ALPS: An Application-Level Proportional-Share Scheduler

Travis Newhouse
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404
newhouse@cs.ucsd.edu

Joseph Pasquale
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404
pasquale@cs.ucsd.edu

Abstract

ALPS is a per-application user-level proportional-share scheduler that operates with low overhead and without any special kernel support. ALPS is useful to a range of applications, including scientific applications that need to control the CPU apportionment to the processes they create, to Web servers that need to limit the proportion of available CPU time given to spawned processes that service Web requests, and to middleware that supports multiple execution environments that are to run at different rates. ALPS works by minimally sampling the progress of processes under its control, and making simple predictions for when it should selectively pause and resume the processes. We present the algorithm, a UNIX-based implementation, and a performance evaluation. Our results show that the ALPS approach is practical; we can achieve good accuracy (under 5% error), and low overhead (under 1% of CPU), despite user-level operation.

1 Introduction

Consider the problem of supporting a multi-process application that can benefit from proportional-share scheduling. By this, we mean an application that spawns a number of processes, each of which should get a pre-specified fraction of the total CPU time allocated to the application. Examples include a Web server that is to limit the proportion of available CPU time given to spawned processes that service web requests, or a scientific application that generates multiple processes, each of which computes over some space such as geographic area or physical volume, and it is desirable that the amount of available CPU time given should be allocated proportionally to the size of that space, e.g., based on adaptive mesh refinement. Another example is a middleware system that provides remote resource-controlled execution environments [19].

One approach to supporting such applications is to provide them with their own process scheduler, which simply executes as another user-level process. This has the benefits of not requiring modifications to the underlying kernel scheduler, thus making the solution highly portable, and allowing the scheduling to be on a per-application basis, thus tailored for each application. In fact, this idea is not uncommon in multi-threaded applications, which can do their own scheduling of threads by incorporating a special scheduling thread.

A problem that arises with a user-level approach is in how to support preemption, a desirable capability we assume throughout this discussion. If preemption is implemented by having the scheduling process run at a higher priority relative to the processes it schedules (so that it is guaranteed to run in response to timer interrupts) a complication arises as to how this affects or is affected by the rest of the workload. This higher priority can be achieved either by raising the priority of the scheduling process, or by lowering the priorities of the processes being scheduled relative to the rest of the workload. In the former case, the privilege of running at higher priority may be unavailable or undesirable (as viewed by other users), and in the latter case, by having less privilege than the rest of the workload, the application's processes may suffer in performance. These issues are compounded when there are multiple multi-process applications that each do their own user-level scheduling.

Another potential drawback of a user-level scheduling approach is that of overhead. Typically, a user-level scheduling process must set up a software (process-level) timer interrupt so that it may periodically wake up, check the statuses of the processes under its control, and decide which should run next. The overheads of context-switching in response to the timer interrupts, reading the statuses of processes to determine the amount of CPU time used, running the scheduling algorithm to decide which process(es) to run next, and invoking the mechanisms that make processes runnable and not runnable, can be high enough to make user-level scheduling not viable (at least for multi-process applications). In fact, we measured the overhead of each of these categories of operations on a modern PC running the FreeBSD variant of UNIX, and found that the overhead can be as high as roughly 20% for every hundred processes being scheduled.

Consequently, our goal in this work is to show how user-level proportional-share scheduling can be implemented with no special privileges (priority or otherwise), and with low overhead. The result is our design of the ALPS application-level proportional-share scheduler. An application that requires proportional-share scheduling may spawn an "ALPS" process that acts as a user-level scheduler for the application's regular processes. ALPS can run as an *unprivileged* process *with no special priority* under a typical unmodified UNIX scheduler [18]. Multiple concurrent applications can each employ their own ALPS (one per application). More generally, ALPS is useful for environments where the underlying kernel scheduler does not support proportional-share scheduling, or where it is desirable to run a multi-process application with no unusual effect on, or with no special privilege with respect to, the rest of the system's workload.

Previous approaches to proportional-share scheduling are designed to replace the kernel scheduler [8, 12, 14, 21, 26]. Simply implementing, as a user-level process, a scheduler that was originally designed to be implemented inside of the kernel may break certain assumptions in a way that can negatively affect accuracy and efficiency. For instance, an ordinary user-level process does not have absolute control of the CPU by which to reliably preempt processes, nor does it have access to the same information that is available to the kernel, such as notification when a running process blocks. Overhead is another potential problem because user-level scheduling must be performed by a process that itself must be scheduled frequently by the kernel to perform scheduling decisions.

Thread schedulers for user-level thread packages, despite sharing user-level operation with ALPS, are in fact more similar to a kernel scheduler as a result of being part of the process that controls the threads. The user-level thread scheduler can accurately preempt threads based upon virtual time alarms set within a process, and a thread notifies the user-level thread scheduler when it yields (e.g., while waiting for I/O). User-level thread schedulers benefit from low-cost context switching overhead, but sacrifice memory protection between threads and require applications to be modified to use a user-level threads package that implements a proportional-share scheduler.

In our approach, ALPS works in tandem with the underlying kernel scheduler, allowing and indeed expecting it to do as much work as it can. ALPS essentially "nudges" the kernel scheduler towards the goal of proportional share to override the kernel's native policy. The novelty of the ALPS scheduling algorithm is that it operates efficiently by minimizing the frequency of observations and of scheduling decisions, while maintaining good accuracy.

The rest of the paper is organized as follows. In Section 2, we describe the ALPS algorithm, including optimizations and support for I/O. In Section 3, we present a performance evaluation, showing results for accuracy and overhead. In Section 4, we show how multiple concurrent ALPSs perform, and we address scalability. In Section 5, we describe an application of ALPS to support a shared Web server. Section 6 contains a discussion of related works, and we present conclusions in Section 7.

2 The ALPS Algorithm

The ALPS scheduling model is based on a two-level approach in which an application spawns its own user-level ALPS scheduling process, which then works in concert with the underlying kernel scheduler to achieve proportional-share scheduling for that application's regular processes.

ALPS makes high-level decisions that determine which group of processes are eligible for execution for a near-term period of time, leaving it to the kernel scheduler to then schedule those processes during that time using its own policy. Thus, the goal of ALPS is to effectively restrict the decision space of the kernel scheduler so that ultimately, a proportional-share policy is achieved (according to a share distribution specified by the application).

2.1 Overview

ALPS selects multiple processes (of the application) to run at once, and then monitors their progress by periodically sampling their execution status. The period between these coarse-grained scheduling decisions is in terms of an *ALPS quantum* (called simply "quantum" from this point on, unless it must be distinguished from the kernel scheduler's quantum). During a quantum, ALPS defers fine-grained time-slicing to the kernel scheduler. The duration of the quantum is a primary configuration parameter that enables an application to balance accuracy and overhead.

ALPS attempts to achieve proportional distribution of CPU time over a period called a *cycle*. Each cycle is composed of a number of quanta (Figure 1). A cycle completes when sufficient CPU time (as opposed to real time) has been consumed by the application's processes such that ALPS may have feasibly scheduled the processes in exact proportion to their shares. If the duration of the quantum is Q time units and the total number of shares is S, then we define the cycle length to be $S \cdot Q$, assuming the shares have been scaled by their greatest common divisor. For example, if three processes have shares n, $2n$, and $3n$, for any integer n, the cycle length is $6Q$. Thus, the cycle dictates the period over which ALPS guarantees fairness in that each process

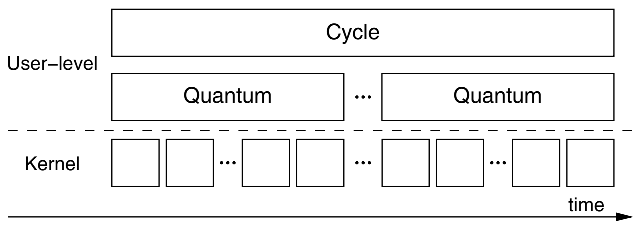

Figure 1. A quantum as defined by ALPS comprises an integral number of smaller kernel quanta; ALPS provides guarantees over a cycle, which comprises an integral number of ALPS quanta

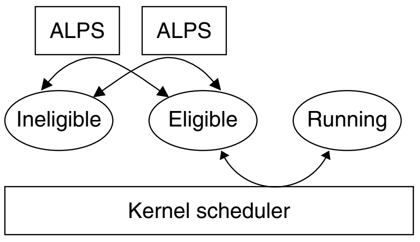

Figure 2. Each ALPS scheduler (one or more) moves processes under its control between the eligible and ineligible groups; the kernel schedules from the eligible group

may execute for a fraction of each cycle in proportion to its share. By defining fairness guarantees in this manner, ALPS effectively performs proportional-share scheduling on a virtual processor that executes at a (variable) rate dictated by the kernel scheduler.

ALPS operates by periodically measuring the progress of processes and enacting scheduling decisions by moving processes between two groups: one group of eligible-to-run processes, each of which has consumed less than its share of the CPU time during the current cycle, and another group of ineligible-to-run processes, each of which has exceeded its share. For the duration of each quantum, the processes in the eligible group contend for CPU time from the kernel scheduler (Figure 2). Just as if ALPS were not present, the task of the kernel scheduler remains to select an available process to execute on an available CPU. The kernel may select a process from the eligible group, or a process that is not under the control of ALPS. The number of processes from the eligible group that actually execute during an ALPS quantum depends on (1) the ALPS quantum length, (2) the maximum duration that the kernel allows a process to run at one time (e.g., the kernel scheduler's quantum), and (3) the scheduling policy of the kernel. If a process blocks during an ALPS quantum, the kernel scheduler will naturally select another process to execute, if one is eligible, without intervention by ALPS. This is a key and important difference between our approach and that of other user-level schedulers which only allow one process to contend for the processor at a time and must execute between each user-level context switch.

2.2 Basic Operation

The central idea of the ALPS algorithm is that each process has an *allowance* that indicates how many quanta of CPU time it may consume before the end of the current cycle. As long as the process's allowance is greater than zero, the process is eligible to run. As a process executes, its allowance is decremented by the amount of CPU time it actually receives. When its allowance becomes less than or equal to zero, the state of the process is changed to ineligible and its execution is suspended. When a cycle completes, the algorithm increases the allowance of each process in proportion to the process's share.

As shown in the pseudo code in Figure 3, the algorithm maintains global and per-process state. Globally, the algorithm maintains the total shares, S, and the time remaining in the current cycle, t_c. Associated with each process i are variables $share_i$ (the number of shares allocated to the process), $state_i$ (whether the process is eligible or ineligible to execute), and $allowance_i$ (the remaining number of quanta for which the process is eligible to run during the current cycle). The cycle time, t_c, is initialized to the cycle length, $S \cdot Q$. The process's allowance is initialized to its share and its state is initialized to ineligible. On account of its positive allowance, the process will become eligible for execution at the next quantum. For the moment, we ignore references to the $count$, $update_i$, and $blocked_i$ variables, which support optimization and I/O, and are discussed in the next subsections.

When ALPS invokes the algorithm during each quantum, it begins by measuring the CPU time consumed by each process that was eligible to run. Ineligible processes can be ignored as they will not have executed in the previous quantum. The value $consumed_i$ equals the CPU time consumed by the process since the algorithm was last invoked. The process's allowance is reduced by the amount it consumed scaled by the quantum length. The algorithm also updates the time remaining in the current cycle by subtracting each process's CPU consumption from t_c.

If t_c is less than zero after measuring the consumption of all processes, the current cycle has completed. As a result, the algorithm increments t_c by the cycle length, $S \cdot Q$, to establish the length of the next cycle. In addition, the allowance of each process, $allowance_i$, is incremented by its share, $share_i$. Finally, the algorithm partitions processes

```
$count \leftarrow count + 1$
for all $i : state_i =$ eligible and $update_i \leq count$ do
  $\langle consumed_i, blocked_i \rangle \leftarrow$ READ-PROGRESS($i$)
  $allowance_i \leftarrow allowance_i - consumed_i / Q$
  $t_c \leftarrow t_c - consumed_i$
  if $blocked_i =$ true then
    $allowance_i \leftarrow allowance_i - 1$
    $t_c \leftarrow t_c - Q$
  end if
end for

$cycles \leftarrow 0$
if $t_c \leq 0$ then
  $cycles \leftarrow 1$
  $t_c \leftarrow t_c + S \cdot Q$
end if

for all $i$ do
  $allowance_i \leftarrow allowance_i + share_i \cdot cycles$
  if $allowance_i > 0$ then
    $state_i \leftarrow$ eligible
  else
    $state_i \leftarrow$ ineligible
  end if
  if $update_i \leq count$ then
    $update_i \leftarrow count + \lceil allowance_i \rceil$
  end if
end for
```

Figure 3. The ALPS Algorithm

based upon the current allowance of each process.

To prevent allocation errors from accumulating from one cycle to the next, the algorithm increments cycle time and allowances to correct allocation errors in future cycles, effectively extending the period over which the target distribution is made, but only when necessary. For example, if a process consumes twice its share in one cycle, then the process will not execute in the next cycle because its allowance will be negative even after incrementing it by the process's share. Thus, considering a period of two cycles, the process will have received its target distribution.

2.3 Optimizations

If ALPS were to be implemented as just described, it would operate very inefficiently. Specifically, obtaining information from the underlying operating system about each and every process's progress incurs high overhead. We measured the time to perform the three major operations of the ALPS algorithm: (1) receive a timer event at the end of each quantum, (2) measure CPU time consumed by a process since its last measurement, and (3) move processes between eligible and ineligible groups by sending a signal. As shown in Table 1, the time to measure the CPU consumption of a process dominates the implementation overhead of the ALPS algorithm. As it turns out, it is *not necessary* that ALPS measures the progress of each process at the end of each quantum. In fact, a key feature of the algorithm is that its logical framework lends itself to the following optimization, which promotes efficient implementation without sacrificing accuracy.

Table 1. Primary ALPS Operations Times (μs)

Receive a timer event	9.02
Measure CPU time of n processes	$1.1 + 17.4n$
Signal a process	0.97

To reduce overhead, we take advantage of the fact that a process can consume at most one quantum of CPU time between each invocation of the ALPS scheduling algorithm (since the algorithm runs each quantum). More generally, a process i must be eligible for a duration of at least $allowance_i$ quanta to consume enough CPU time for it to become ineligible for execution. Therefore, the algorithm can postpone measuring the CPU consumption of a process for $allowance_i$ quanta from the last measurement. If a process's allowance contains a fractional number of quanta, we round up to the next integer to determine how many quanta to wait. So, for example, if a process's allowance is 4.3, there is no way this process can complete in less than 5 quanta, and so checking its status before the 5th quantum expires is wasted work that can be eliminated.

To implement this optimization, the algorithm uses the variable *count* to index the timer events that it services, and for each process i, the variable $update_i$ stores the index of the quantum at which to next measure the consumption of the process. The algorithm increments *count* upon each invocation. In the measurement loop, we augment the conditional to test whether to measure a process's progress during the current quantum. Finally, if a process is measured during an invocation of the algorithm, then the algorithm uses the process's current allowance to compute a new value for $update_i$ (the next quantum at which to measure the process).

2.4 Accounting for I/O

At user level, ALPS lacks precise knowledge of when a process blocks for and resumes from an I/O request. Yet, we do not want a process that performs I/O to limit the progress of other processes that are ready to execute, by delaying the end of a cycle. The approach we take is simple, with the relevant support code in the body of the measurement loop.

When the algorithm measures the progress of a process, it also determines whether the process happens to be blocked (e.g., by reading the "wait channel" state variable of a process in the UNIX kernel, which indicates the event for which a process is waiting, if any). If the algorithm detects that a process is blocked, then we simply assume that the process has been blocked for an entire quantum. Since a blocked process has voluntarily relinquished its interest in the CPU, the algorithm reduces the process's allowance by one quantum because the process "gave up" its right to execute for that period of time.

The algorithm also reduces the remaining cycle time, t_c, by the length of one quantum for each blocked process. Recall that the length of a cycle is determined by the number of quanta required to provide each process with its proportional share, namely $S \cdot Q$. If the algorithm decreases a process's allowance in a given cycle, then the number of quanta of CPU time required to fulfill the proportional-share guarantee decreases by an equal amount. The effect is that if a process blocks for all of its allocated quanta during a cycle, then the cycle will end early, as if the blocked process's shares had never contributed to the length of the cycle. The remaining processes, that will have consumed their allowance if they were ready-to-run during the entire cycle, will earn a new allowance such that they can become eligible to run again.

Note that the process may have been blocked for some time before ALPS detects that it blocked, but this cannot be known because our only evidence is that it has not consumed CPU time, but this may simply be due to not getting the CPU because of other competing processes. Hence, all we know is that the process is now blocked, and may remain blocked for an unknown period of time. Since we can check again at the next quantum, we reduce the allowance by only one quantum. If the process does indeed remain blocked for the quantum, then the cycle length is correctly reduced by one since the blocked process is out of contention. However, if the process happens to wake up, then it will have effectively been penalized by having its allowance reduced by one. On the other hand, since the process was not penalized for its time blocked before it was detected as blocked, this simple heuristic seems reasonable, and indeed, seems to work well based on our experiments.

3 Performance Evaluation

We evaluate the ALPS algorithm using workloads that vary in the number of processes and in share distribution. The number of processes in a workload is either 5, 10, or 20. The shares assigned to processes follow one of three distribution models: *linear*, *equal*, or *skewed*. We chose the total number of shares as follows: a workload of 5 processes has 25 total shares, a workload of 10 processes has 100 total shares, and a workload of 20 processes has 400 total shares. Selecting the total number of shares to be n^2, where n is the total number of processes, is purely for convenience, as the distribution of shares for each model result in integral amounts. (We did not scale the shares of any workload by their greatest common divisor.) Table 2 summarizes the share distribution of the workloads. The test machine for all experiments is a 2.2 GHz Pentium 4 processor with 512 MB memory. The host operating system is FreeBSD 4.8.

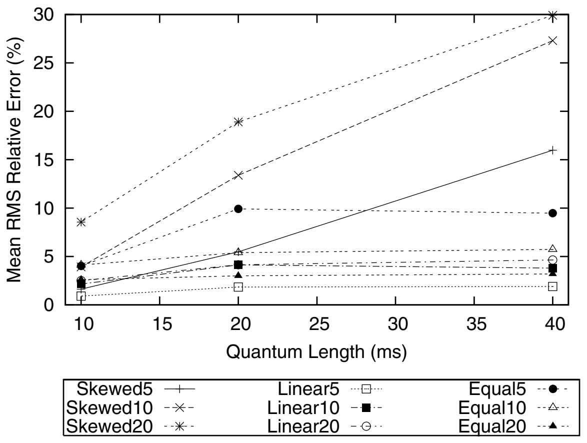

Figure 4. Accuracy: mean relative error of ALPS using various quantum lengths

3.1 Accuracy

To evaluate accuracy, we instrument ALPS to record a log of the CPU time consumed by each process in every cycle. To arrive at a single value that represents the accuracy of the algorithm for a particular workload and quantum length, we first compute the root mean square (RMS) of the per-process relative errors in a cycle (actual to ideal CPU time consumed), and then we compute the mean of the RMS relative error over all cycles in an experiment (200 cycles). Figure 4 contains the summarized accuracy of the ALPS algorithm for various workloads scheduled at different quantum lengths. Each point is the mean of 3 tests. For most workloads, the RMS relative error is low, less than 5%.

The ALPS algorithm exhibits the highest relative error for the skewed workloads. In the skewed workloads, a majority of the processes have only a single share. As a result, quantization effects have more effect on the relative error. However, as the quantum length is reduced, this problem is minimized. The question then becomes, how much overhead is incurred, especially for smaller quantum lengths?

Table 2. Workload Share Distributions

	5 processes	10 processes	20 processes
Linear	$\{1, 3, 5, 7, 9\}$	$\{1, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$	$\{1, 3, 5, \ldots, 35, 37, 39\}$
Equal	$\{5, 5, 5, 5, 5\}$	$\{10, 10, 10, 10, 10, 10, 10, 10, 10, 10\}$	$\{20, 20, 20, \ldots, 20, 20, 20\}$
Skewed	$\{1, 1, 1, 1, 21\}$	$\{1, 1, 1, 1, 1, 1, 1, 1, 1, 91\}$	$\{1, 1, 1, \ldots, 1, 1, 381\}$
Total Shares	25	100	400

3.2 Overhead

To measure overhead, we use the `getrusage()` system call to measure the amount of CPU time consumed by ALPS during a test run. We calculate overhead as the ratio of the CPU time consumed by ALPS to the wall time duration of the experiment. We verified our measurements by comparing the amount of work (a loop counter) performed per unit of real time by the workload processes with and without control of ALPS (though we found higher variance in this measurement technique).

Figure 5 shows the overhead of the algorithm when scheduling workloads at quantum lengths of 10, 20, and 40 milliseconds. In general, overhead is very low, typically under 0.3% for most of the workloads. When comparing the ALPS algorithm to a version without the optimization described in Section 2.3, we found that this optimization reduces overhead by a factor of at least 1.8 and as much as 5.9, for the workloads that we tested.

The overhead is highest for the equal share distributions because fewer processes become ineligible during a cycle. For the skewed and linear workloads, the processes with small shares (relative to others) quickly consume their allowance. Once they become ineligible, the ALPS does not measure the progress of those processes, and ALPS can wait longer between measurements of the remaining processes that have (relatively) larger remaining allowances. On the other hand, all of the processes of the equal share workloads progress at a similar rate. Until the cycle nears completion, few processes consume their allowance, which results in fewer opportunities for ALPS to reduce work.

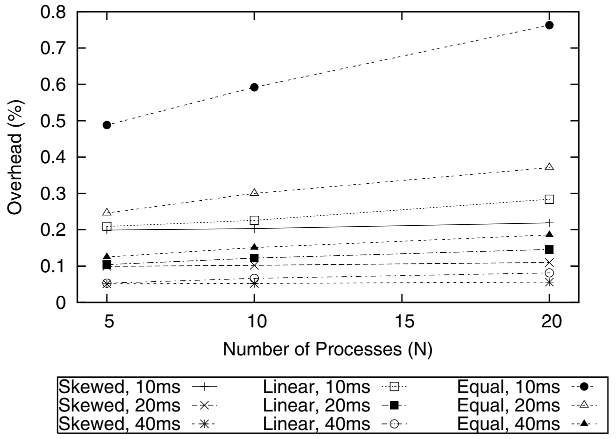

Figure 5. Overhead: fraction of time ALPS executes vs. duration of experiment

3.3 I/O

To illustrate how ALPS reacts to a process that performs I/O, we use a simple workload consisting of 3 processes, A, B, and C, with a share distribution of 1, 2, and 3, respectively. ALPS uses a 10 millisecond quantum. After waiting for the processes to reach a steady state of execution, process B begins simulating I/O requests by sleeping for 240 milliseconds after every 80 milliseconds of execution time. Because ALPS schedules process B to execute at a rate of 33.3% of the CPU, it requires 240 milliseconds of real time to receive 80 milliseconds of CPU time. Therefore, the time process B spends in a non-blocked (ready or running) state will equal the time spent doing I/O, and it will alternately be non-blocked for 4 cycles and be blocked for 4 cycles. While blocked, we expect ALPS to distribute CPU time in a ratio of 1:3 between the process A and C.

As Figure 6 depicts, ALPS does indeed proportionally redistribute the CPU time relinquished by process B. Near cycle 590, process B begins performing I/O. Prior to this point, the processes receive the correct shares of the CPU. Afterward, during the 4 cycles that process B is non-blocked, ALPS continues to maintain the same ratios of 1:2:3. However, while process B is blocked, ALPS distributes 25% of the CPU to process A and 75% of CPU to process C, as expected.

4 Advanced Experiments

In this section, we show that when multiple ALPSs execute simultaneously, each ALPS schedules processes with the fraction of the CPU time that the kernel assigns to its workload. In addition, we discuss scalability by presenting empirical results on the limit of the number of processes over which ALPS can maintain control.

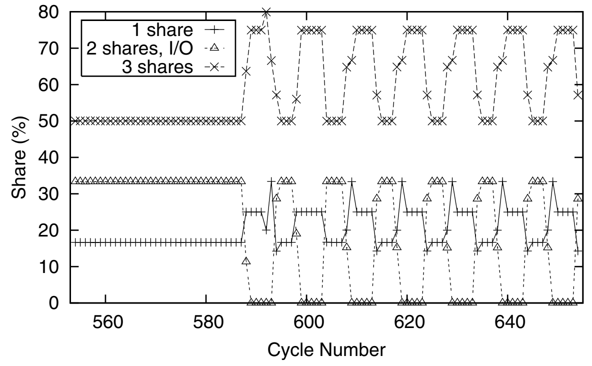

Figure 6. I/O: ALPS proportionally distributes CPU time when the 2-share process blocks

Table 3. Accuracy of Multiple ALPSs

	Target	Phase 1		Phase 2		Phase 3	
S	%cpu	%cpu	%re	%cpu	%re	%cpu	%re
1	16.7	-	-	-	-	16.5	1.2
2	33.3	-	-	-	-	33.1	0.6
3	50.0	-	-	-	-	50.4	0.8
4	26.7	-	-	27.3	2.2	26.5	0.7
5	33.3	-	-	34.0	2.1	33.2	0.3
6	40.0	-	-	38.7	3.3	40.3	0.8
7	29.2	29.5	1.0	29.2	0.0	28.9	1.0
8	33.3	33.2	0.3	33.3	0.0	33.1	0.6
9	37.5	37.3	0.5	37.5	0.0	38.0	1.3

4.1 Multiple Applications

The optimized ALPS scheduling algorithm proportionally schedules whatever CPU time the workload processes receive from the kernel scheduler. We show this capability by executing multiple ALPSs simultaneously. Though we use multiple ALPSs to generate load on the machine, each ALPS does not know what causes a reduction in the CPU time available to its workload; it simply uses whatever is made available to it and correctly apportions that time to the processes under its control. In fact, it does not matter what the workload outside an ALPS's control is (i.e., whether they are processes under the control of other ALPSs or not); we show that each ALPS, when there are multiple ones, operates equally well.

In the experiment, there are 3 independent groups of processes, that we label A, B, and C, where each group has 3 processes, with share distributions of {7, 8, 9}, {4, 5, 6}, and {1, 2, 3}, respectively. The experiment has three phases. The first phase starts at time 0 and ends at time 3000, during which group A processes run exclusively. The second phase then begins, and ends roughly at time 6000, during which group B processes run simultaneously with those already running from group A. Finally, the third phase then begins, and ends roughly at time 15000, during which those in group C run with those of the other groups.

Figure 7 shows the cumulative CPU time received by each process. The x-axis is in units of real time. Each data point occurs at the end of a cycle for the ALPS that schedules a process. The cycles of distinct ALPSs are not synchronized. The real time duration of an ALPS's cycle depends on the total number of shares in its process group and the rate at which its processes execute (as dictated by the kernel scheduler).

In each phase, the rise in cumulative CPU time for each process is linear. Using linear regression, we calculated the slopes of fitted lines for each process during each phase. From this, we determined the fractional CPU time that each process received relative to the other processes in its group. So, for example, process 1, which only ran in phase 3, received 16.5% of the total CPU time received by the processes in its group (C: processes 1, 2, and 3). Thus, within its group, given that it should have received 1 share out of 6 (making that target fractional CPU time equal 16.7%), the 16.5% that it received is very close to its target, resulting in a relative error of 1.2%.

In fact, within each group, the amount of CPU time the processes receive is very close to what they are supposed to receive. Table 3 lists, for each phase, the relative percentage of CPU time received by a process and the relative error. The relative error ranges are 0.3-1.0% in Phase 1, 0.0-3.3% in Phase 2, and 0.3-1.3% in Phase 3, resulting in an average relative error of 0.93%. Thus, each ALPS is operating accurately, despite the presence of other ALPSs scheduling other processes.

We note that this conclusion regarding accuracy is relative, in the sense that whatever CPU time is made available to a group of processes under the control of an ALPS, the ALPS apportions CPU time very close to their specified shares. However, what is not under the control of each ALPS is the total CPU time made available to its group of processes, which is determined by the underlying kernel scheduler. For example, it may be the case that a fair-share kernel scheduler gives each group an amount roughly in proportion to the number of processes in the group (so, if each group had the same number of processes, they should get the same fraction of CPU time). Hence, for our experiment, the kernel scheduler would give 100% of the total CPU time (assuming no other load) to group A during phase 1; during phase 2, it would give 50% to each of the groups A and B; during phase 3, it would give 33.3% to each of the groups A, B, and C. In fact, this is what we observed (very roughly, i.e., with up to 20% error, because each of the processes is not always running or eligible to run all of

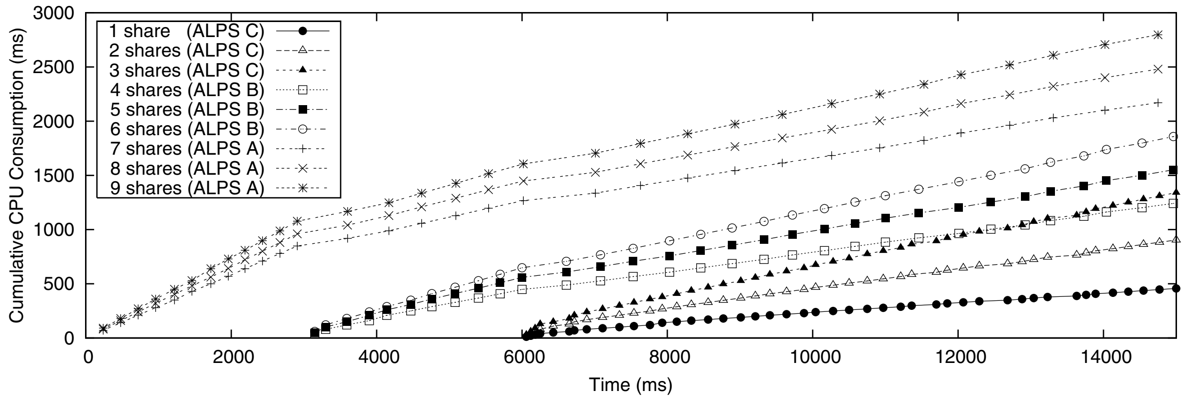

Figure 7. Cumulative CPU consumption vs. wall time for processes scheduled by 3 distinct ALPSs

the time). This can be seen, for example, with the behavior of processes 3 and 4. While process 3 has fewer shares than process 4, they are in different groups, and so process 3 receives $3/6$, or 50% of the CPU time given to its group, and process 4 receives $4/15$, or 26% of the CPU time given to its group. What they ultimately receive in absolute CPU time is determined by the FreeBSD kernel scheduler. In fact, process 3 executes at a higher absolute rate (as can be seen from its higher slope) both because it receives a larger share of the CPU time allocated to its group, and each group is getting roughly $1/3$ of the CPU.

In addition to the above "average behavior" effects over time, from Figure 7 we can observe the more detailed dynamic effects of how the FreeBSD kernel scheduler allocates CPU time to the various process groups. For example, as each new phase begins, CPU time is being spread over more processes, and so the absolute rate of execution of the existing processes decreases. Since the processes receive CPU time at a lower rate, the real time duration of a cycle increases in length. Also, the cycle lengths of existing process groups are a bit longer at the transition point between phases, as work is performed to fork a new ALPS and 3 workload processes. In fact, these new processes will be initially favored by the FreeBSD kernel scheduler as their dynamic kernel priority will be higher than the existing processes (since the new processes have not yet consumed any CPU time and the existing processes are compute-bound).

In conclusion, the long-term behavior of the system is stable in that each individual ALPS apportions CPU time accurately within its process group. This is the best we could expect, given that we do not have (and do not assume) any control of the underlying kernel scheduler.

4.2 Scalability

Here we address the question of how many processes ALPS can schedule before it breaks down (which it will since it runs as a user process which has no special privileges, not even a special higher priority). We use an equal share workload because in our performance evaluation, ALPS has the largest overhead for an equal share distribution (see Figure 5). In the experiment, we set the number of shares per process to be 5, and we increase the number of processes that ALPS schedules until we observe a loss of control. We test ALPS at quantum lengths of 10, 20, and 40 milliseconds. Figure 8 shows the overhead of ALPS, and Figure 9 shows the RMS relative error.

Although the overhead of ALPS does not exceed 2.5%, the significant factor is the amount of work that ALPS performs relative to the processes that it schedules. For each quantum length, overhead increases linearly until a threshold is reached. The threshold is determined by the point where the overhead (the CPU time used by ALPS per quantum) exceeds the inverse of the number of workload processes plus one (to account for the ALPS process itself). The latter determines the fraction of a quantum for which ALPS may run (e.g., if there are 20 processes, ALPS has $1/21$ of a quantum to complete its work for that quantum before exceeding its "fair share" of the CPU as scheduled by the kernel scheduler).

If ALPS requires more time, then it may not be scheduled promptly by the kernel when a quantum expires. In the experiment, this limit is imposed by the FreeBSD kernel scheduler that tries to allocate CPU time to competing processes by calculating a dynamic priority based on prior execution time [18]. To the kernel, ALPS is a process no different than the workload processes, and if its dynamic priority is lower than that of a workload process (e.g., since it has executed longer), then the kernel will schedule the workload process rather than ALPS.

Using linear regression, we calculated lines for the initial (linear) portions of the percentage overhead:

$$U_{10}(N) \quad = \quad .0639N + .0604$$

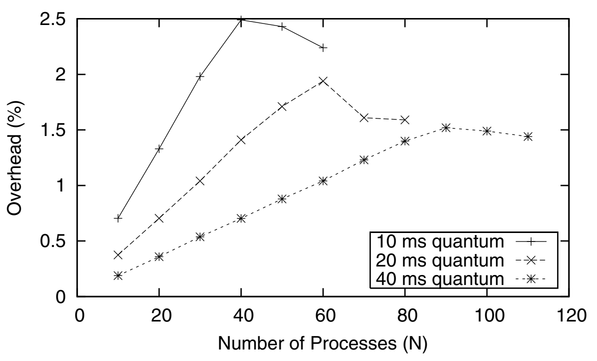

Figure 8. Overhead for equal share workload

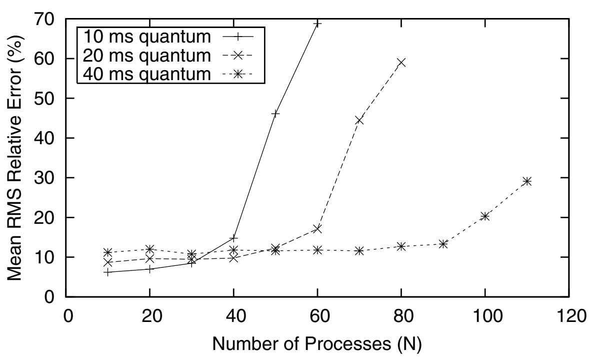

Figure 9. Accuracy for equal share workload

$$U_{20}(N) = .0338N + .0340$$
$$U_{40}(N) = .0172N + .0160,$$

where N is the number of the processes and the subscript indicates the quantum length, Q, in milliseconds. The breakdown threshold, N^*, will occur at or beyond the point at which the overhead, $U_Q(N)$, intersects the percentage of a quantum available to ALPS, which can be determined by solving the following equation:

$$U_Q(N^*) - 100/(N^* + 1) = 0.$$

The predicted thresholds are 39, 54, and 75 processes for quantum lengths of 10, 20, and 40 milliseconds, respectively. The observed thresholds are 40, 60, and 90 processes for the same quantum lengths, which match well. With a 40 millisecond quantum length, ALPS is able to maintain control past the theoretical threshold. We attribute this to the fact that ALPS is asleep for longer periods of time over which the kernel scheduler will credit its priority because it is not contending for the CPU (i.e., the FreeBSD kernel scheduler favors interactive tasks [18]).

5 An ALPS-based Shared Web Server

The experiments in the preceding section characterized the accuracy and overhead of ALPS when applied to a synthetic, compute-bound workload. In this section, we demonstrate the utility and competency of ALPS when applied to a realistic application. A prevalent example of resource sharing is a shared Web server that hosts several users' content bases. If a shared Web server supports dynamic content, the administrator must prevent a single user from degrading the service of other users by deploying malicious or buggy code that overloads the CPU.

The flexibility to implement new resource sharing policies is a paramount feature of ALPS. For this experiment, we enforce a resource sharing policy that differs from the kernel in two important aspects. The resource distribution among principals is not an equal share policy like that of the kernel scheduler. Additionally, the principal that is scheduled is not a process, but rather a user. Thus, the policy is that CPU consumption by any process of a particular user counts against that user's allocation. We schedule a user's processes as a whole when their total consumption is above or below the user's allocation. The idea of decoupling the resource principal from the process abstraction has been previously introduced in the form of kernel abstractions [6, 24]. Our results show that it is possible to implement similar functionality with acceptable accuracy and overhead using ALPS.

Amza, et al. developed three benchmarks for evaluating Web sites composed of dynamic content [1, 2]. The benchmarks model an online bookstore, an auction site, and a bulletin board. They found that the CPU was the Web server's bottleneck resource for the auction site and bulletin board. We use the RUBBoS bulletin board benchmark for our experiments because it is representative of the type of application that a customer of a shared Web server might install. The benchmark implements a bulletin board site with functionality similar to Slashdot [23]. The bulletin board maintains a database of stories and user comments about each story. When a client accesses the bulletin board, a PHP script retrieves a story and its associated comments from the database and presents them in a single HTML page.

5.1 Experiment

The setup for our experiments consists of a Web server, a database server, and three client workstations that generate requests. The Web server is a 2.2 GHz Pentium 4 processor with 512 MB of memory running FreeBSD 4.8, Apache 2.0.48 configured with the "prefork" MPM, and PHP 4.3.4 loaded as a dynamic Apache module. The database server and client machines are dual-Pentium III 600MHz processors with 1024 MB of memory running the Linux 2.4.20

kernel. The database server software is MySQL 3.23.58. A 100 Mbps and 1000 Mbps switched Ethernet connects the machines. The RUBBoS benchmark provides the data files for the database, PHP scripts for the Web server, and a client workload driver that runs in parallel on the client workstations.

We host three instances of the bulletin board Web site on the Web server machine by running a distinct instance of Apache on three different ports. Each instance of the Apache server runs as a different user account and is configured to use at most 50 processes (a number chosen for maximum throughput). Apache automatically regulates the number of active processes up to this maximum. The same level of concurrency might also be achieved by using multiple user-level threads executing within one or a small number of processes. Such a server configuration will significantly reduce the number of processes that ALPS must monitor. Development of a multi-process, multi-threaded Web server by the Apache group is in progress, but was not sufficiently operational at the time of this writing.

We first measure how the kernel schedules the Web servers by feeding requests from the client workstations. Each workstation uses 325 simultaneous clients to drive one of the three bulletin board Web sites. The number of clients was selected experimentally to achieve highest total throughput. The throughputs, measured in requests per second by the workstation machines, for the three Web sites are $\{29, 30, 40\}$. The kernel scheduler allocates the CPU roughly evenly with the three Web sites.

Next, we employ ALPS to isolate the performance of the three different Web servers. The distribution of shares is $\{1, 2, 3\}$ and the quantum length is 100 milliseconds. Again, we generate a request workload from the client workstations using 325 simultaneous clients. The throughputs we measured are $\{18, 35, 53\}$ requests per second. As can be seen, ALPS allocates the CPU in the desired proportions.

5.2 Remarks

Because we ran Apache as multi-process application that dynamically spawns processes, we had two choices of how to impose ALPS on the Apache Web servers. We could modify Apache to notify ALPS each time a new process was created, or ALPS could monitor the processes created by Apache. We chose the latter so that we did not need to modify Apache. We modified the implementation of ALPS to treat a group of processes as a single resource principal and to update the processes associated with each principal once per second. To perform the update, ALPS selects all the processes belonging to the user under which the Web server is running. The `kvm_getprocs()` library call in FreeBSD provided a convenient way to obtain all the process identifiers belonging to a given user.

We do not concern ourselves with overloading beyond the capacity of the CPU, such as caused by flash crowds or a denial-of-service attack. Prior work has investigated solutions to prevent overload caused by receiver live-lock [7, 11, 15], and is complementary to our solution for sharing the CPU.

6 Related Works

Several approaches have been taken to support application-level resource policies. An exokernel and an infokernel both provide kernel-level interfaces suitable for applications to implement resource management policies at user-level [5, 13]. Gray-box systems are similar to infokernels, but the operating system is not modified; applications infer information from the existing kernel interface [4]. We take the approach of gray-box systems by treating the operating system as an unmodifiable component that provides the information and control mechanisms necessary to implement a proportional-share scheduling policy at user-level.

Scheduling research most related to our work is that focused on proportional-share scheduling and scheduling that guarantees rates of execution for soft real-time applications [8, 12, 14, 16, 20–22, 26, 27]. A distinguishing feature of our approach is that the ALPS scheduling algorithm promotes a user-level implementation which is practical, portable, and effective: running under an unmodified UNIX-based kernel, ALPS provides accurate proportional-share execution while minimizing overhead by frugal sampling of processes' progress. Key to this result is that ALPS selects a *group* of processes for execution, deferring fine-grained time-slicing of processes within a group to the kernel, promoting efficiency and correct operation in the presence of I/O.

Thread schedulers for user-level threads offer another mechanism to implement application-level resource policies. A scheduler activation is a mechanism for the kernel to express CPU availability to a user-level thread scheduler so that it may improve thread concurrency on multiprocessor systems [3]. The Capriccio user-level threads package includes a resource-aware scheduler that schedules threads based on their predicted resource usage in an effort to maximize throughput of network services [25]. Like kernel schedulers, thread schedulers assume knowledge of key events in a thread's lifetime, such as I/O requests, and observe such events by running in the threads' address space. ALPS runs external to an unmodified, multi-process application or group of processes. By sampling processes' CPU consumption and wait status, we trade a degree of accuracy for simpler deployment by avoiding modifications to the kernel or applications.

Some UNIX variants support fixed "real-time" priorities. This higher class of priorities can be used to implement a user-level, reservation-based scheduler that supports soft real-time applications [9]. Using this special capability requires administrator privileges. With ALPS, our goal is more modest: we simply seek to reapportion the CPU time the kernel allocates to a set of processes. We show how this can be implemented in conjunction with standard UNIX scheduling with no special priorities, and with no special privileges.

Other works present a control-theoretic approach to controlling application execution in which a feedback loop manages application resources. This requires modifying the application to report progress [10], or an understanding of the operation and performance goals of the application [17].

7 Conclusions

We presented the design and implementation of the ALPS application-level proportional-share scheduler, which provides any application with proportional-share scheduling for its processes. Under UNIX, ALPS runs as an unprivileged process (with no required kernel modifications). ALPS makes high-level decisions that determine which group of processes are eligible for execution for a near-term period of time, leaving it to the kernel to then schedule those processes during that time.

The key to ALPS's efficiency is in allowing the kernel scheduler to do as much work as possible, and then filling in the details, with minimal impact, to ultimately achieve proportional-share scheduling. After measuring all of the various operations invoked by ALPS, we determined that the most expensive by far was in reading each controlled process's state. Not only is this operation expensive on a per-invocation basis, but it grows with the number of processes being scheduled. Hence, ALPS minimizes invocations of this operation, essentially relying on predictions of what the future state will be and determining when action will be needed. We found that this leads to significantly lower overhead (under 1%) without sacrificing accuracy (relative error remained under 5%).

Finally, we showed that multiple ALPSs for multiple multi-process applications are each able to accurately schedule the CPU time made available by the kernel scheduler, regardless of how this CPU time availability varies over time. In addition, ALPS is capable of accurately enforcing proportional share when it detects that a process is doing I/O. We also showed that a limitation to a purely user-level approach to proportional-share scheduling is that, since ALPS itself is subject to the scheduling policy of the kernel scheduler, there are practical limits on the number of processes ALPS can schedule, depending on the amount of CPU time ALPS requires relative to the workload. However, we found that these limits are quite reasonable (many tens of processes) on current PCs.

References

[1] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Bottleneck characterization of dynamic Web site benchmarks. Technical Report TR02-389, Rice University, 2002.

[2] C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic Web site benchmarks. In *Proc. IEEE 5th Annual Workshop on Workload Characterization*, 2002.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proc. 13th ACM Symp. on Op. Sys. Princ.*, 1991.

[4] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. 18th ACM Symp. on Op. Sys. Princ.*, 2001.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proc. 19th ACM Symp. on Op. Sys. Princ.*, 2003.

[6] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proc. 3rd Symposium on Operating Systems Design and Implementation*. USENIX, 1999.

[7] A. Bavier, T. Voigt, M. Wawrzoniak, L. Peterson, and P. Gunninberg. SILK:scout paths in the linux kernel. Technical Report 2002-009, Department of Information Tecnology, Uppsala University, 2002.

[8] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *Proc. 4th OSDI*, 2000.

[9] H. Chu and K. Nahrstedt. A soft real time scheduling server in UNIX operating system. In *Proc. 4th Intl. Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, 1997.

[10] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *Proc. 17th ACM Symp. on Op. Sys. Princ.*, 1999.

[11] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proc. 2nd Symposium on Operating Systems Design and Implementation*. USENIX, 1996.

[12] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. 17th ACM Symp. on Op. Sys. Princ.*, 1999.

[13] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symp. on Op. Sys. Princ.*, 1995.

[14] P. Goyal, X. Guo, and H. M. Vin. A hierarchial CPU scheduler for multimedia operating systems. In *Proc. 2nd OSDI*, 1996.

[15] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *Proc. IEEE Real-Time Systems Symposium*. IEEE Computer Society, 1998.

[16] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *Proc. 16th ACM Symp. on Op. Sys. Princ.*, 1997.

[17] Y. Lu, T. F. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for QoS guarantees and its application to differentiated caching services. In *Proc. 10th Intl. Workshop on Quality of Service*, 2002.

[18] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[19] T. Newhouse and J. Pasquale. Java Active Extensions: Scalable middleware for performance-isolated remote execution. *Elsevier Computer Communications Journal*, 28(14):1680–1691, 2005.

[20] J. Nieh and M. S. Lam. A SMART scheduler for multimedia applications. *ACM Trans. Comput. Syst.*, 21(2):117–163, 2003.

[21] J. Nieh, C. Vaill, and H. Zhong. Virtual-time round-robin: An O(1) proportional share scheduler. In *Proc. 2001 USENIX Annual Technical Conf.*, 2001.

[22] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proc. 22nd IEEE Real-Time Systems Symposium*, 2001.

[23] Slashdot. http://www.slashdot.org/, 2004.

[24] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[25] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proc. 19th ACM Symp. on Op. Sys. Princ.*, 2003.

[26] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, 1995.

[27] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proc. 19th ACM Symp. on Op. Sys. Princ.*, 2003.