. Up to this point, both the

[1] Other implementations of lottery scheduling may have implicitly leveraged abbreviated quanta on the operating systems they were implemented on. However, we know of no work demonstrating the importance of using abbreviated quanta with lottery scheduling.

FreeBSD and hybrid lottery schedulers behave the same. The FreeBSD scheduler will almost certainly pick X to run because, having accrued less CPU time than the CPU-bound process (which always runs when X and `emacs` are sleeping), it resides in a higher priority runqueue. Because neither X nor the CPU-bound process have kernel priorities, the lottery scheduler will perform the standard lottery algorithm against them. Sometimes the CPU-bound process will be chosen and run for one quantum (100 ms). When this occurs, X delays in updating the `emacs` window, resulting in choppiness.

We have experienced this scenario on a workstation running a preliminary version of our hybrid lottery scheduler with only abbreviated quanta and kernel priorities implemented. When holding a key down (equivalent to pressing 120 keys/s) over approximately 7 s, we generated 825 key events, of which 23 where delayed when the CPU-bound process was chosen to run before X, perceptibly delaying window updates. In summary, delayed dispatching of an interactive process is likely to happen if it does not hold a kernel priority (because it was preempted at user level) and a CPU-bound process was preempted and granted compensation tickets. This will also occur in deterministic proportional-share schedulers like stride scheduling.

To solve this problem we make tickets serve a dual purpose depending on the nature of the process. For CPU-bound jobs, tickets correspond to the rate of CPU time consumed, as before. For interactive jobs, tickets determine how responsive the processes are. If we increase the tickets held by interactive jobs, then we increase the chance that they will be chosen to run before CPU-bound jobs, shortening dispatch latency. Hence, if we can identify processes as interactive, we can boost their tickets to make them more responsive.

Our central observation is that most interactive processes do not use CPU time in proportion to their ticket allocations because they often block for long time periods on user input. By tracking the total number of base tickets in the system and total number of tickets per currency at regular intervals, we can find the processes that have received less CPU time than deserved. Transparently to the user, we classify these processes as interactive and boost their effective tickets to make them more responsive. There is no fear that they will use more than their allotted share because we only boost the tickets of exactly those processes using less than their share.

We desire the dispatch latency of interactive jobs to be below the minimum latency that humans can discern. When only one interactive job is runnable, abbreviated quanta and an arbitrarily high ticket boost will ensure this. However, when multiple interactive jobs are simultaneously runnable, there is a chance that the dispatch latency of some of them will be noticeable. We can reduce this chance by influencing the scheduling order of multiple interactive jobs to minimize their average dispatch latency. It is well known that response time is minimized by scheduling tasks via shortest-processing-time first (SPT) [3]. If we knew how much processing time would be used by the interactive jobs in handling an event (such as a keystroke) before going back to sleep, we could schedule them optimally to minimize dispatch latency. Since we do not have this information, we approximate SPT by using history. We make the ticket boost inversely related to the fraction of deserved CPU time used. Specifically, we boost the effective tickets of a job that used almost none of its allotted CPU time by a large factor, such as 10,000, making it many times more likely to be chosen as before. A job that used its deserved share receives no boost. Finally, we use interpolation to derive the boost of a job whose fraction of deserved CPU time was between these extremes. Hence, interactive jobs that barely use the CPU are more likely to be scheduled before interactive jobs that use a moderate, but still less than deserved, amount of CPU, approximating SPT.

Jobs are rarely continuously interactive or CPU-bound. For example, an `emacs` process may be interactive most of the time, but then become CPU-bound for some time as it executes elisp code. To adapt to these types of processes, we look at a sliding window interval of history when checking to see if a process is using its fair share of the CPU. We would like the scheduler to be "agile," in the sense that if a previously interactive job became CPU-bound, we restore its ticket allocation quickly, and contrariwise, if a previously CPU-bound job became interactive, we quickly boost its ticket allocation. This argues for a small window size, *e.g.,* on the order of one second. However, if the window size is too small, there is a chance that a CPU-bound job will not run during this window simply because the machine is heavily loaded. If this were to occur, we would mistakenly label such a process as interactive. While this parameter may require tuning for different environments, we found that a window interval of ten seconds works adequately for making a bimodal process such as `emacs` responsive within a few seconds after going from a CPU-bound to interactive stage, when competing against completely CPU-bound processes.

We call this extension *windowed ticket boost*. Our motivation was the priority decay present in the stock FreeBSD scheduler. Rather than lower the priority of processes consuming CPU time, we raise the priority of those that do not. Since we only boost the priorities of only those processes that are not receiving their deserved CPU time, we do not violate the proportional-share semantics. The combination of the abbreviated quanta and windowed ticket boost extensions together provide the responsiveness of the stock FreeBSD scheduler.

3.4 `nice` Emulation

All UNIX variants have a utility called `nice` that enables a user to vary the priority of her processes from -20 (highest) to $+20$ (lowest) relative to all other processes in the system. After deploying our hybrid lottery scheduler on production systems, users complained that ticket adjustments alone did not give them the type of control provided by `nice`. Since `nice` priority adjustments only allow a general favoring or shunning of process priority and do not provide absolute, or even predictable guarantees [18], we do not attempt to exactly emulate `nice` semantics, but rather we approximate the aspects of its behavior relied on by its users; specifically, that a user (system administrator) can downgrade (upgrade) one of her processes relative to all others without downgrading (upgrading) the rest of her processes. Of course, the targeted process will no longer receive the CPU utilization allocated by its tickets.

A naïve approach to emulating `nice` semantics is to only modify the tickets of the target process according to a mapping of `nice` values to tickets. This fails because only the relative execution rate of a user's own processes are affected. If a user has only one CPU-bound process running that is `nice`'d to lowest priority ($+20$), the exchange rate will still assign the process all of the base tickets funding the user's processes (see Section 2.2). Another incorrect approach adjusts only the base tickets funding a user's processes with a mapping of `nice` values to base tickets. In this instance, if a user lowers the priority of one process with `nice`, all of the user's processes will have lower priority relative to the rest of the processes on the system. One could instead create a new currency for each `nice`'d process, funded with a number of base tickets reflecting the `nice` value. Unfortunately, a user can run a large number of `nice`'d processes and get an unfair share of CPU time. We could not take that many base tickets from her normal currency, because then her normal processes would not get their share of CPU time if any `nice`'d process momentarily went to sleep. Further, without adding more complexity, we would run out of base tickets to take from the user's currency if the user started many `nice`'d processes.

Our solution adjusts both tickets and base tickets to achieve the `nice` function. We raise (lower) a process's number of tickets according to the process's `nice` value, ensuring that the process will have more (less) priority relative to other processes owned by the user. When a scheduling decision is made, we convert the tickets held by the process into base tickets and make the following adjustments to the allocated base tickets. If the process has been `nice`'d to have a lower priority, we force the number of base tickets held by the process to be *at most* a threshold determined by the `nice` value. Likewise, if the process has been assigned a higher priority with `nice`, we force the number of base tickets held to be *at least* a specific threshold. This thresholding ensures that a `nice`'d process will have more or less priority relative to processes outside of the user's currency.

4 Implementation

Our system is divided into two parts. Most of the code resides in the kernel files implementing the hybrid lottery scheduler. The rest of the system consists of small user-level programs that make a new *lott_sched()* system call to adjust or query scheduling parameters.

4.1 Kernel Functionality

We first describe critical pieces of the FreeBSD scheduler necessary for understanding our scheduler implementation and for understanding the benchmarks in Section 5. After becoming runnable, processes are put onto the appropriate runqueue by the assembly language routine *setrunqueue()*. Processes are removed from runqueues when chosen by the scheduler. The assembly language routine *cpu_switch()* saves process context, chooses the next process to run, and switches to that process (or idles if no processes are runnable).

We decoupled the scheduling policy from mechanism by placing a function call from within *cpu_switch()* to *lott_choose_next_runner()*, a C function which implements the hybrid lottery scheduling algorithm. We assign processes a default number of tickets when they are created by *fork1()*. The *setuid()* system call (*cf.,* `login`) was modified to create a new user currency funded with a default number of base tickets. Naturally, we do not create a new currency if the user is already logged onto the machine. We reference count the processes owned by a user so that we can garbage collect her currency when all of her processes terminate. Although Waldspurger's original framework allows currencies to be nested indefinitely, our implementation only distinguishes between the base and per-user currencies.

Any lottery scheduler with compensation tickets and currencies will be more computationally expensive than the FreeBSD scheduler. A FreeBSD scheduling decision is a fast $O(1)$ operation because the scheduler simply removes a process from the head of the first non-empty runqueue. Our hybrid lottery scheduler implementation employs an $O(n)$ algorithm, where n is the number of runnable processes[2]. As we sought to achieve performance comparable to the FreeBSD scheduler, we expended much effort in optimizing our implementation. Since floating point operations are not permitted while running in the FreeBSD kernel, we do nearly all of our computations with fixed-point integers. We use 32-bit integers so that we can utilize hardware instructions for most of the arithmetic. When we needed to use 64-bit integers, we wrote an in-line assembly routine which issues the "32-bit times 32-bit to 64-bit" hardware instruction after discovering that `gcc` inefficiently compiled this operation. We also defer work, use in-line

[2] An $O(\lg n)$ lottery scheduling algorithm exists, but as we rarely see large n, we believe its extra overhead outweighs its lower computational complexity.

```
lott_choose_next_runner() {
    if(empty(kernel_pri_list && lottery_list))
        return 0; /* idle */
    if(!empty(kernel_pri_list))
        return head(kernel_pri_list);
    foreach(process on lottery_list) {
        compute effective tickets based on
          ticket_boost or frac. of quanta used;
        convert effective tickets to base tickets
          based on user's exchange rate;
        use threshold if process is nice'd;
        update our running count of base tickets;
    }
    pick random number from 1 to total base
      tickets;
    foreach(process on lottery_list)
        if(process holds winning base ticket)
            return process;
}
```

Figure 2: Pseudo-code for *lott_choose_next_runner()*.

functions, and aggressively cache the computed values described further below.

While in theory the number of tickets or base tickets allocated to a process or user, respectively, can be any positive integer, implementation efficiency encourages the range of values to be bounded. We bound the ratio of minimum to maximum ticket and base ticket shares from 1–100 as we believe two orders of magnitude expresses sufficient resolution. We carefully set the range of tickets and base tickets from 1–100 and 100–10,000 respectively so that we would not incur overflows and underflows during our computations with 32-bit fixed-point integers. By default, processes and users start with 100 tickets and 1,000 base tickets respectively.

To help us optimize and understand the system, we instrumented both the FreeBSD and our hybrid lottery schedulers to provide us with detailed profiling information. We can measure the last *n* scheduling events, including when and for what reasons processes go to sleep, and time quanta expirations.

To implement windowed ticket boost, we maintain circular arrays that track whether a process is getting less than its share of CPU time. Every second we update a circular array that holds the number of base tickets in the system over the last ten seconds. We also update similar arrays which track the number of tickets held in each user's currency. Updating the circular arrays more often would give us more accurate data at the expense of higher overhead. For a particular process, *setrunqueue()* derives the amount of time it deserved to run from the values in the circular arrays. Then this function finds the fraction, f, of deserved time actually used over the ten second interval. If f is greater than one, the process is getting at least its fair share of the CPU. If not, we deem the process interactive and derive a scaling factor called `ticket_boost` which we later multiply against the process's effective tickets. While `ticket_boost` can simply be set to $1/f$, raising this to a power will dramatically increase our system's ability to ensure SPT ordering for interactive jobs while still using a probabilistic scheduler. Although this parameter may require tuning for different environments, our current implementation uses $(1/f)^4$. To prevent an overflow in a later computation, we clip `ticket_boost` at 10,000.

If a process has been asleep for more than ten seconds, we discount it from the above measurements because it is not actively competing for CPU time. When such a process wakes up, *setrunqueue()* assigns it the maximum `ticket_boost` since it most probably has not received its share of CPU time.

The heart of the hybrid lottery scheduling algorithm resides in *lott_choose_next_runner()*, illustrated in Figure 2. The lists `kern_pri_list` and `lottery_list` contain runnable processes that do and do not hold kernel priorities, respectively. If a process used less CPU time than it deserved, we compute the process's effective tickets by multiplying the number of tickets that the process holds by its `ticket_boost`. If not, we are dealing with a CPU-bound process and need to assign compensation tickets. We compute its effective tickets by dividing the number of microseconds in one time quantum by the number of microseconds used by the process during the last time that it was scheduled and multiplying this value by the number of tickets that the process holds. We compute the base tickets held by the process by dividing the number of base tickets that fund the user's currency by the number of tickets in all of the user's runnable processes and multiplying this exchange rate by the process's effective tickets.

When a process is `nice`'d, we change the number of tickets that it holds so that it will have more or less priority relative to the rest of the user's processes. Since tickets range from 1 to 100, a given `nice` value equals $10^{\frac{1}{20}(-\mathtt{nice}+20)}$ tickets. After converting a process's tickets to base tickets in *lott_choose_next_runner()*, we apply the `nice` base ticket threshold. Since base tickets range from 100 to 10,000, the mapping from `nice` values to base tickets is $10^{\frac{1}{20}(-\mathtt{nice}+20)+2}$. If the process has been `nice`'d to have a lower priority, we force the number of base tickets held by the process to be *at most* this value, and vice-versa if the process has been assigned a higher priority with `nice`.

4.2 User-level Programs

Users adjust and query lottery scheduling parameters via a number of user-level programs. `set_tickets` changes the number of tickets held by a process while `show_tickets` displays a `ps`-like listing of a user's processes and their ticket allocations. A user executes a given program with a specific number of tickets with `run_tickets`. Root uses `set_funding` to set the number of base tickets that fund a

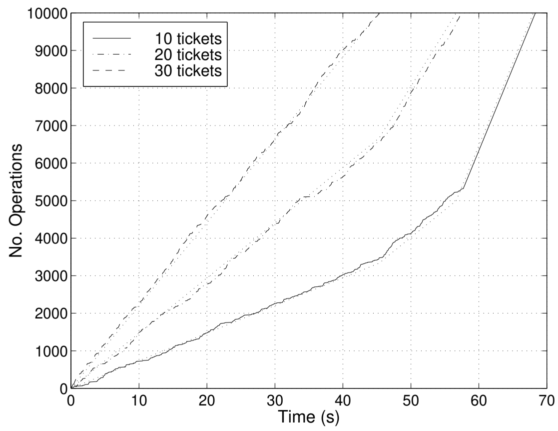

Figure 3: This figure demonstrates the progress of three CPU-bound processes under the hybrid lottery scheduler. Each operation on the Y-axis represents a fixed amount of work. Also shown are straight dotted lines representing ideal processor utilization. Notice that when a process finishes, after completing 10,000 operations, the remaining processes execute faster.

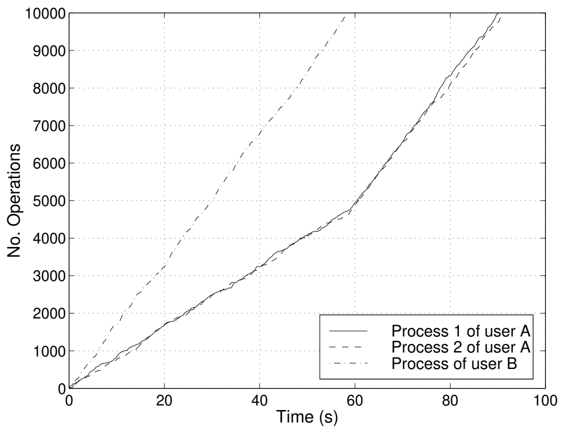

Figure 4: This figure shows the progress of three CPU-bound processes under lottery scheduling. The top curve represents a process run by one user while the bottom curves represent two processes run by another user. When the first process finishes, the other processes have made 47% and 46% progress toward completion (50% each is ideal).

user's currency while show_funding displays the number of base tickets that fund a user's currency. Today the funding commands only apply to logged in users; soon we will record the number of base tickets funding a user's currency in the user's account record. Finally, lott_chuser takes a process and places it under another user's currency, which is useful for moving processes like the X Window System which run as root under the currency of the process's primary user.

5 Evaluation

We first demonstrate some properties of proportional-share resource management that the FreeBSD scheduler lacks. We then show that the hybrid lottery scheduler is more responsive and helps reduce waiting time for processes blocked on kernel resources relative to our initial lottery scheduler implementation. Finally, we show that the hybrid lottery scheduler incurs minimal overhead relative to the FreeBSD scheduler.

Unless otherwise noted, all results in this section were obtained from the personal computer called partita. This machine has one 200 MHz AMD K6 (Pentium compatible) processor, 64 MB of main memory, and 3 GB of ultra-wide SCSI storage. No tests caused the machine to page.

5.1 Flexible Execution Rate Control

Here we demonstrate the features that we have gained by replacing FreeBSD's decay usage scheduler with lottery scheduling. We refer the reader to Waldspurger's thesis [20] for a wide range of additional examples and an extensive analysis of lottery scheduling.

We demonstrate the ease with which a user can control the execution rate of her programs in Figure 3. This figure shows three CPU-bound processes assigned tickets in a 3:2:1 ratio making progress at approximately the same ratio. Curious as to how hard this is to achieve using the FreeBSD scheduler, we found through trial and error a number of nice values that come close. The nice values that we discovered, +10, +5, and 0, are not intuitively mappable to our goal of 3:2:1. Naturally, any other ratio would be equally difficult to implement. Further, while lottery scheduling maintains the 3:2:1 ratio irrespective of system load, the FreeBSD scheduler unpredictably schedules these processes if other jobs are running.

We demonstrate user workload insulation in Figure 4. Despite one user running two CPU-bound processes, the second user is able to receive approximately twice the throughput from one CPU-bound process.

5.2 Performance

In this section we show the utility of windowed ticket boost, abbreviated quanta, and kernel priorities.

The minimum latency that humans can discern varies between 50–150 ms depending on the individual [15]. We set up the following experiment to test the responsiveness of our hybrid lottery scheduler. We run one completely CPU-bound process against a bimodal process, both with the same number of tickets. This bimodal process is CPU-bound for the first 15 seconds, interactive for the next 20 seconds, and then CPU-bound again for the remaining 15 seconds. In the interactive stage, the process repeatedly computes for approximately 5 ms and then sleeps for about 100 ms, simulating an editor such as emacs. While these

	Dispatch Latency (milliseconds)			
	0–15 s	15–35 s	35–50 s	22–35 s
Without Windowed Ticket Boost	29.85 (58.18)	9.13 (27.79)	27.24 (49.64)	10.59 (31.63)
With Windowed Ticket Boost	27.79 (58.46)	2.68 (12.84)	27.58 (50.92)	0.05 (0.02)

Table 1: This table presents the data from Figures 5 and 6 numerically. We show the average dispatch latency (and standard deviation in parentheses) in milliseconds for the bimodal application for each of its three stages of execution, with and without windowed ticket boost. Between 15–35 seconds we perform better with windowed ticket boost, although some of this time is spent discovering that the process became interactive. In the fourth column we show the dispatch latency during seconds 22–35, which is when the scheduler has identified the process as interactive and is applying a substantial ticket boost.

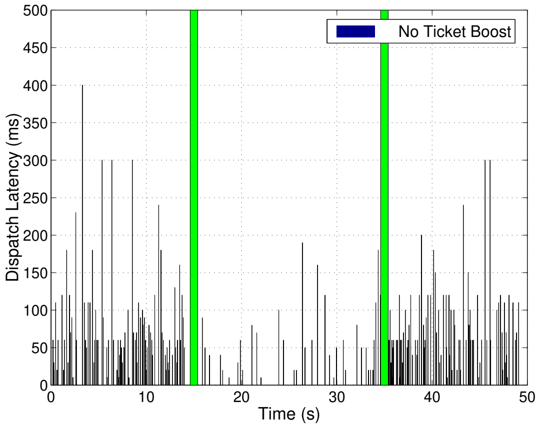

Figure 5: This figure shows the dispatch latency of a bimodal process competing against a CPU-bound process under our hybrid lottery scheduler with windowed ticket boost disabled. From seconds 15 to 35 (marked by the thick bars) the process is interactive, yet the scheduler is unable to schedule it in a timely manner.

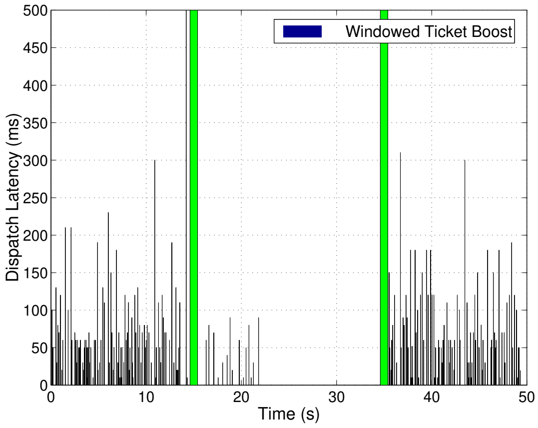

Figure 6: This figure shows the dispatch latency of a bimodal process competing against a CPU-bound process under our hybrid lottery scheduler with windowed ticket boost enabled. From seconds 15 to 35 (marked by the thick bars) the process is interactive and after a short adjustment period, the process exhibits excellent dispatch latencies.

two processes compete, a third process wakes up every 50 ms and goes back to sleep immediately simply to cause occasional rescheduling events as would be triggered by X or background processes in an actual workload.

The most important metric for responsiveness is dispatch latency after user input, that is, the time elapsed between when a process becomes runnable in *setrunqueue()* and when it is chosen to run by *lott_choose_next_runner()*. We ran the described benchmark with and without windowed ticket boost as shown in Figures 5 and 6, and Table 1. During the interactive phase (seconds 15–35) the scheduler with windowed ticket boost disabled is unable to schedule the process fast enough to avoid discernible choppiness. Although the dispatch latency is somewhat lower than during the CPU phases due to compensation tickets, there are still many points over 100 ms. This occurs because the CPU-bound process occasionally gets preempted, earning compensation tickets which puts it on equal footing with the interactive process. When windowed ticket boost is enabled, we see an adjustment period for about seven seconds after the bimodal process becomes interactive. As the 10 second sliding window moves forward, the process's `ticket_boost` increases until it is always favored by the scheduler when it becomes runnable. From seconds 22 to 35, the dispatch latency is in the tens of microseconds, far below human perception. Once the process becomes CPU-bound again, the system quickly adapts by lowering the process's `ticket_boost` parameter, ensuring that it will not get more than its share of CPU time.

To ensure that windowed ticket boost does not negatively effect the completely CPU-bound process, we show the progress of both processes with windowed ticket boost enabled in Figure 7. Between seconds 0–15 both processes consume CPU time at the same rate. When the bimodal process becomes interactive between seconds 15–35, the CPU-bound job nearly doubles its throughput, getting at least its share. Finally, when the interactive process becomes CPU-bound from seconds 35–50, the sliding window quickly adapts to lower its `ticket_boost` so that it does not dominate the CPU. The bimodal job only needs to consume the CPU at a faster rate for less than one second before losing its boost. We omit the same graph with windowed ticket boost disabled because it is nearly identical.

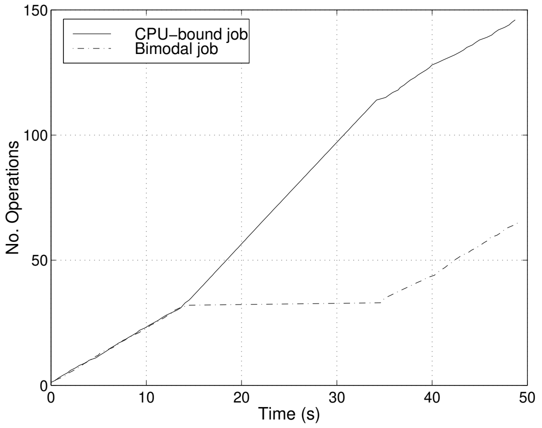

Figure 7: This figure shows the computational progress of the CPU-bound and bimodal jobs with windowed ticket boost enabled. When the bimodal job enters its interactive stage between seconds 15 to 35, the CPU-bound job makes progress 1.9 times faster than when both jobs are CPU-bound.

Without abbreviated quanta, the dispatch latency for the bimodal process when it is interactive would never be below about 100 ms. Note that the bimodal process is always chosen by the scheduler at the start of a new quantum. After it goes to sleep, the scheduler chooses the CPU-bound process. Unless abbreviated quanta is employed, it will consume CPU time until the time slice elapses.

To show the utility of kernel priorities, we instrumented the FreeBSD kernel to provide us with statistics concerning kernel lock contention. Every 15 minutes we took a 30 second log of each time a process went to sleep (via *tsleep()*), for what purpose it went to sleep, and for how long. We ran one set of numbers with kernel priorities enabled, and one set without, each for a 24-hour period on `soda`, a busy production machine on which we have deployed our hybrid lottery scheduler. Although there is some uncertainty in our measurements due to our inability to completely control the workloads over both runs, we took care to ensure that the workloads were roughly equivalent by comparing the number of users logged in, the context switch rate, and the paging activity.

Over a 24-hour period, there were about 40 distinct reasons why processes went to sleep. We present an abbreviated version in Table 2. We omit sleeps that occur less than 100 times and sleeps initiated by processes that neither held a lock before going to sleep nor held one upon waking (such as when a process goes to sleep on a timer event). We compare the percent reduction in sleep frequency, sleep duration, and weighted (by frequency) sleep duration when running with kernel priorities enabled. Due to latency variations in network and terminal-related events which cause long-tailed wait distributions, we used the sleep duration median to generate the data presented.

		% improvement with kernel priorities		
wmesg	type	freq.	duration	weighted
`biowait`	disk	4.43	20.38	23.91
`ffsfsn`	fs	8.53	25.26	31.63
`getblk`	fs	39.37	39.15	63.11
`pipdwt`	ipc	75.97	27.33	82.54
`piperd`	ipc	44.51	−1487.28	−780.72
`sbwait`	net	−14.11	63.39	58.22
`swpfre`	vm	−101.97	52.75	4.57
`swread`	vm	16.09	34.53	45.06
`ttywai`	tty	−9.25	53.08	48.74
`ttywri`	tty	11.69	53.74	59.15
`ufslk2`	fs	59.43	49.63	79.57
`vnread`	fs	−11.62	27.38	18.94
`wait`	proc	18.12	15.94	31.18

Table 2: This table shows the effect of kernel priorities. We show the places within the kernel where processes went to sleep, categorized as those related to the network, disk I/O, the file system, inter-process communication, the virtual memory system, terminal I/O, and process administration. We compare the percent reduction in sleep frequency, sleep duration, and weighted (by frequency) sleep duration when running with kernel priorities enabled.

With kernel priorities, processes holding kernel resources are preferentially scheduled, reducing the duration of sleeps, which in turn reduces the frequency of sleeps because there is a smaller window of time that a process will find a resource in use. These trends are apparent although we made no effort to artificially increase kernel resource contention. One major anomaly is the wait duration for processes going to sleep waiting for data in a pipe read (`piperd`). In the run with kernel priorities disabled, we saw an unusually large number of very short sleeps from the process `ssh` (a secure telnet shell) on a pipe read during one 30 second interval. We believe that this activity caused this anomaly.

5.3 Overhead

We measure scheduling code fragments to quantify scheduling overhead. To obtain accurate measurements, we employ the `RDTSC` (Read Time-Stamp Counter) instruction which reads a counter incremented every clock cycle. In the following figures, error bars represent 95% confidence intervals. The number of independent runs for each experiment is listed with the experiment.

The two most common scheduling operations in both the FreeBSD and hybrid lottery schedulers are *cpu_switch()* and *setrunqueue()*. Under the FreeBSD scheduler, *cpu_-switch()* makes a scheduling decision and performs a context switch. In the hybrid lottery scheduler, *cpu_switch()* performs a context switch after calling *lott_choose_next_-*

		FreeBSD		Lottery	
		mean	std. err.	mean	std. err.
1 process	*cpu_switch()*	2.86	0.011	7.25	0.019
	setrunqueue()	0.57	0.003	17.36	0.038
25 processes	*cpu_switch()*	4.18	0.012	14.37	0.124
	setrunqueue()	0.66	0.003	17.79	0.037
50 processes	*cpu_switch()*	4.32	0.012	22.95	0.172
	setrunqueue()	0.64	0.004	17.59	0.062
75 processes	*cpu_switch()*	4.74	0.014	26.63	0.153
	setrunqueue()	0.83	0.003	17.08	0.081
100 processes	*cpu_switch()*	7.37	0.066	36.28	0.241
	setrunqueue()	0.69	0.009	16.63	0.102

Table 3: This table presents the data from Figures 8 and 9 in numerical format. The times are in microseconds. *cpu_switch()* makes a scheduling decision and performs a context switch. *setrunqueue()* marks a process as runnable, and in the hybrid lottery scheduler also computes `ticket_boost` for the windowed ticket boost.

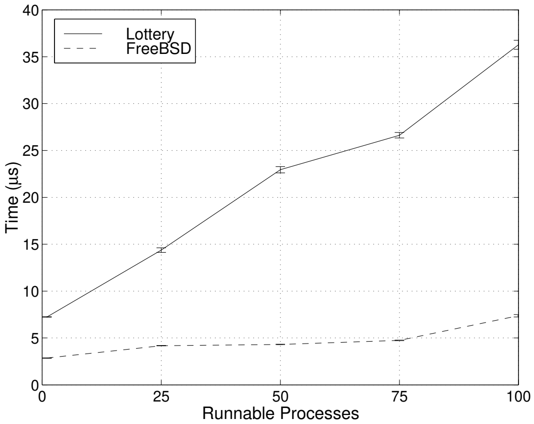

Figure 8: This figure shows the average number of microseconds (out of at least 1,000 measurements) to perform a context switch via the *cpu_switch()* function while varying the number of runnable processes.

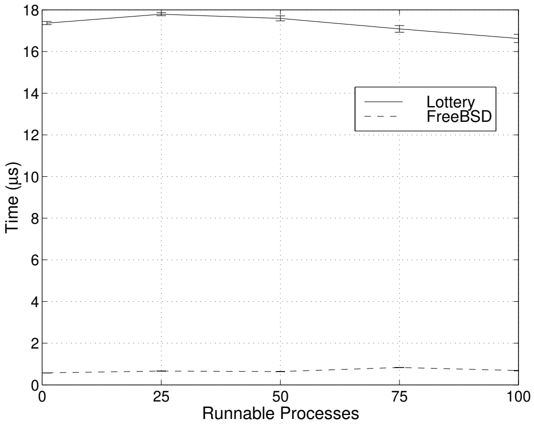

Figure 9: This figure shows the average number of microseconds (out of at least 1,000 measurements) to make a process runnable via the *setrunqueue()* function while varying the number of runnable processes.

runner() which makes a scheduling decision. Figure 8 shows the time it takes to run *cpu_switch()*[3] while varying the number of runnable processes. Naturally, we include the time in *lott_choose_next_runner()* in the hybrid lottery scheduler measurements. As described in Section 4, our scheduling algorithm is $O(n)$ in the number of runnable processes while the FreeBSD scheduler is $O(1)$. This difference in algorithmic complexity is apparent in these results.

Figure 9 shows the time it takes to execute *setrunqueue()*, which makes a process runnable. This function is short in the FreeBSD scheduler. In the hybrid lottery scheduler, we also compute the process's `ticket_boost`, which requires iterating through a 10 element array. While the overhead is substantially higher in the lottery scheduler, this function is $O(1)$ in both schedulers. We do not know why neither curve is entirely flat. Table 3 presents the data from both Figures 8 and 9 in numerical format.

The preceding experiments uncovered measurable differences between the FreeBSD and hybrid lottery schedulers. Now we determine how appreciable these differences are on the scale of compute-bound applications.

To measure the throughput of batch processes we use `rc564`, a program that tries to find the solution to RSA's 64-bit secret-key challenge. To exacerbate the effect of our added overhead while running `rc564`, we increase the number of context switches that occur by running up to 10 processes called `interactive` at the same time. An `interactive` process continually goes to sleep for the shortest time possible and causes a context switch upon

[3] Other work often records context switch time as the elapsed time between passing control from user space in one process to user space in another. Thus our reported times, which record just the elapsed time of *cpu_switch()*, may appear low.

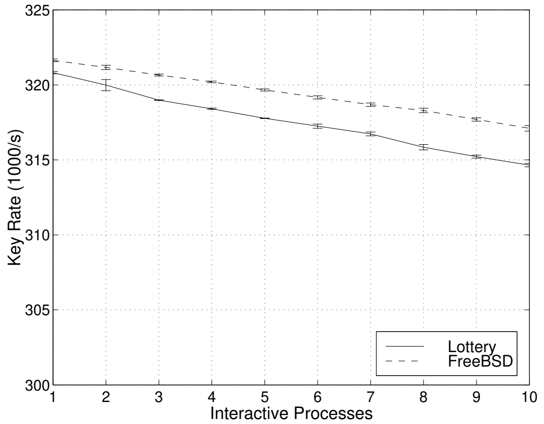

Figure 10: This figure shows the average number of keys tried per second (out of 5 trials) by `rc564` while varying the number of `interactive` processes. Note that the Y-axis begins at 300,000 keys/s, exaggerating the apparent differences. The performance under the hybrid lottery scheduler is always within 1% of the FreeBSD scheduler.

waking due to abbreviated quanta. One `interactive` process generates 128 context switches per second while 10 generate 626 context switches per second. Adding an interactive process adds approximately 55 context switches per second. The throughput of `rc564` versus the number of `interactive` processes is shown in Figure 10. We note that as more `interactive` processes are run, the performance of `rc564` under the FreeBSD and hybrid lottery schedulers worsens and diverges. In all runs, `rc564` under the hybrid lottery scheduler is less than one percent slower than under the FreeBSD scheduler.

Curious as to what the context switch rate is on busy systems, we measured `wcarchive`, the world's largest and busiest FTP site[4]. The average number of context switches over a 30 second interval on this site was 2589 per second. As the previous experiment did not show a large difference between the FreeBSD and hybrid lottery schedulers, we ran a program which simply loops and maintains a counter of how many loops it made for 5 minutes, while simultaneously running 100 `interactive` processes. These `interactive` processes pushed the number of context switches per second up to 5160 averaged over the run. In this very extreme test we were about 15% slower than the FreeBSD scheduler. If such an scenario realistically occurred, we could minimize our overhead at the cost of some accuracy by not computing `ticket_boost` on every call to *setrunqueue()*, but perhaps every 10th call.

[4]When we took this measurement in December 1997, `wcarchive` stored 142 GB on-line and supported up to and often reached 2750 simultaneous connections. `wcarchive` is located at ftp://ftp.cdrom.com/.

PID	USERNAME	PRI	NICE	SIZE	RES	STAT	TIME	WCPU	CPU	COMMND
555	jwm	92	0	808K	164K	RUN	0:17	16.34%	16.25%	rc564
553	peterm	90	0	7392K	8012K	RUN	0:18	16.28%	16.21%	xoopic
552	peterm	90	0	7392K	8012K	RUN	0:18	16.12%	16.06%	xoopic
550	peterm	90	0	7392K	7852K	RUN	0:18	16.12%	16.06%	xoopic
551	peterm	90	0	7392K	7864K	RUN	0:18	16.08%	16.02%	xoopic
554	peterm	89	0	7392K	8012K	RUN	0:18	16.05%	15.98%	xoopic

Table 4: This table shows the output from `top` while two users are running one and five CPU-bound processes respectively under the FreeBSD scheduler. The lack of load insulation enables `peterm` to obtain an unfair percentage of the CPU.

6 Experience

We have deployed our hybrid lottery scheduler on `soda.-csua.berkeley.edu` and `meeko.eecs.berkeley.edu`, two production machines. `soda` is the central machine for the Computer Science Undergraduate Association at UC Berkeley. `soda` is powered by one 200 MHz AMD K6 (Pentium compatible) processor, 256 MB of RAM, and 15 GB of ultra-wide SCSI storage. `soda` supports over 2400 shell accounts and often has over 150 unique users simultaneously logged on accessing USENET, reading mail, participating in chat rooms, and developing code. `soda` also manages over 200 mailing lists, and on an average day, completes roughly 170,000 `sendmail` transactions. Finally, `soda` runs a web server that receives approximately 50,000 accesses a day. `meeko` runs on one 200 MHz AMD K6, 128 MB of main memory, and 22 GB of ultra-wide SCSI storage. `meeko` belongs to the FreeBSD Users' Group at UC Berkeley. `meeko` runs a web server and mirrors part of `wcarchive` which is offered on its anonymous FTP server. In addition, `meeko` exports a filesystem via NFS. There are usually 5 users logged onto `meeko` actively developing code while a couple dozen users engage in a multi-user game (MUD). These systems have been running our hybrid lottery scheduler since December 1997 (1.5 years). That we have received no complaints is a testament to our implementation's stability and performance.

It is especially important to have load insulation on a machine like `soda` that supports a large user community on one processor. We show the load insulation properties of both the FreeBSD and hybrid lottery schedulers by looking at the output of the UNIX `top` utility while two users run the CPU-bound processes `xoopic` and `rc564`. `xoopic` is a particle-in-cell plasma simulation that calculates fields on a 2-D mesh using Maxwell's equations. Tables 4 and 5 show no load insulation under the FreeBSD scheduler, and reasonably accurate load insulation under the hybrid lottery scheduler.

Our latest version of the hybrid lottery scheduler incorporating windowed ticket boost has not been deployed to `soda` and `meeko` because they are used exclusively over networks of significant latency and thus would not appreciate the benefits offered by this extension.

PID	USERNAME	PRI	NICE	SIZE	RES	STAT	TIME	WCPU	CPU	COMMND
296	jwm	98	0	808K	392K	RUN	0:28	52.21%	48.71%	rc564
272	peterm	76	0	7392K	7544K	RUN	1:02	11.63%	11.63%	xoopic
275	peterm	65	0	7392K	7716K	RUN	0:57	9.61%	9.61%	xoopic
282	peterm	64	0	7392K	8032K	RUN	0:50	9.50%	9.50%	xoopic
274	peterm	55	0	7392K	7636K	RUN	0:57	7.90%	7.90%	xoopic
273	peterm	53	0	7392K	7600K	RUN	0:55	7.13%	7.13%	xoopic

Table 5: This table shows the output from `top` while two users are running one and five CPU-bound processes respectively under the hybrid lottery scheduler. `jwm` is able to receive about 50% of the CPU despite having only one runnable process.

7 Related Work

Process scheduling on time-sharing systems has been studied extensively [13, 11]. A number of fair-share schedulers fairly allocate CPU time to classes of processes over long time spans [12, 5]. Recently introduced proportional-share schedulers such as lottery scheduling [21] and EEVDF [17] strive for instantaneous fairness; that is, making fair scheduling decisions against only the currently runnable set of processes. Another proportional-share scheduler from Waldspurger *et al.* is stride scheduling, which deterministically schedules processes with higher throughput accuracy and lower response time variability compared to lottery scheduling [22]. Since they both employ the same ticket framework, our extensions to lottery scheduling are also applicable to stride scheduling. We choose to extend lottery scheduling over other schedulers because the core algorithm is simple.

Although tickets enable flexible resource control, it is often difficult for users to assign tickets among workloads to meet higher-level performance goals. Recent work from Sullivan *et al.* introduces application-specific "negotiators" that enable automatic ticket exchanges between processes desiring different resource allocations [19]. In other work, a feedback-driven reservation-based scheduler by Steere *et al.* monitors process progress to divine appropriate CPU-time allocations transparently to the user [16].

Arpaci-Dusseau *et al.* studied stride scheduling in the network of workstations context [1]. Part of their aim was to provide better responsiveness under mixed workloads. They award a sleeping (interactive) process exhaustible tickets that expire when it receives its fair share of CPU time. However, most interactive processes will never use their allocation because they are usually sleeping. For these processes, rather than strive for CPU-time fairness, we believe that dispatch latency should be minimized. Further, without modification, their system does not handle processes with interactive *and* compute phases. A process that has slept for a long time and wakes up will dominate the CPU for an extended duration in virtue of holding exhaustible tickets. Finally, their algorithm for computing exhaustible tickets assumes that the total number of runnable tickets is constant. In reality, this number fluctuates as processes are created and destroyed, and sleep and wake up.

8 Future Work

Hybrid lottery scheduling heuristically identifies and rewards interactive processes by how much of their allocated CPU time they consume. However, some interactive processes, such as those that render graphics, also consume moderate amounts of CPU. Evans *et al.* suggest several methods based on past user action and window manager cooperation for an operating system to recognize interactive processes [6]. We wish to incorporate these methods into our scheduler so that once recognized, these processes can be allocated more tickets and preferentially scheduled.

In Section 3.1 we argued that kernel priorities are more desirable than ticket transfers for encouraging processes to release kernel resources quickly. However, to our knowledge, the chosen ordering of kernel priorities has not been rigorously studied and thus may not provide optimal performance in all cases. Ticket transfers are more dynamic because they enable additive and transitive transfers from multiple blocked processes. If the kernel can identify the loaner and borrower when a kernel resource is under contention, the kernel can perform this *implicit* ticket transfer transparently to the user. We wish to compare the throughput of different workloads with implicit ticket transfers versus kernel priorities.

9 Conclusion

This work incorporates into a lottery scheduler the specializations present in typical operating system schedulers to improve interactive response time and reduce kernel lock contention. We began with a straightforward implementation of lottery scheduling which enabled control over process execution rates and processor load insulation at the cost of interactive responsiveness relative to the FreeBSD scheduler baseline. To match the performance of the FreeBSD scheduler, we added kernel priorities, abbreviated quanta, and windowed ticket boost to lottery scheduling, resulting in a hybrid lottery scheduler. Further, user feedback prompted us to add support for the UNIX `nice` utility. These techniques have been applied without squandering the proportional-share resource management semantics. The principle technique used by these mechanisms is dynamic ticket adjustments that influence scheduling order while preserving CPU utilization targets.

Our measurements show that our optimized scheduler incurs more overhead than the FreeBSD scheduler, but that these differences are negligible even under heavy workloads. We achieve throughput and responsiveness nearly equal to the FreeBSD scheduler. Our system has been deployed to two production machines with success. This paper demonstrates that our hybrid lottery scheduler is a viable process scheduler for the workloads that we have tested.

Availability

Our hybrid lottery scheduler is available from http://www.-cs.cmu.edu/~dpetrou/hls.tgz. Included are two new kernel source files, a *context diff* that patches 14 existing kernel files, and the source for 10 user-level programs that interact with the scheduler.

Acknowledgments

We thank the anonymous reviewers for their careful and detailed comments. Thanks also go to UC Berkeley's Computer Science Undergraduate Association and FreeBSD Users' Group for permitting us to deploy our experimental kernel on their production machines. Our colleagues Remzi Arpaci-Dusseau, Joan Digney, Jason Flinn, Greg Ganger, Dushyanth Narayanan, and David Rochberg kindly reviewed drafts. Finally, thanks go to Aaron Smith for asking us to support `nice` semantics and David Greenman for providing us with `wcarchive` statistics.

References

[1] Andrea C. Arpaci-Dusseau and David E. Culler. Extending Proportional-Share Scheduling to a Network of Workstations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, June 1997.

[2] D. L. Black. Processors, priority, and policy: Mach scheduling for new environments. In *Proceedings of the USENIX 1991 Winter Conference*, pages 1–12, January 1991.

[3] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, Reading, 1967.

[4] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An experimental time-sharing system. In *Proceedings of the 1962 AFIPS Spring Joint Computer Conference*, volume 21, pages 335–344, May 1962.

[5] Raymond B. Essick. An event-based fair share scheduler. In *Proceedings of the Winter 1990 USENIX Conference*, pages 147–162. USENIX, January 1990.

[6] Steve Evans, Kevin Clarke, Dave Singleton, and Bart Smaalders. Optimizing Unix Resource Scheduling for User Interaction. In *Proceedings of the 1993 Summer USENIX*, pages 205–218. USENIX, June 1993.

[7] The FreeBSD Operating System, 1999. See http://www.-freebsd.org/.

[8] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4, an Open Systems Design*. Prentice-Hall, 1994.

[9] Joseph L. Hellerstein. Achieving Service Rate Objectives with Decay Usage Scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[10] Kevin Jeffay, F. Donelson Smith, Arun Moorthy, and James Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[11] L. Kleinrock. A continuum of time-sharing scheduling. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 453–458, 1970.

[12] J. Larmouth. Scheduling for immediate turnround. *Software—Practice and Experience*, 8(5):559–578, September/October 1978.

[13] J. M. McKinney. A survey of analytical time-sharing models. *ACM Computing Surveys*, 1, 2:105–116, 1969.

[14] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc., 1996.

[15] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing Co., 2nd edition, 1992.

[16] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[17] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy Baruah, Johannes Gehrke, and C. Greg Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *IEEE Real-Time Systems Symposium*, December 1996.

[18] Jeffrey H. Straathof, Ashok K. Thareja, and Ashok K. Agrawala. UNIX scheduling for large systems. In *Proceedings of the USENIX 1986 Winter Conference*, pages 111–139. USENIX, Winter 1986.

[19] David G. Sullivan, Robert Haas, and Margo I. Seltzer. Tickets and currencies revisited: Extensions to multi-resource lottery scheduling. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 1999.

[20] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995.

[21] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11, November 14–17 1994.

[22] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Mangement. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, MIT Laboratory for Computer Science, June 1995.