J Grid Computing (2007) 5:251–270
DOI 10.1007/s10723-007-9067-7

Achieving Efficiency and Accuracy in the ALPS Application-level Proportional-share Scheduler

T. Newhouse · J. Pasquale

Received: 25 August 2006 / Accepted: 12 January 2007 / Published online: 21 February 2007

Abstract We present the design and implementation of ALPS, a per-application user-level proportional-share scheduler. It provides an application with a way to control the relative allocation of CPU time amongst its individual processes. The ALPS scheduler runs as just another process (belonging to the application) at user level; thus, it does not require any special kernel support, nor does it require any special privileges, making it highly portable. To achieve efficiency, ALPS delegates fine-grained time-slicing responsibility to the underlying kernel scheduler, while itself making coarse-grained decisions to achieve proportional-share scheduling, all in a way that is transparent to the underlying kernel. Our results show that the ALPS approach is practical; we can achieve good accuracy (under 5% relative error) and low overhead (under 1% of CPU time), despite user-level operation.

Key words proportional-share scheduling · multi-process applications · user-level scheduling · performance

T. Newhouse · J. Pasquale (✉)
Computer Science and Engineering,
University of California, San Diego, La Jolla,
CA 92093-0404, USA
e-mail: pasquale@cs.ucsd.edu

T. Newhouse
e-mail: newhouse@cs.ucsd.edu

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. Prime examples are scientific applications that generate 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). Other examples include Web servers that seek to limit the proportion of available CPU time given to spawned processes that service Web requests. More general examples include middleware systems that provide remote resource-controlled execution environments [20] for computing-utility servers. Along the same lines, the idea of running a massive number of virtual machines (VM) on a (much smaller) set of physical machines [7] to host applications in firewalled environments or to simulate very large-scale systems has become popular; being able to quantitatively apportion CPU time to the VMs would be valuable.

To support such scenarios, we present the design of the ALPS application-level proportional-share scheduler. ALPS runs as an unprivileged user-level process with no special priority when running under a typical unmodified UNIX scheduler [19]. ALPS can be "applied" to any group of processes to allocate the CPU amongst them in user-specified proportions. These processes may be related (e.g., they are spawned by a single application), or unrelated (e.g., spawned by numerous independent applications). Multiple ALPS schedulers, each controlling an independent group of processes, may run simultaneously on the same machine. Because each ALPS has no special privilege, it has no unusual effect on the rest of the system's workload. It is simply a competing process, and because it runs very infrequently, its interference is minimal.

The challenges of developing user-level scheduling mechanisms for processor resources arise from the lack of system information and the tradeoff between accuracy in policy enforcement and operational overhead. In particular, one cannot simply extract a proportional-share scheduler designed to operate within an operating system's kernel [9, 13, 15, 22, 27], and have it run at user level, as this may break assumptions regarding system knowledge and control capabilities that can negatively impact accuracy and efficiency. For instance, an ordinary user-level process cannot exact 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 a potential problem because user-level scheduling must be performed by a process that itself must be scheduled frequently enough by the kernel to effectively make scheduling decisions.

Our approach is to design a user-level scheduler that works in tandem with, but at the same time transparent to, the underlying kernel scheduler, allowing and expecting kernel to do as much work as it can (and thus *not* replace it). The user-level scheduling process essentially "nudges" the kernel scheduler towards the goal of proportional share to override the kernel's native policy for the group of processes under its control. The novelty of the ALPS scheduling algorithm is that it operates very efficiently by minimizing the frequency of observations and of scheduling decisions, while maintaining good accuracy. It also allows for processes that do I/O, without limiting the work performed by processes that are ready to execute. It runs as a normal unprivileged user process, with no expected modifications to kernel.

ALPS has numerous advantages. It is applicable to the shared server scenarios described above because clients of, say, a compute utility do not have the privilege to invoke administrator-level mechanisms for enhanced priority nor the freedom to modify the host operating system. Multiple ALPS can run simultaneously, each scheduling the processor resources assigned to their associated processes by the kernel scheduler. The processes of an application need not be modified to use ALPS. Plus, a user-level design that relies on only a few commonly supported operating system mechanisms, as ours does, can be easily ported to other operating systems.

In this paper, we describe the design, implementation, and a performance evaluation of the ALPS application-level proportional-share scheduler. In Section 2, we further motivate our design and describe the framework that we use for our implementation. In Section 3, we describe and evaluate a basic and somewhat simplistic version of the algorithm, determining its accuracy and overhead to establish a baseline of performance. In Section 4, we describe an optimized version of the algorithm that reduces overhead without sacrificing accuracy, and we extend it to allow for I/O. In Section 5, we present advanced experiments, showing how multiple concurrent ALPSs perform, the scalability of ALPS, and the application of a ALPS to a super-server utility that proportionally schedules multiple Web servers. Section 6 contains a discussion of related works, and we present conclusions in Section 7.

2 Framework

The ALPS scheduling framework is based on a two-level approach, where 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. Any application that requires proportional-share scheduling of its processes will have its own ALPS, and so many ALPSs may be running simultaneously. To simplify exposition, we initially focus on the operation of a single ALPS. Later, we will discuss the behavior of multiple co-existing ALPSs.

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) for those processes under its control*.

An alternative approach is for a user-level scheduler to effectively make all scheduling decisions by selecting at most a single process that is eligible to run, and so by default, the kernel must select that process to actually run. This approach can be inefficient because the user-level scheduler must run every time a process is to be scheduled, yet is not notified when a process yields the CPU (e.g., blocking on I/O). By scheduling a group of processes, ALPS runs less frequently. Furthermore, since the kernel scheduler generally has detailed system knowledge (more than any user-level process can have) regarding pending process I/O requests, processor affinity, etc., ALPS exploits this by scheduling a group of multiple processes, leaving it to the kernel to do what it can do best. This leads to more efficient CPU utilization, as in the case when one of the processes in an ALPS-scheduled group performs I/O; the kernel will schedule one of the other processes to run without requiring the intervention of ALPS.

2.1 Design

ALPS selects multiple processes of the application to run, 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 (Fig. 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 may execute for a fraction of each cycle in proportion to its share. By defining fairness guarantees in this manner, ALPS 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 obtained or exceeded its share. For the duration of each quantum, the processes in the eligible group contend for CPU time from the kernel scheduler (Fig. 2). Just as if ALPS were not present, the task of the kernel scheduler remains to select

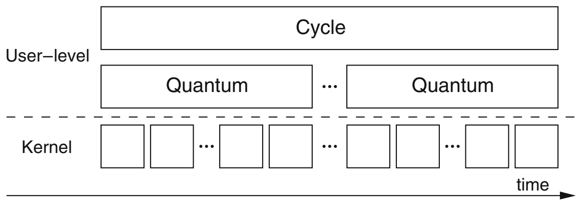

Fig. 1 The kernel scheduler makes decisions at the finest granularity – scheduling single processes. A quantum as defined by ALPS comprises an integral number of smaller kernel quanta; ALPS provides guarantees over a cycle, which comprises an integral number of ALPS quanta

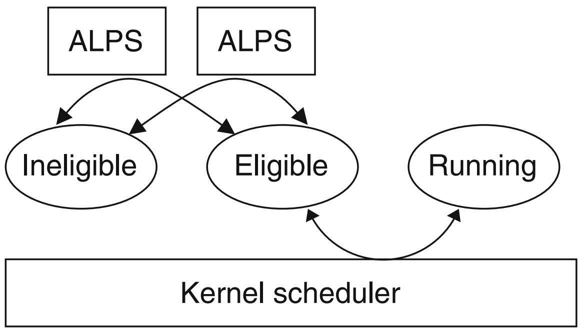

Fig. 2 Each ALPS scheduler (one or more) moves processes under its control between the eligible and ineligible groups; the kernel schedules from the eligible group

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 CPU at a time and must execute between each user-level context switch.

2.2 UNIX-based Implementation

Within the context of UNIX, ALPS can be implemented as a user-level daemon process that does not require administrator privileges to run. ALPS uses mechanisms that are basic to all UNIX-based operating systems, and are fairly common (in some form) to most modern operating systems. An interface is provided to allow an application to register its processes with its associated ALPS.

ALPS uses the `setitimer()` system call to set a periodic real-time timer, and registers a signal handler to execute when the timer expires. The timer duration is set equal to the length of the quantum.[1] The signal handler executes the ALPS scheduling algorithm to classify processes as eligible or ineligible.

To determine the eligibility of a process, ALPS reads a process's execution status from the `proc` file system (i.e., `/proc`). In particular, ALPS reads the amount of CPU time consumed by a process and whether the process is currently blocked inside of the kernel awaiting an event, such as waiting for the completion of an I/O request.

To enact scheduling decisions, ALPS uses signals to transition processes between the eligible and ineligible states. The `SIGSTOP` and `SIGCONT` signals enable ALPS to suspend and resume the execution of a process. When ALPS sends a `SIGSTOP` signal to a process, the kernel suspends the execution of the process rather than delivering a signal. A process can neither block nor catch the `SIGSTOP` signal. The process remains suspended until it is sent a `SIGCONT` signal.

We chose to use signals rather than priorities to influence the kernel's scheduling decisions for two reasons. In many UNIX-like operating systems, a non-privileged user application cannot increase the priority of a process. Furthermore, if ALPS were to raise or lower the priority of its process, then it would affect the share of CPU time that an application receives in relation to other applications. The goal of ALPS is not to affect the proportion of CPU time an application receives in relation to the entire system, but to proportionally distribute CPU time to which an application is naturally entitled amongst the processes of that application.

3 Basic ALPS

To make process selection decisions, ALPS invokes a scheduling algorithm during each quan-

[1]In Sections 3 and 4, we present experiments using quantums of 10, 20, and 40 ms. We chose 10 ms simply because it is the smallest interval allowed by the FreeBSD kernel available to us, and presents the highest potential for accuracy, but overhead as well. Twenty and forty millisecond quantums are used for purposes of comparison. Beyond 40 ms, there are diminishing returns as far as reducing overhead, while accuracy can suffer significantly.

tum. In this section, we describe a simple version of the ALPS scheduling algorithm, called *Basic ALPS*, which is an obvious and perhaps naive way such an algorithm might be designed before optimization. This is for purposes of both exposition, as it will make understanding the optimized version easier, and for comparison, as performance and I/O issues become highlighted in the basic version and then become focal points for the optimized version.

3.1 Description

The central idea of the ALPS algorithm is very simple: each process gets 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 replenishes the allowance of each process in proportion to the process's share.

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, $S \cdot Q$. When a process is registered with ALPS, the process's allowance is initialized to its share (specified during registration) and its state is initialized to ineligible. On account of its positive allowance, the process will become eligible for execution at the next quantum.

When ALPS runs 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.

Algorithm 1 The Basic ALPS Scheduling Algorithm

```
for all $i : state_i = \text{eligible}$ do
    $consumed_i \leftarrow \text{READ-PROGRESS}(i)$
    $allowance_i \leftarrow allowance_i - consumed_i / Q$
    $t_c \leftarrow t_c - consumed_i$
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
        $state_i \leftarrow \text{eligible}$
    else
        $state_i \leftarrow \text{ineligible}$
    end if
end for
```

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 based upon the current allowance of each process. There are two important properties of Basic ALPS that are worth noting, and that will carry over to the optimized algorithm. First, by incrementing the cycle time and allowances (rather than resetting), allocation errors are not accumulated from one cycle to the next. The errors arise because ALPS cannot guarantee that a cycle will end precisely when a quantum expires because the kernel scheduler dictates the CPU time allocated to an application during a quantum. A consequence is that the algorithm is self-correcting, in that any error in target distributions

Table 1 Workload share distributions

	5 processes	10 processes	20 processes
Linear	{1, 3, 5, 7, 9}	{1, 3, 5, ..., 19}	{1, 3, 5, ..., 39}
Equal	{5, 5, 5, 5, 5}	{10, 10, 10, ..., 10}	{20, 20, 20, ..., 20}
Skewed	{1, 1, 1, 1, 21}	{1, 1, 1, ..., 1, 91}	{1, 1, 1, ..., 1, 381}
Total Shares	25	100	400

will be resolved in future cycles. For instance, if a quantization error causes a process to receive twice as much CPU time as entitled in a cycle, then that process will remain ineligible for the duration of the following cycle because its allowance will be negative even after incrementing it by the process's share. Thus, during the subsequent cycle, the process will receive no additional CPU time to correct for the over-allocation in the prior cycle.

Second, the algorithm assigns correct relative amounts of CPU time between processes (specified by the share distribution) regardless of any competing load, i.e., other processes that are unrelated (be they scheduled by some other ALPS or not). As this competing load increases, the real time required for a cycle to expire increases. Yet, as long as the ALPS process is able to execute during each quantum, it will be able to maintain the proper consumed run-time relationship between the processes it is controlling. The time at which ALPS is executed relative to these processes will, of course, depend on the underlying kernel scheduler. We discuss this further below.

3.2 Test Methodology

We evaluate the Basic ALPS algorithm using a single ALPS that schedules workloads (a set of processes) 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 five 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 solely 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.)

In the linear distribution model, the lowest share value is 1, and shares increase by two for each subsequent process: 1, 3, 5, ..., $2n - 1$. A workload that follows the equal distribution consists of processes that all have the same share value: $n, n, \ldots, n$ (n times). In the skewed distribution, all processes but one have a share value of 1, and the remaining process has the remaining shares: 1, 1, 1, ..., 1, $n^2 - (n - 1)$ (i.e., $n - 1$ 1's, with the number of shares for the last process equal to $n^2 - (n - 1)$). Table 1 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 the UNIX-variant FreeBSD 4.8. The machine was connected to the network during the experiments; though to minimize variation caused by external system load (for these initial experiments; later, we include competitive load), we disabled a majority of non-essential services on the machine.

3.3 Accuracy

To evaluate accuracy, ALPS is instrumented to record a log of the CPU time consumed by each process in every cycle. We focus on the steady state operation of the algorithm, and exclude from the following graphs any start-up effects[2] that occur during the initial portion of the experiment (4 cycles).

We first consider a five-process linear workload that is scheduled using a 10 ms quantum. The total number of shares in the workload is 25, and thus the cycle length is 250 ms. Figure 3 depicts the CPU time received by each process. During

[2]These include time to start processes, time for their kernel-level priorities to stabilize from their initial values, etc.

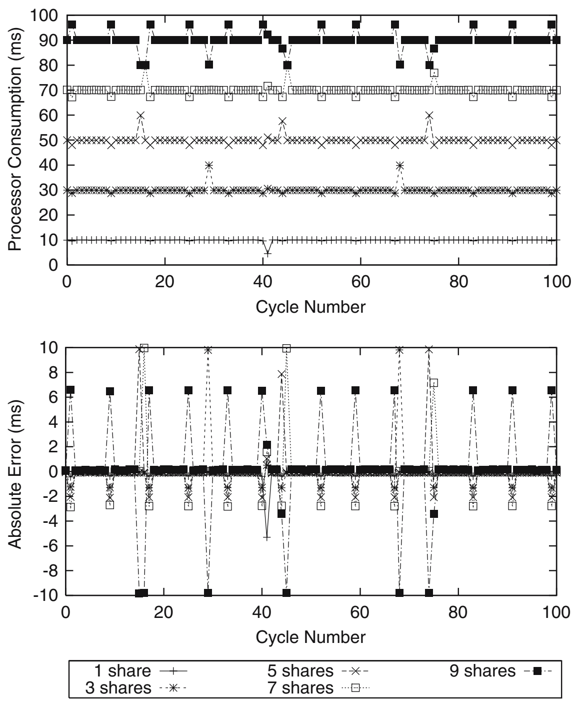

Fig. 3 CPU time received per cycle and absolute error for a five-process linear workload scheduled with a 10 ms quantum

every cycle, the amount of CPU time received by each process is roughly equal to the process's share value multiplied by the quantum length, as is expected.

Figure 3 also shows each process's absolute error at the end of a cycle. We base our calculation of absolute error on the amount of CPU time consumed by all processes during a cycle (as opposed to the length of the cycle in real time). Thus, a process p's *entitlement*, $E_p(n)$, during cycle n is its share of the total CPU time consumed by all processes scheduled by ALPS:

$$E_p(n) = \frac{share_p}{\sum_i share_i} \sum_i consumed_i(n),$$

where $consumed_i(n)$ is the CPU time consumed by a process during cycle n and $share_i$ is a process's share. A process's absolute error is the difference between its consumption and its entitlement during a cycle:

$$AbsErr_p(n) = consumed_p(n) - E_p(n).$$

Notice in Fig. 3 that the absolute error does not exceed the length of one quantum. We believe the cyclic occurrences of small but noticeable error spikes occur due to quantization errors that accumulate.

Because the significance of an absolute error measurement depends on the entitlement of the process, we use relative error to calculate a single value that summarizes the accuracy of the algorithm for a particular workload. We compute a process's relative error at cycle n as follows:

$$RelErr_p(n) = \frac{AbsErr_p(n)}{E_p(n)}.$$

Using the relative errors of each process, we summarize the error at each cycle by calculating the root mean square (RMS) relative error:

$$RMS(n) = \sqrt{\sum_i \frac{RelErr_i(n)^2}{N}}.$$

Figure 4 shows the RMS relative error for the same test presented in Fig. 3. The spike at cycle 41 is a result of the process with 1 share receiving slightly less than 5 ms of CPU time, a relative error of over 50%. More generally, the RMS relative error is very small, less than 0.25%.

To arrive at a single value that represents the accuracy of the algorithm for a particular workload and quantum length, we compute the mean of the RMS relative error over all cycles in an experiment (200 cycles). For example, the mean of the RMS relative errors for the test in Fig. 4 is 2.05%. Figure 5 contains the summarized accuracy of Basic ALPS for various workloads sched-

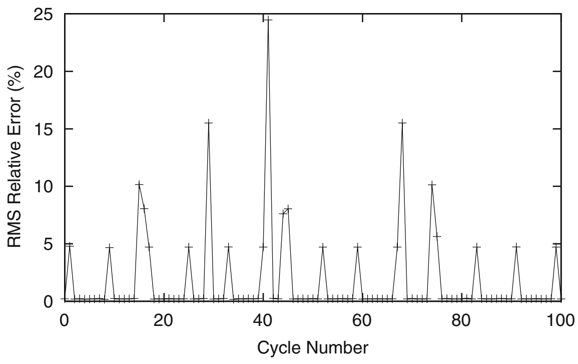

Fig. 4 RMS relative error for five-process linear workload scheduled with a 10 ms quantum

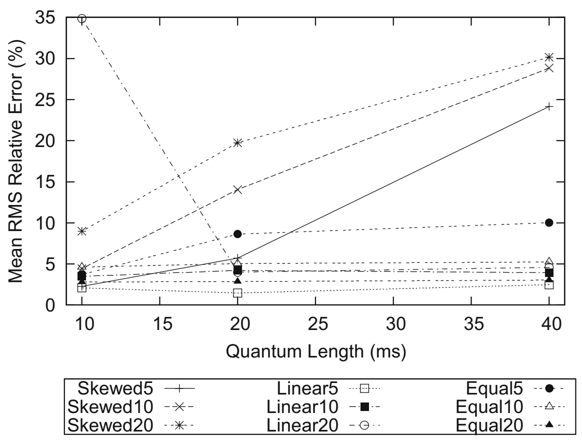

Fig. 5 Mean relative error of Basic ALPS using various quantum lengths

uled at different quantum lengths. Each point is the mean of three tests. For most workloads, the RMS relative error is less than 5%.

Basic ALPS 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. For example, a process that has 1 share may receive slightly less than one quantum of CPU time when scheduled by the kernel. Since its allowance is still positive, albeit much less than a full quantum, the process is still eligible to run. We observe that because low-share processes receive less than their "fair share" over time, the FreeBSD kernel scheduler is likely to select the process to execute for an entire quantum, resulting in a relative error of nearly 100% for the cycle. Though a process with more than 1 share is also susceptible to such quantization errors, the effect on relative error is smaller because the process has a higher entitlement. For instance, the relative error of receiving 6 quanta instead 5 is 20% versus a relative error of 100% when a process receives 2 quanta instead of 1.

Thus, for the skewed workloads, which have a large number of processes with a single share, there is a high probability that in every cycle at least one process will have a large relative error. However, we point out that the algorithm does not favor any single process. For a skewed workload of 20 processes scheduled with a 40 ms quantum, we compute for each process the RMS of the process's per-cycle relative errors over all cycles of the test. The mean of the RMS relative errors of the 19 single-share processes is 31.2%, which agrees with that reported in Fig. 5. The standard deviation of the RMS relative errors of the 19 single-share processes is 3.07%, indicating that ALPS treats each process uniformly. Also, the mean relative error of the 381-share process is 0.36%; it is not susceptible to quantization errors on account of its large number of shares.

The behavior of the algorithm when scheduling the 20-process linear workload using a 10 ms quantum deserves attention. For the majority of cycles, the relative error is in line with the error of the algorithm when using quantum lengths of 20 and 40 ms. However, there are eight times during the experiment where we observe that the FreeBSD kernel scheduler does not execute ALPS promptly at the beginning of a quantum. At these times, ALPS is delayed approximately 400 ms, and the kernel schedules the four lowest-share processes (with shares 1, 3, 5, and 7) for execution. As a result, in eight cycles of the experiment, relative error for the 1-share process is as high as 1901%. Even though this occurs in only 0.01% of the cycles, the error is large enough to raise the mean relative error for the experiment. The FreeBSD kernel scheduler chooses the lower-share processes over ALPS because ALPS performs more work during the cycle than the lower-share processes.

3.4 Overhead

To measure the overhead, we use the `getrusage()` system call to measure the amount of CPU time consumed by a single ALPS during a test run. We calculate overhead as the ratio of the CPU time consumed by ALPS to the wall time of the experiment. We verified our measurements by comparing the amount of work (a loop counter) performed by the workload processes with and without control of ALPS (though we found higher variance in this measurement technique). Figure 6 depicts the overhead versus process count for the linear, equal, and skewed share distributions when ALPS operates with 10, 20, and 40 ms quantum lengths.

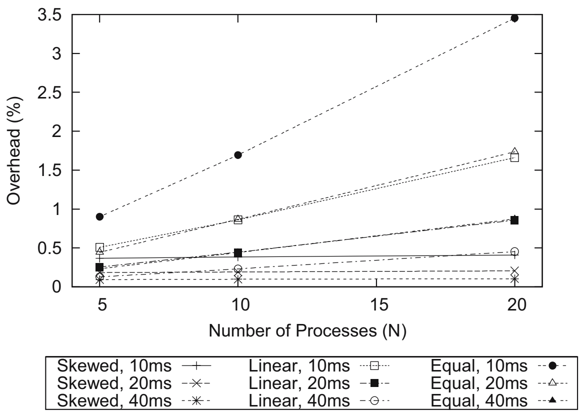

Fig. 6 Overhead of Basic ALPS computed as the fraction of time ALPS executes relative to the duration of the experiment

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 algorithm does not measure the progress of those processes. On the other hand, the processes of the equal share workloads progress at a similar rate. Until the cycle nears completion, few processes exceed their allowance, which results in fewer opportunities for the algorithm to reduce work. As we describe in the next section, the act of measuring a process's progress is the most expensive operation in the scheduling algorithm.

3.5 Analysis of Overhead

The basic operation of ALPS is to wait until a timer event is received, measure the progress of processes under its control, and signal all processes whose eligibility has changed. The time to receive a timer event, I, is a fixed value that is independent of the number of processes that are controlled by ALPS. To measure the progress of a process, ALPS reads the raw status information from a file in the `proc` file system. Let $R(r)$ be the time to perform r reads. After reading the status file, ALPS parses the data and calculates the process's CPU consumption since the last measurement; this computation time per read is ρ. To send a signal, ALPS invokes the `kill()` system call; let S be the time to send a signal. Finally, let $\sigma(N)$ be the time required to execute the loops of Basic ALPS when scheduling N processes (this is all the work required that is not included by the above factors).

The values of I, R, and S depend on the performance of operations provided by the underlying kernel. The values of ρ and $\sigma(N)$ depend on the implementation details of the ALPS algorithm. We measured the values of these factors, which are summarized in Table 2.

Putting it all together, if ALPS performs an average of r reads and sends an average of s signals during a quantum, then the execution time per quantum is given by the following analytical model:

$$T = I + R(r) + \rho \cdot r + S \cdot s + \sigma(N).$$

The overhead is T divided by the quantum length, Q. Using the measured values in Table 2, the model is highly accurate, with a mean relative error of under 0.1% when applied to the 27 workloads presented in Section 3.4.

From this analysis, we see that the primary component of overhead is the time to measure the progress of a process. Each complete read operation (i.e., $R(r) + \rho$) takes between 17.4–18.5 μs per eligible process. Not only does this operation have the highest cost, it increases with the number of processes that are scheduled. In the next section, we describe an optimized version of the ALPS scheduling algorithm that reduces the number of reads without negatively impacting accuracy.

4 Optimized ALPS

We now present our optimizations which reduce overhead while maintaining good accuracy. To re-

Table 2 ALPS operation times (μs)

Receive a timer event (I)	9.02
Read the `status` file ($R(r)$)	$1.1 + 13.9r$
Compute CPU time (ρ)	3.5
Signal a process (S)	0.97
Implementation overhead ($\sigma(N)$)	$1.48 + 0.142N$

duce 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, we add one global variable and one per-process variable. The algorithm uses *count* to index the timer events that it services. 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).

Algorithm 2 The Optimized ALPS Scheduling Algorithm

```
  $count \leftarrow count + 1$
  for all $i$ : $state_i$ = eligible and $update_i \leq count$ do
    $consumed_i \leftarrow$ READ-PROGRESS($i$)
    $allowance_i \leftarrow allowance_i - consumed_i / Q$
    $t_c \leftarrow t_c - consumed_i$
  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
```

4.1 Accuracy

We tested the accuracy of Optimized ALPS using the same workloads and test methodology as for Basic ALPS. Though we do not expect Optimized ALPS to be more accurate than Basic ALPS (since the latter has more information than the former), Optimized ALPS will certainly have less overhead. And so the question is, how much more efficient is Optimized ALPS, and how small is the reduction, if any, in accuracy?

Figure 7 shows the RMS relative error for the nine workloads when using quantum lengths of 10, 20, and 40 ms. Each data point is the mean of three trials of the experiment. The accuracy of Optimized ALPS is virtually the same as that of Basic ALPS, within the margin of error for our measurements. Although Optimized ALPS measures the progress of each process less often, it is only skipping measurements that provide non-essential information. When there are many processes, competition for the CPU prevents any single process from capitalizing on the small opportunities for it to exceed its share. Interestingly, because of its lower overhead, Optimized ALPS actually achieves *higher accuracy* than Basic ALPS in certain cases. Specifically, Optimized ALPS does not exhibit the large error for the 20-process linear workload when scheduled using a 10 ms quantum. The large error measured for Basic ALPS is caused by the FreeBSD kernel not scheduling the ALPS process to run every 10 ms, i.e., because of its higher CPU consumption. Because Optimized ALPS consumes less CPU time per cycle, the ALPS process main-

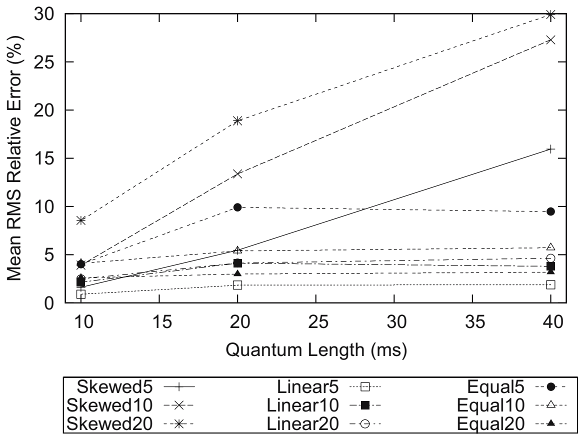

Fig. 7 Mean relative error of Optimized ALPS using various quantum lengths

tains a higher dynamic priority than the workload processes, and the kernel schedules ALPS promptly upon each timer event.

4.2 Overhead

We use the same method to measure the overhead of Optimized ALPS as we use for Basic ALPS. Figure 8 shows the overhead of the optimized algorithm when scheduling workloads at quantum lengths of 10, 20, and 40 ms. In general, Optimized ALPS reduces overhead by a factor of at least 1.8 and as much as 5.9, for the workloads that we tested. (The analytical model that we used to guide the development of the optimized algorithm also accurately estimates the overhead of the optimized algorithm. Over the 27 experiments, the model's mean relative error is 0.73%, with a standard deviation of 1.6%.)

Optimized ALPS achieves the greatest improvement in overhead for the linear workload, ranging between a factor of 2.4 to 5.9. The processes in a linear workload incrementally become ineligible as the cycle progresses, reducing the work performed by ALPS. Though Basic ALPS also benefits from this characteristic of the workload, Optimized ALPS further reduces the number of reads because the processes that remain eligible as the cycle progresses are the processes with larger allowances. Hence, Optimized ALPS can wait several quanta between reads. Part of the improvement can also be attributed to the scheduling policy of the FreeBSD kernel (typical of most UNIX-based systems), which will favor those processes with lower shares because they execute less often relative to the other processes. The net effect is that, as lower-share processes become ineligible during a cycle, the remaining eligible processes have large allowances and the frequency at which the optimized algorithm reads the status of processes decreases. For the equal shares workload, all the processes are eligible for a majority of the quanta in a cycle. In the best case, Optimized ALPS will only need to read the progress of each process once every *N* quanta (with *N* equal to both the number of shares and number of processes). However, in practice, a process will not execute for the entire duration between the points at which the algorithm measures the progress of the process. Still, Optimized ALPS performs many fewer process status file reads compared to Basic ALPS. While the average number of reads per quantum for Basic ALPS increases linearly with the number of processes, Optimized ALPS has a sub-linear increase that enables it to reduce the overhead by a greater factor as the number of processes increases.

The workload for which Optimized ALPS exhibits the smallest improvement is the skewed workload; however, it is still better than that of Basic ALPS by a factor of 1.87 in our experiments. For the skewed workload, all *n* processes compete for CPU time during the first *n* quanta of a cycle,

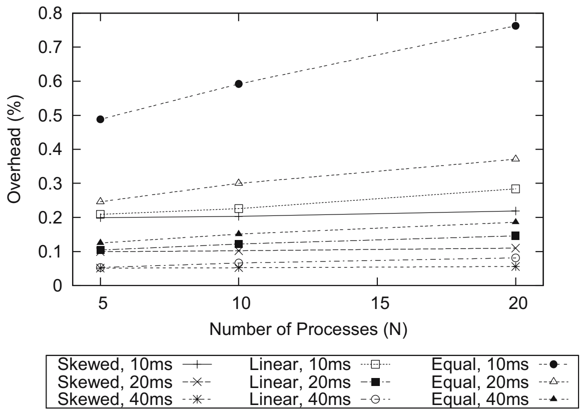

Fig. 8 Overhead of Optimized ALPS computed as the fraction of time ALPS executes relative to the duration of the experiment

after which ALPS will have suspended the $n - 1$ 1-share processes. For the remainder of the cycle, only a single process is eligible. Hence, during the first portion of a cycle, the progress of each eligible 1-share process must be read at every quantum; this occurs for both Basic ALPS and Optimized ALPS. During the latter portion, Basic ALPS reads the status of the remaining process each and every quantum, whereas Optimized ALPS reads the status only once (or possibly a few but small number of times, due to fractional allowances occurring near the end of the cycle).

4.3 Allowing for I/O

The second important optimization is in allowing for I/O. By this, we simply mean that 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.[3] The problem is that at user level, a scheduler lacks precise knowledge of when a process blocks for and resumes from an I/O request. The approach we take is simple, and the only changes to the algorithm occur in the body of the measurement loop (see listing of modified loop).

When ALPS 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 kernel, which indicates the event for which a process is waiting, if any). If ALPS 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 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. 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

Algorithm 3 Modified loop to allow for I/O

```
...
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
...
```

[3]Our goal is to adhere to the ideal that, on average, a process gets its proportional share for each quantum it is ready to run. If it is doing I/O, the remaining processes should compete for the CPU according to their shares with respect to the shares of other processes that are ready to execute (and excluding the shares of processes that are doing I/O). Furthermore, the policy regarding how quickly a process that is returning from I/O gets to run is still primarily dictated by the underlying kernel, as in the priority the kernel provides the process. However, it is true that such a process may be delayed if it has run out of allowance during the current cycle. If achieving fast response is an issue for an I/O-bound process, the user can simply provide it with an artificially large number of shares (most of which will not be utilized if it is truly I/O-bound) to increase its opportunity to run when it completes I/O.

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 the time it was blocked before ALPS detected it as blocked, this simple heuristic seems reasonable, and indeed, works well based on our experiments, which we now describe.

The modifications to allow for I/O do not affect the behavior of the algorithm for compute-bound workloads. To illustrate how the scheduler behaves in the presence of a process that performs I/O, we use a simple workload consisting of three processes, A, B, and C, with a share distribution of 1, 2, and 3, respectively, all under the control of a single ALPS. ALPS uses a 10 ms quantum. After waiting for the processes to reach a steady state of execution, process B begins simulating I/O requests by sleeping for 240 ms after every 80 ms of execution time. Because process B executes at a rate of 33.3% of the CPU, it requires 240 ms of real time to receive 80 ms of CPU time (from ALPS). Therefore, the time process B spends in a ready-to-run state will equal the time spent doing I/O, and it will alternately be ready-to-run for four cycles and be blocked for four cycles. While blocked, we expect ALPS to distribute CPU time in a ratio of 1:3 between the process A and C.

As Fig. 9 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 four cycles that process B is ready-to-run, ALPS continues to maintain the same ratios of 1:2:3. However, while process B is asleep, ALPS distributes 25% of the CPU to process A and 75% of CPU to process C, as expected.

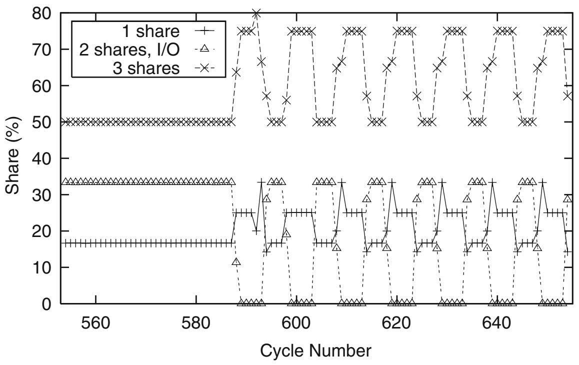

Fig. 9 ALPS proportionally distributes the CPU time available when the two-share process blocks

5 Advanced Experiments

In this section, we present results from experiments that illustrate additional characteristics of Optimized ALPS (simply referred to as ALPS from this point). 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.

5.1 Multiple Applications

The 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 three independent groups of processes, that we label A, B, and C, where each group has three 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 3,000, during which group A processes run exclusively. The second phase then begins, and ends roughly at time 6,000, 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 15,000, during which those in group C run with those of the other groups.

Figure 10 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 the target intra-group relative percentages of CPU time each process should receive based on its shares, and, for each phase, the intra-group relative percentages of CPU time actually received by each process and the relative error. (Each process is identified by the integer that corresponds to its shares *S*.) So, for example, processes 4, 5, and 6, all comprising group B, are targeted to receive 26.7% (4/15), 33.3% (5/15), and 40.0% (6/15), respectively, of the CPU time relative to each other; in Phase 2, they actually receive 27.3, 34.0, and 38.7, resulting in relative errors of 2.2, 2.1, and 3.3%, respectively. Overall, 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

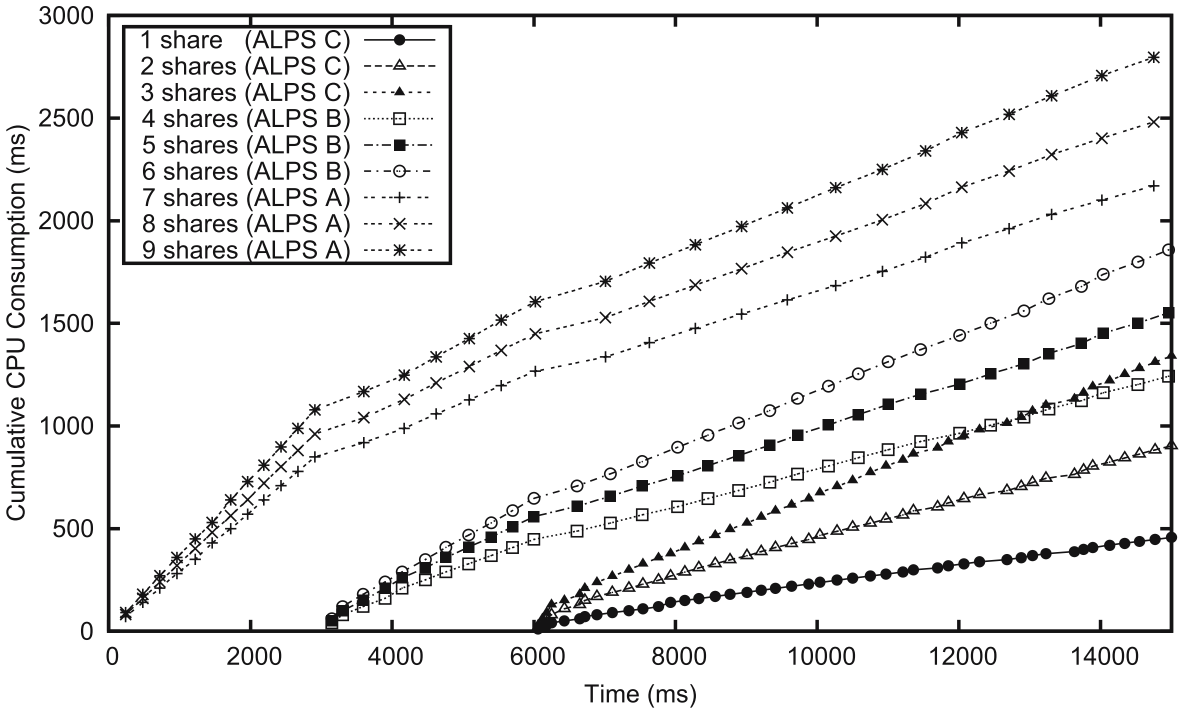

Fig. 10 Cumulative CPU consumption versus wall time for processes scheduled by three distinct ALPSs

Table 3 Accuracy of multiple ALPSs

S	Target	Phase 1		Phase 2		Phase 3	
	%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

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, in 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 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 Fig. 10 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 three 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.

5.2 Scalability

Here we address the question of how many processes (a single) ALPS can schedule before it breaks down (which it will since it runs as a user process which has no special privileges, not even

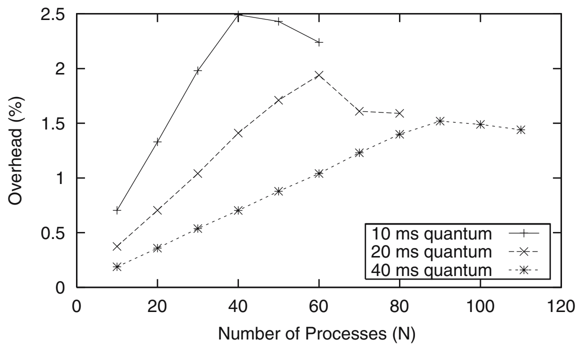

Fig. 11 Overhead for equal share workload

a special higher priority). We use an equal share workload because in the evaluation of Optimized ALPS, the largest observed overhead is for an equal share distribution (see Fig. 8). 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 ms. Figure 11 shows the overhead, and Fig. 12 shows the RMS relative error.

The overhead does not exceed 2.5%, but 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 [19]. 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:

$$U_{10}(N) = .0639N + .0604$$
$$U_{20}(N) = .0338N + .0340$$
$$U_{40}(N) = .0172N + .0160,$$

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 ms, respectively. The observed thresholds are 40, 60, and 90 processes for the same quantum lengths, which match well. With a 40 ms 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 [19]).

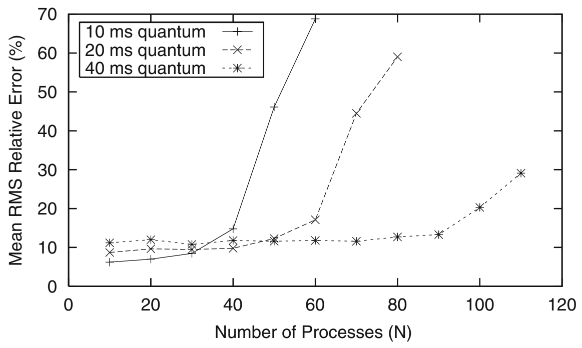

Fig. 12 Accuracy for equal share workload

5.3 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 another key 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, 25]. 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 [24]. 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.

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 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.

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.

The distribution of shares is {1, 2, 3} and the quantum length is 100 ms. 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. ALPS is capable of sharing the processor in the proportion that we desire and consumed only 3.2% of the total processor.

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 [8, 12, 16], and is complementary to our solution for sharing the CPU.

6 Related Work

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, 14]. 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 [9, 13, 15, 17, 21–23, 27, 28]. 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 [26]. Like a kernel scheduler, a user-level thread scheduler can accurately preempt threads (i.e., based on virtual time alarms set within a process), and each thread can notify the scheduler when it is blocked awaiting an I/O request to be serviced. A limitation of a user-level thread scheduler is that it only works in the context of an application designed as a multithreaded process, rather than multiple processes in separate address spaces. 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 [10]. 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 [11], or an understanding of the operation and performance goals of the application [18].

7 Conclusions

We presented the design and implementation of the ALPS application-level proportional-share scheduler, which provides an application with proportional-share scheduling for its processes. Under UNIX, ALPS runs as an unprivileged process, and requires only basic and common kernel functionality as provided by typical UNIX systems. 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. Amza, C., Cecchet, E., Chanda, A., Cox, A., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., Zwaenepoel, W.: Bottleneck characterization of dynamic web site benchmarks. Technical Report TR02-389, Rice University (2002)
2. Amza, C., Cecchet, E., Chanda, A., Cox, A., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., Zwaenepoel, W.: Specification and implementation of dynamic web site benchmarks. In: Proc. IEEE 5th Annual Workshop on Workload Characterization (2002)
3. Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M.: Scheduler activations: effective kernel support for the user-level management of parallelism. In: Proc. 13th ACM Symp. on Op. Sys. Princ. (1991)
4. Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Information and control in gray-box systems. In: Proc. 18th ACM Symp. on Op. Sys. Princ. (2001)
5. Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Burnett, N.C., Denehy, T.E., Engle, T.J., Gunawi, H.S., Nugent, J.A., Popovici, F.I.: Transforming policies into mechanisms with infokernel. In: Proc. 19th ACM Symp. on Op. Sys. Princ. (2003)
6. Banga, G., Druschel, P., Mogul, J.C.: Resource containers: a new facility for resource management in server systems. In: Proc. 3rd Symposium on Operating Systems Design and Implementation. USENIX (1999)
7. Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., Wawrzoniak, M.: Operating system support for planetary-scale services. In: Proc. First Symposium on Network Systems Design and Implementations (NSDI) (2004)
8. Bavier, A., Voigt, T., Wawrzoniak, M., Peterson, L., Gunninberg, P.: SILK: scout paths in the linux kernel. Technical Report 2002-009, Department of Information Tecnology, Uppsala University (2002)
9. Chandra, A., Adler, M., Goyal, P., Shenoy, P.: Surplus fair scheduling: a proportional-share CPU scheduling algorithm for symmetric multiprocessors. In: Proc. 4th OSDI (2000)
10. Chu, H., Nahrstedt, K.: A soft real time scheduling server in UNIX operating system. In: Proc. 4th Intl. Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (1997)
11. Douceur, J.R., Bolosky, W.J.: Progress-based regulation of low-importance processes. In: Proc. 17th ACM Symp. on Op. Sys. Princ. (1999)

12. Druschel, P., Banga, G.: Lazy receiver processing (LRP): a network subsystem architecture for server systems. In: Proc. 2nd Symposium on Operating Systems Design and Implementation. USENIX (1996)
13. Duda, K.J., Cheriton, D.R.: Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In: Proc. 17th ACM Symp. on Op. Sys. Princ. (1999)
14. Engler, D.R., Kaashoek, M.F., O'Toole, J. Jr.: Exokernel: an operating system architecture for application-level resource management. In: Proc. 15th ACM Symp. on Op. Sys. Princ. (1995)
15. Goyal, P., Guo, X., Vin, H.M.: A hierarchial CPU scheduler for multimedia operating systems. In: Proc. 2nd OSDI (1996)
16. Jeffay, K., Smith, F.D., Moorthy, A., Anderson, J.: Proportional share scheduling of operating system services for real-time applications. In: Proc. IEEE Real-Time Systems Symposium. IEEE Computer Society (1998)
17. Jones, M.B., Rosu, D., Rosu, M.-C.: CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In: Proc. 16th ACM Symp. on Op. Sys. Princ. (1997)
18. Lu, Y., Abdelzaher, T.F., Lu, C., Tao, G.: An adaptive control framework for QoS guarantees and its application to differentiated caching services. In: Proc. 10th Intl. Workshop on Quality of Service (2002)
19. McKusick, M.K., Bostic, K., Karels, M.J., Quarterman, J.S.: The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, Reading (1996)
20. Newhouse, T., Pasquale, J.: Java active extensions: scalable middleware for performance-isolated remote execution. Elsevier Computer Communications Journal **28**(14), 1680–1691 (2005)
21. Nieh, J., Lam, M.S.: A SMART scheduler for multimedia applications. ACM Trans. Comput. Syst. **21**(2), 117–163 (2003)
22. Nieh, J., Vaill, C., Zhong, H.: Virtual-time round-robin: an O(1) proportional share scheduler. In: Proc. 2001 USENIX Annual Technical Conf. (2001)
23. Regehr, J., Stankovic, J.A.: HLS: a framework for composing soft real-time schedulers. In: Proc. 22nd IEEE Real-Time Systems Symposium (2001)
24. Slashdot. http://www.slashdot.org/ (2004)
25. Verghese, B., Gupta, A., Rosenblum, M.: Performance isolation: sharing and isolation in shared-memory multiprocessors. In: Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems (1998)
26. von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: scalable threads for internet services. In: Proc. 19th ACM Symp. on Op. Sys. Princ. (2003)
27. Waldspurger, C.A., Weihl, W.E.: Stride scheduling: deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology (1995)
28. Yuan, W., Nahrstedt, K.: Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In: Proc. 19th ACM Symp. on Op. Sys. Princ. (2003)