Extending Proportional-Share Scheduling to a Network of Workstations

Andrea C. Arpaci-Dusseau and David E. Culler

Computer Science Division
University of California, Berkeley
{dusseau, culler}@cs.berkeley.edu

Abstract

As networks of workstations (NOW) emerge as a viable platform for a wide range of workloads, a new scheduling approach is needed to allocate the collection of resources across competing users. In this paper, we show that extensions to a proportional-share scheduler for improving response time can still fairly allocate resources to a mix of sequential, interactive, and parallel jobs in this distributed environment.

We find that a proportional-share scheduler, specifically a stride-scheduler, running on each node in the cluster is a good building-block. Simple extensions are implemented and analyzed which provide better response-times for interactive jobs by giving those jobs their share of resources over a longer time-interval. When scheduling jobs across the cluster, we show that fairness can be guaranteed if each local scheduler knows the number of tickets issued to each user and if the tickets are balanced across all workstations. Finally, we show that a proportional-share of resources can be provided to time-shared parallel applications through a combination of stride-scheduling and *implicit coscheduling*.

Keywords: Networks of Workstations, Clusters, Parallel Computing, Coscheduling, Fairness, Proportional-Share

1 Introduction

The definition of a network of workstations has changed in recent years. In the not-too-distant past, networks of workstations (NOWs) were a loose collection of workstations physically scattered across users' desks [2, 11, 23, 33, 41]. One of the main research issues was effectively exploiting idle resources when users were not running jobs on their desktop machines [1, 4, 34]. Due to the expense of finding available cycles and of migrating jobs if a workstation's owner returned, such systems performed only batch-processing, optimizing for throughput of long-running computationally-intensive applications. Furthermore, the shared-medium Ethernet interconnections and high-overhead messaging protocols necessitated that only sequential and very coarse-grain parallel applications were run.

However, recent advances in low-latency, high-bandwidth switches [8] have enabled networks of workstations to more closely resemble massively parallel processors (MPPs). By connecting these switches and commodity workstations one can build incrementally-scalable, cost-effective, and highly-available shared servers capable of supporting a large number of users running a variety of workloads [3, 6, 7, 10, 20].

Pooling resources has the advantage that resource-intensive applications can use processors, memory, and disks that may otherwise go unused. However, sharing resources raises new challenges in guaranteeing that each user obtains his or her fair share when demand is heavy. Even on a single workstation, most traditional priority-based schedulers permit one user running many processes to acquire more resources; in a NOW, this problem could be exacerbated such that one user consumes the entire cluster. New scheduling and resource allocation policies are therefore needed for our new environment.

To time-share a cluster fairly, one must first be able to fairly schedule jobs on a single workstation. Several variations of proportional-share scheduling have been proposed that solve this problem for compute-bound jobs [15, 16, 31, 37]. We use our implementation of one of these approaches, stride scheduling [38], as a building-block in our evaluations.

However, a tension exists between scheduling users fairly in the cluster and appropriately handling the mixed workload we expect to see in NOWs. In this distributed environment, we must appropriately schedule compute-bound, interactive, multimedia, and I/O-intensive jobs that may each be serial, client-server, or parallel. In this paper, we show that the basic proportional-share scheduler is not sufficient for scheduling such a mixed workload in a distributed environment.

First, one must extend proportional-share scheduling to improve the response time of interactive jobs and those performing I/O, while not penalizing competing users. In this paper, we examine extensions to stride scheduling that give credit to jobs not competing for resources; in this way, jobs are given incentive to relinquish the processor when not in use, and will receive their share of resources over a longer time-interval. Thus, because interactive jobs are scheduled more frequently when they awake, they receive better response time with our extension,

Second, running a proportional-share scheduler on each node in a cluster is not sufficient for allocating resources fairly throughout the cluster. Just as balancing *load* is important for optimizing response-time in a distributed environment, balancing *tickets* is important when providing clients with a proportional-share of resources. We show that when the number of tickets is equal on each workstation and the stride-scheduler running on each workstation is informed of the number of tickets issued to each user, clients running sequential jobs receive their proportional-share.

Finally, we combine these two extensions to fairly schedule time-shared *parallel* jobs in the cluster. We show that a stride-

scheduler that gives credit to jobs that relinquish the processor implicitly coschedules [12] parallel applications, dynamically coordinating time-slices across machines in a completely distributed fashion.

In the next four sections we build towards a NOW which can allocate a proportional-share of resources to a mix of sequential, interactive, and parallel jobs. In each section, we first discuss the limitations of the solutions in previous systems and then describe our proposals and some preliminary measurements of our implementations. We begin in Section 2 by reviewing stride scheduling. In Section 3, we extend stride-scheduling to provide better response time to interactive jobs. In Section 4 we explore scheduling sequential jobs in a distributed environment. Finally, we show that parallel jobs can be proportionately scheduled in Section 5 and conclude in Section 6.

2 Single-Node: Compute-Bound Jobs

Our first step in building a system which fairly allocates the resources in the NOW is to address a relatively simple problem: fairly allocating a single processor among competing users. Fortunately, this problem has already been solved by others; we leverage their results as a building-block as we construct our system.

2.1 Background

Numerous studies have shown that priority-based schedulers are difficult to understand and give more processing time to users with many jobs. Fair-share scheduling was one of the first approaches to address the problem of allocating resources to users fairly over time [18, 19, 21]. However, because these systems are built on top of the traditional priority-based schedulers, they inherit their difficulties and can provide fairness only over relatively long intervals and are difficult to tune.

Proportional-share scheduling appears to provide an adequate solution for our needs, many variations of which have been recently proposed [15, 16, 31, 37]. With proportional-share scheduling, the resource consumption rights of each active process are proportional to the relative shares that it is allocated. We focus on stride scheduling [38] because it is easy to understand and to implement, has been well described and analyzed in the literature, and supports modular resource management.

2.2 Stride Scheduling

Stride scheduling [38] is a deterministic allocation algorithm that encapsulates resource rights with tickets. Like *lottery scheduling* [37], resources are allocated to competing clients in proportion to the number of *tickets* they hold. For example, if a client has t tickets in a system with T total tickets, the client receives t/T of the resources. A *client* may be either a user or a process, depending upon the context.

In stride scheduling, each client has a time interval, or *stride*, inversely proportional to their ticket allocation, that determines how frequently it is scheduled. For example, a client with twice the tickets of another, has half the *stride* and is allocated twice as frequently. As shown in Figure 1, a *pass* associated with each client is incremented by the client's *stride* each time it is scheduled; the client with the minimum *pass* is selected each quantum.

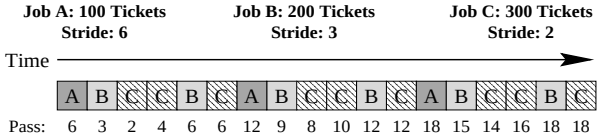

Figure 1: **Stride Scheduling.** *Three jobs compete with a 1:2:3 ticket ratio. The job with the minimum* pass *is scheduled at each interval;* pass *is then incremented by the job's* stride*, which is inversely proportional to its ticket allocation.*

Currencies allow clients to distribute tickets in a modular fashion. By assigning one currency per user, a proportional-share of resources can be allocated to each user, who can in turn allocate a proportional-share of her resources to her processes. For example, if a user has 100 tickets in the base currency, and allocates 300 tickets in her currency to one job and 100 to another job, then the jobs have 75 and 25 tickets each, respectively, in the base currency. Different users can be given different proportions of the CPU simply by issuing them different numbers of tickets in the base currency. In the examples in this paper, we refer to numbers of tickets after they have been translated into the base currency.

We have implemented a version of stride scheduling with ticket currencies as a scheduling class in Solaris 2.5.1, a derivative of UNIX System V Release 4 (SVR4). Measurements show that our implementation incurs negligible overhead relative to the default Solaris time-sharing class. We build upon this basic implementation in the next sections when we investigate supporting interactive jobs and users throughout the NOW.

3 Single-Node: Interactive Jobs

If proportional-share schedulers are to handle the workloads on NOWs, they must deal with more than fairly scheduling CPU-bound jobs: they must provide good response time to interactive jobs and throughput to I/O-intensive jobs. In this section, we motivate the need for extensions to stride-scheduling for interactive jobs on a single-node. We present two approaches for compensating jobs for the time they voluntarily relinquish the processor and evaluate their relative strengths and weaknesses. When we examine scheduling parallel jobs across the cluster in Section 5, we will show that the same extensions are useful.

3.1 Background

It is well known that interactive jobs should be scheduled soon after they awaken. First, this decreases the average wait time in the system, since it is likely that an interactive job will use the processor for only a short time and then go back to sleep. Second, improving the response times of interactive jobs dramatically improves user productivity.

In all current proportional-share schedulers of which we are aware, interactive jobs are treated identically to CPU-bound jobs. The definition of proportional-share scheduling states that clients *actively* competing for resources receive a portion of resources in proportion to their ticket allocations. That is, when a process sleeps on an I/O event, it no longer competes for resources and obviously receives no portion of the processor. When the process wakes up, it is treated as though it never went to sleep and continues to receive its instantaneous proportional-

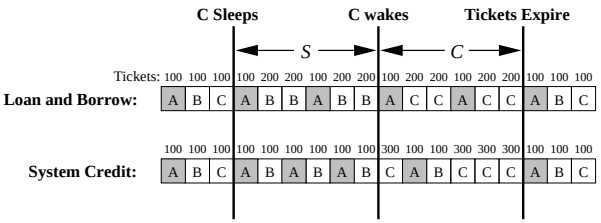

Figure 2: **Loan & Borrow versus System Credit.** *Three jobs with equal ticket allocations are competing for resources. Job A desires a constant service rate, job B is compute-intensive and willing to borrow tickets, and job C is an interactive job. Job C temporarily exits the system, and sleeps for an interval S; in the time-interval C, job C catches up for the time it missed. In both cases, all jobs receive their proportional-share of 6 allocations over the entire interval of 18 time-units; however, only with the* loan & borrow *policy is job A always scheduled 1 out of every 3 time units.*

share.[1] Thus, processes are given no incentive for relinquishing the processor when they perform I/O.

3.2 Sleep Credit

Interactive jobs should receive credit, or additional tickets, for the time that they are asleep, thus receiving their proportional-share of resources over a longer time-interval. We describe two frameworks that improve the response time of interactive jobs while not compromising the fairness of other competing jobs. Both policies build upon *exhaustible tickets* [39], which are simply tickets with an expiration time. In general, when a client's exhaustible tickets expire, the client has caught-up to its desired proportional-share of resources.

In our first approach, *loan & borrow*, exhaustible tickets are traded among competing clients; by keeping the number of tickets in the system constant, particular service rates can be guaranteed to clients. The second approach is an approximation of the first and while it has a much simpler implementation, it is not precise; with *system credit*, clients are given exhaustible tickets from the system when they awake.

3.2.1 Loan & Borrow

In our first policy, when a client temporarily exits the system, other clients can borrow these otherwise inactive tickets. The *borrowed tickets* have an unknown expiration time (*i.e.*, when the client rejoins the system). When the sleeping client wakes after S time units, it stops loaning tickets and is paid back in exhaustible tickets by the borrowing clients. In general, the lifetime of the exhaustible tickets, C, (*i.e.*, the time before the sleeping job catches up to its proportional-share), is equal to the length the original tickets were borrowed, S. The newly awakened client will have positive exhaustible tickets in this interval, C, while the borrowing client will have negative exhaustible tickets. An example is presented in the top half of Figure 2.

The advantage of the *loan & borrow* approach relative to the *system credit* approach, is that the total number of tickets in the system is kept constant over time; thus, clients can accurately determine the amount of resources they receive. For example, soft real-time clients, such as those desiring constant service rates for multimedia applications, or interactive clients that want their full share when they are active, simply decline to borrow tickets. On the other hand, a compute-bound client which is concerned only with throughput can borrow all available tickets. The disadvantage of *loan & borrow* is that, while the implementation is tractable, it introduces an excessive amount of computation into the scheduler on every sleep and wake-up event.

3.2.2 System Credit

In those cases where it is not necessary to support jobs requiring constant service rates, we propose an approximation to the previous policy. The *system credit* policy has the advantage that it is easy to implement and does not add significant overhead to the scheduler on sleep and wakeup events. The bottom half of Figure 2 shows an example.

The idea behind *system credit* is that after a client sleeps and awakens, the scheduler calculates the number of exhaustible tickets for the client to receive its proportional share over some longer interval. In order for a client with t tickets in a system with T total tickets to receive its proportional-share, t/T, in the interval $S + C$, it should be allocated for $A = (t/T) \cdot (S + C)$ units. However, the client is only active for C units in which to receive this allocation; to receive A allocations within C time units, the ratio of the client's tickets $(t + e)$ to the tickets in the system $(T + e)$, must be equal to $\frac{A}{C}$. Therefore, we can calculate the e exhaustible tickets as follows:

$$\frac{(S+C)\cdot\frac{t}{T}}{C} = \frac{t+e}{T+e}$$

$$e = \frac{tTS}{CT-(S+C)t}$$

$$e = \frac{tT}{T-2t} \qquad \text{if } C = S$$

An implicit assumption in this analysis is that there is only one client sleeping and obtaining credit in this interval. The problem when multiple clients re-enter the system is that each client calculates its exhaustible tickets independently of other clients' simultaneous (or later) calculations. Since jobs do not account for the increase in T due to other jobs also introducing exhaustible tickets, jobs may acquire too little of the resources. Correcting for these errors stretches the *system credit* policy beyond its function as an algorithm that is simple to implement with little overhead. Thus, this extension performs most accurately when few clients are simultaneously leaving and joining the system.

A reasonable question to ask for both policies is how to choose the catch-up time, C. If C is longer than the remaining execution time of the client, the client will not acquire its proportional-share before terminating. Similarly, if the client continues to alternate between running and sleeping, then C must be equal or less than the average run interval for the client to use its exhaustible tickets at the same rate it accumulates them. One approach would be to dynamically determine C based on the past history of a job's run lengths. Our current implementations set $C = S$, when possible, to simplify the computations; future work will investigate other options.

[1] Clients leaving the system for multiple time quanta should not be confused with clients relinquishing the processor before the end of their quantum. For example, in basic stride scheduling, clients consuming only a fraction of their quantum are charged for only the amount they used; they receive their full proportional-share only if they are active again at the time of their next allocation.

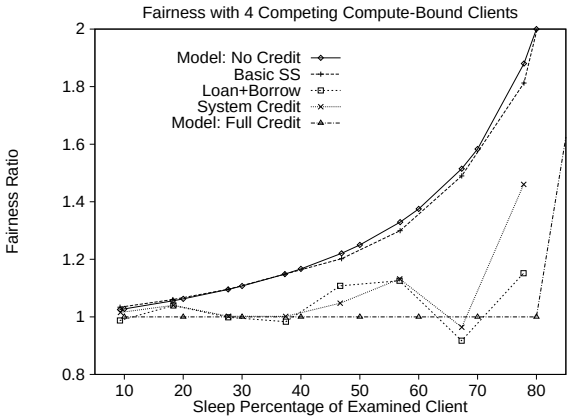

Figure 3: **Comparison of Proportional-Share Scheduling with a Sleeping Client.** *One client which periodically sleeps and runs competes for resources with four clients which always run. Along the x-axis the percentage of sleep time for one client is varied. The y-axis indicates how fair the scheduling was over the lifetime of the sleeping client.*

3.3 Measurements

To demonstrate the basic functionality of our policies, we measure our two prototype implementations built as scheduling classes in Solaris 2.5. In our experiments, we consider five clients with an equal number of tickets. While four of the clients are strictly CPU-bound, the percentage of time one client sleeps is varied.

The evaluation metric we use requires some explanation. In a default proportional-share scheduler, the ratio of the time a client waits to be scheduled, W, to the time the client is scheduled to run, R, equals the ratio of the number of competing tickets in the system, $T - t$, to the number of tickets for this client, t. However, in a time-averaged proportional-share scheduler, when a client sleeps for an interval S, that time in effect counts as wait time, leading to the following equality:

$$\frac{W+S}{R} = \frac{T-t}{t}$$

The fairness metric we use is simply $\frac{W+S}{R} \cdot \frac{t}{T-t}$. In the ideal case, when a client is given full credit for sleeping, the metric is 1. However, in some situations it is not possible for a client to receive its proportional-share. For example, if there is only one client in the system and it voluntarily relinquishes the processor, it can never obtain 100% of the resources. More precisely, a client can not receive its proportional share if $\frac{S}{R} > \frac{T-t}{t}$. In this case, the client's wait time, W, should be zero.

Figure 3 plots two idealized models and three implementations. The bottom line shows our fairness metric as a function of $\frac{S}{S+R}$; this metric is 1 until the client sleeps more than 80%, at which point the client is not active enough to receive its proportional-share. For comparison, the top line in the figure shows the fairness metric when a job is given no credit for sleeping, as in the default stride-scheduler; this line increases slowly with the sleep percentage.

In addition to the two idealized lines, Figure 3 shows the achieved fairness of the default stride-scheduler and of our two extensions for interactive jobs. As expected, our implementation of the default stride-scheduler closely follows the line for when a client is given no credit for sleeping. The *loan & borrow* and *system credit* implementations have similar behavior in this test, because only one job is entering and exiting the system. Both approaches come close to the idealized value of 1, but are more erratic than desired. The values above 1 indicate that the sleeping job achieves less than its proportional-share; values below 1 indicate the sleeping job receives more. We are currently tuning and analyzing our implementations in more detail to correct this behavior.

In summary, both the *loan & borrow* and the *system credit* extensions support time-averaged fairness for interactive jobs. Due to its more complex implementation, *loan & borrow* should only be used when the system must support jobs desiring a constant service-rate, such as multimedia applications. In the remaining sections of this paper, we use the *system credit* extension of stride-scheduling as our building-block.

4 Cluster: Sequential Jobs

Now that we have shown we can allocate a proportional-share of resources to both compute-intensive and interactive jobs on a single workstation, we move our attention to the next problem: time-sharing a proportional-share of resources in a distributed cluster. In this section we consider only collections of sequential jobs; we extend our analysis in the next section to include communicating parallel jobs. We show that clients are guaranteed a proportional-share of resources if the total number of base tickets allocated on each workstation is balanced and if each local stride-scheduler is aware of the number of tickets issued in each currency.

4.1 Background

We begin by briefly reviewing how previous distributed systems insured fairness. Long-term fairness was obtained in Condor by allowing users who have executed fewer jobs in the past to preempt users who have run more jobs [27]. Not only would we like a finer-level of control than the Condor algorithm provides, but we also need to support time-sharing of individual workstations.

On the more theoretical side, micro-economic systems can provide more precise allocation [14, 25]. Their basic idea is that clients have funding which they use to purchase resources; when servers have available capacity, they accept bids and sell their resources to the highest bidder. The major drawback of this approach is the complexity of implementation: very few systems have been built in this manner. Furthermore, due to the high overhead of holding auctions on every scheduling decision, micro-economic system focus on space-sharing and require clients to estimate their run-time.

4.2 Distributing Currencies

One might initially think that simply running a proportional-share scheduler on each workstation in the cluster would give a proportional-share of the total resources to each user; unfortunately, this is not sufficient. Because each local scheduler does not know the total number of tickets issued in a user's currency across the cluster, the schedulers can not convert a job's tickets to the correct number of base tickets. Thus, as shown in Figure 4, users running jobs on more workstations receive more of the total resources. Fortunately, the solution is simple: each local scheduler is informed of the number of tickets issued in each currency, and then can correctly calculate the base funding of each local job.

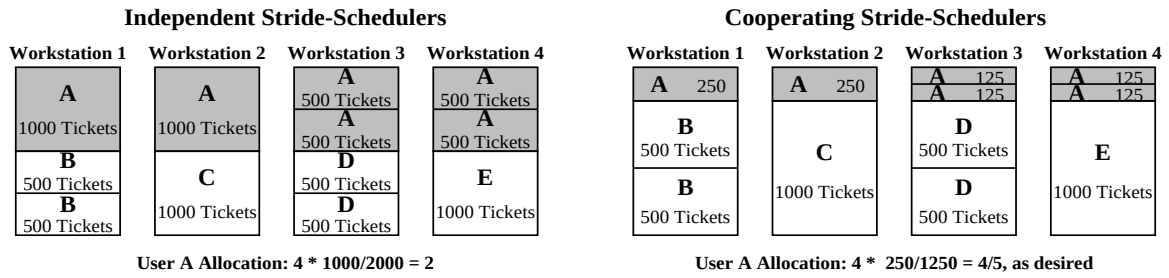

Figure 4: **Fairness in Cluster with Independent versus Cooperating Schedulers.** *With independent proportional-share schedulers, users do not necessarily receive their expected share; e.g., user A receives more resources by running on more workstations. However, if each scheduler knows the number of tickets issued across the cluster, resources can be allocated fairly.*

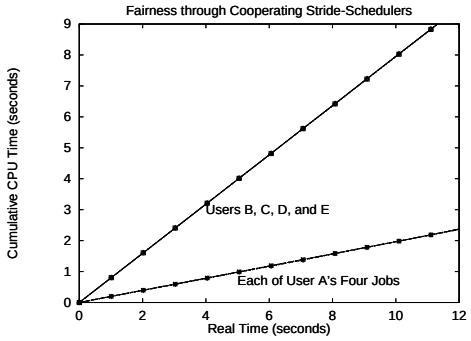

Figure 5: **Measurement of Cooperating Stride-Schedulers.** *This experiment on four UltraSPARCs shows the cumulative CPU acquired by the same four users and sets of jobs as in Figure 4. As desired, users B, C, D, and E each acquire 80% of their workstation and each of user A's jobs acquires only 20%.*

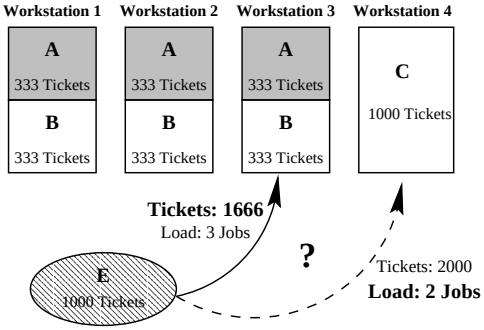

Figure 6: **Tradeoff between Ticket- and Load-Balancing.** *If the system tries to balance tickets, user E's job is placed on workstation 3, receiving 3/5 of the workstation, but slowing down two jobs. If the system balances load, the job is placed on workstation 4, slowing only one other job, but receiving only 1/2 of the workstation.*

Our solution for distributing tickets to the stride-schedulers is to run a user-level *ticket-server* on each of the nodes in the cluster. Using a user-level parallel program has the attractive feature that if the ticket-server crashes, the behavior of the local schedulers degenerates to the case where there is no global information. This behavior is similar to an approach advocated for distributing load information in large-scale shared-memory multiprocessors [9].

Even though our ticket-server uses a fast communication network and a low-overhead messaging protocol [24, 35], it is still important to minimize the amount of communication. Because many processes are expected to be short-lived, each stride-scheduler only periodically contacts the local ticket server to update and determine the value of currencies, rather than every time a job begins or ends. The extensions described in the previous section also reduce communication. In the default proportional-share scheduler, when a client goes to sleep, the other workstations should be notified of the change in issued tickets, thus incurring communication. However, if a client is compensated with exhaustible tickets, the tickets stay active on the local machine.

Figure 5 shows the ability of our parallel ticket to maintain fairness on a cluster of four workstations. The same five competing users are running jobs as illustrated in Figure 4. As desired, the four users running one job a piece each receive $4/5$ of one workstation, while the one user running four jobs receives $1/5$ of each of the four workstations.

4.3 Job Placement by the System

The workstation chosen for executing a process can greatly affect both the average response-time of the processes in the cluster as well as the proportional-share of resources seen by competing users. The ability to migrate jobs has been shown to significantly help balance load, and thus improve average job response-time [17]. Migration is also expected to help ensure that users receive their proportional-share of resources. Unfortunately, space constraints limit our discussion to *systems without migration*. We now briefly discuss our proposal for choosing a destination workstation for a process.

In the previous subsection, we showed an example where knowing the number of tickets issued in each currency enabled the schedulers to allocate the correct share to each client; however, this information is not always sufficient. After a job has been placed, tickets compete only with other tickets on the same machine. The result is that tickets on nodes with less competition are worth more than tickets on nodes with more competition. This effect is very similar to that in load-balancing, where jobs on machines with fewer competing jobs receive more processing time.

To guarantee that clients receive their proportional-share, not only must the system track the number of tickets issued in each currency, but also the number of tickets per machine must be balanced. This is simple to show. If there are N machines and U users each with F tickets, and if there are an equal number of tickets, T, on each machine, then by definition

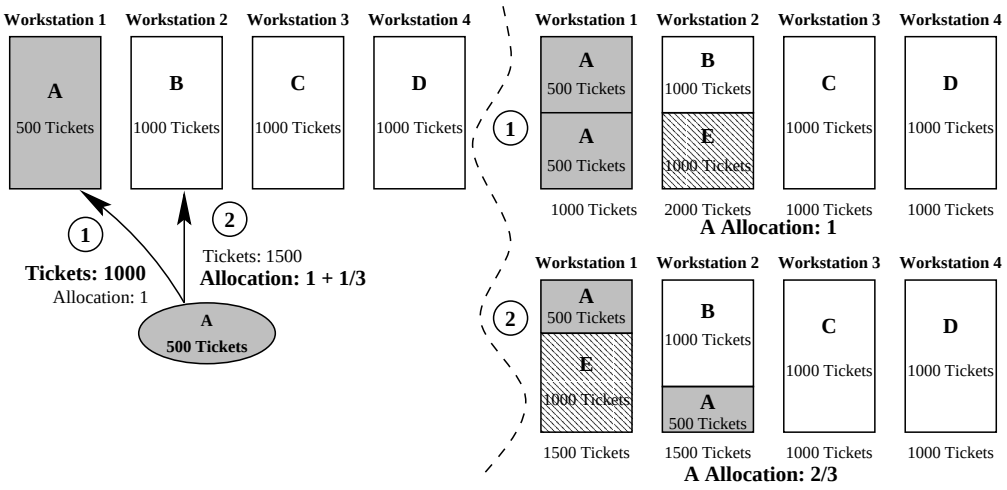

Figure 7: **Impact of a Greedy User in the Cluster.** *On the left side, user A can temporarily receives more than his share of resources by ignoring option 1 which balances ticket, and instead choosing option 2 which does not force his jobs to compete with one another. However, the right side shows that when a new user E arrives, user A will be better off if he originally chose the first option of balancing tickets.*

$T = \frac{U \cdot F}{N}$. Given that the F tickets of a given user are distributed across the nodes in the cluster such that f_i tickets are on node i, then each user receives the desired allocation of N/U:

$$\frac{f_1}{T} + \frac{f_2}{T} + \frac{f_3}{T} + ... + \frac{f_N}{T} = \frac{F}{T} = \frac{F}{\frac{U \cdot F}{N}} = \frac{N}{U}$$

As expected, a tension often exists between balancing tickets and balancing load: *i.e.*, between achieving fairness and optimizing per-process response times. Placing a job on the machine with the least competing tickets leads to the best allocation of a proportional-share of resources; placing a job on the machine with the least load gives the best response-time averaged over all jobs. An example is shown in Figure 6. In keeping with our goals, our system optimizes for fairness over response time by first trying to balance tickets, and then comparing load in the case of ties.

An important related issue is whether ticket counts and load information are kept in a centralized location or are available only from the originating workstations. A centralized location is not a realistic design decision when building a scalable, fault-tolerant cluster on the scale of hundreds of workstations. A promising approach for our environment is for each client to randomly query k nodes and pick the one best qualified [13, 26, 40]. Future work will involve investigating this distributed approach, as well as the impact of migration.

4.4 Impact of Job and Ticket Placement by Users

We have described our approach for placing jobs in the cluster when the system undertakes this role. However, users may also specify for themselves the workstation on which to execute a process. This functionality is required so that users can compute on machines near file data [5], can use special devices, can run diagnostics, or explicitly avoid faulty machines. We now briefly describe the impact on competing clients of this level of control.

If users can execute their processes on arbitrary machines, users can temporarily receive more than their proportional-share. One may wonder how this occurs, since an individual job receives the most resources by running on nodes with the least competing tickets, which is how the system places jobs. The difference is that a "greedy" user may consider the impact of this job on his other jobs and not place the jobs in competition on the same machine. As shown in Figure 7, the user can maximize his *total* allocation by minimizing the number of tickets with which *each* of his jobs competes. However, as shown in the second half of the figure, because this greedy allocation does not balance tickets, later arriving jobs from competing users are placed in competition with the greedy user. As a result of this competition, the tickets in the system move towards being balanced.

In addition to specifying job placement, clients should also be able to specify the distribution of tickets across their jobs. This functionality is important so that users can prioritize the tasks that are more important to them. However, this capability can further exacerbate temporary biases in allocation. Greedy users can allocate fewer tickets to those jobs running on machines with less competition and more to those with more competition, and thus receive more resources. Once again, later arriving jobs from competing users will be placed on the nodes with fewer tickets, and the greedy user will no longer receive more than his share. The job that was allocated few tickets will now receive a relatively small proportion of the resources, thus reducing the incentive to greedy users for performing unfair ticket allocation.

One remaining issue is whether or not users should be able to dynamically readjust ticket allocations after a job has been placed. This functionality would encourage greedy users to continuously perform the uneven ticket allocations described above. In this case, after the greedy user has allocated only a few tickets to a job with little competition, the greedy user can reallocate more tickets when competing jobs are drawn to those nodes. Therefore, dynamic ticket readjustments may not lead to a proportional-share across users. However, they could be allowed in a system with migration since competing jobs

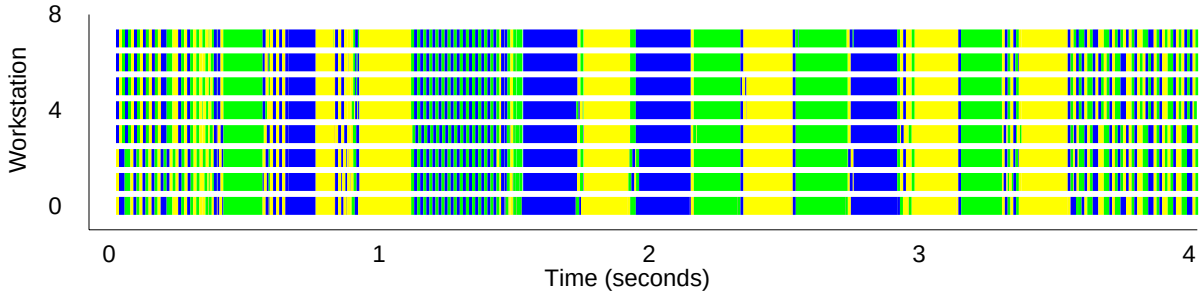

Figure 8: **Implicit Scheduling and Solaris Time-Sharing Scheduler.**

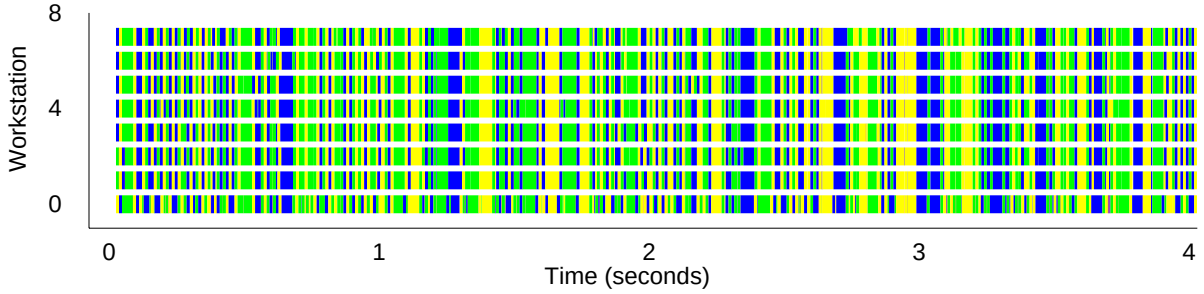

Figure 9: **Implicit Scheduling and Stride Scheduler with Sleep Credit.** *Three parallel jobs each consisting of 32 processes compete for resources on 32 workstations; only the first 8 workstations and the first few seconds are shown, but the other data points are similar. The three jobs each perform barriers approximately every 10 ms.*

can always move to the nodes with fewer tickets.

In conclusion, users should be able to specify the placement of their jobs in the cluster and the distribution of tickets. While this level of control allows users to temporarily receive more than their proportional-share, later arriving users will place jobs in a manner that levels out the discrepancies. Future work will investigate these interactions more rigorously.

5 Cluster: Parallel Jobs

In the previous section, we showed that a proportional-share of resources can be allocated to clients running sequential jobs in a network of workstations. In this final step, we combine stride-scheduling and implicit coscheduling [12] to extend our analysis to parallel jobs.

5.1 Background

We begin by discussing how parallel jobs are scheduled and how fairness is provided in other distributed systems. The most relevant example of an implementation is Spawn [36], a computational economy for concurrent applications in a distributed network. However, because the authors only consider coarse-grain applications, such as Monte-Carlo simulations, they do not simultaneously allocate multiple nodes to a single client. Their implementation shows one of the main drawbacks of such market-driven approaches: their measurements reveal a 10.3% overhead with a time-slice of 60 seconds. Further, their implementation requires processes to have an accurate approximation of their lifetime and does not allow time-sharing of workstations.

A more recent simulation study allows applications to purchase time on multiple nodes simultaneously, thus supporting frequently communicating parallel applications that must run concurrently [32]. However, they also support only space-sharing and applications must predict their lifetime. An implementation of their system appears to need a centralized auction.

In general, resource allocation in a parallel system can be performed by either space-sharing or time-sharing. Space-sharing is a popular approach for optimizing the throughput of compute-bound, production-level jobs; however, it is not appropriate for everyday, developmental workloads due to large queuing delays. Space-sharing also does not efficiently handle interactive, or I/O-intensive workloads, because when a job is blocked, waiting for I/O to complete, the processor remains idle.

When time-sharing parallel jobs, traditional MPPs use explicit *coscheduling*, or gang scheduling, to coordinate communicating processes across nodes. By scheduling the processes of one job at the same time on each node, each parallel application is given the impression of a dedicated machine [28]. Coscheduling, in contrast to independent time-sharing across nodes, allows programs with fine-grain synchronization and communication to avoid busy-waiting and context-switching.

Unfortunately, explicit coscheduling has a number of disadvantages that are accentuated in the NOW environment. First, a straight-forward implementation of explicit coscheduling requires a centralized master to determine a fixed schedule for running parallel jobs at the same time across workstation. Ensuring that context-switches occur simultaneously on all processors increases the cost of each context-switch. Second, explicit coscheduling does not interact well with interactive jobs or parallel jobs performing I/O [22]. Finally, explicitly coscheduling requires that communicating processes are identified *a priori*, and so can not be applied to distributed client/server applications.

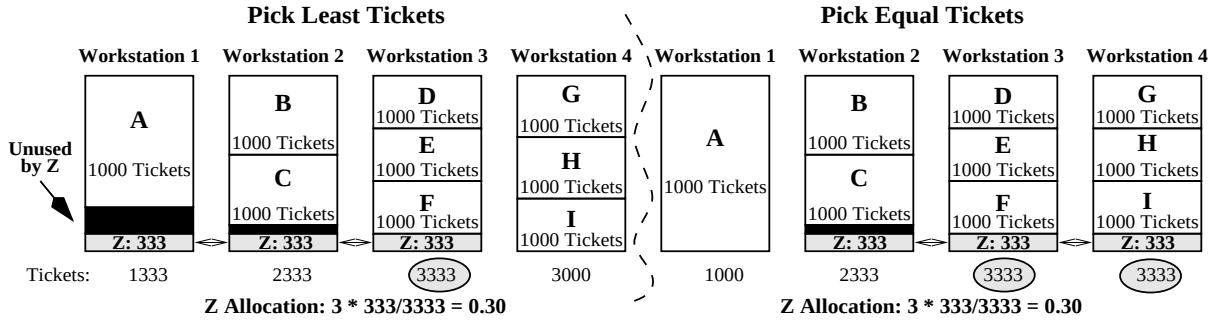

Figure 10: **Placing Fine-grain Parallel Jobs in the Cluster.** *User Z's parallel job cannot proceed faster than the slowest node: workstation 3. Therefore, Z receives the same allocation whether it runs on machines with less or equal competing tickets relative to workstation 3. However, to best balance tickets and give more users closer to their proportional-share (i.e., users G, H, and I), the parallel job should still be placed on the three nodes with the least tickets.*

5.2 Implicit Coscheduling

A promising time-sharing approach for scheduling parallel applications in a cluster is *implicit coscheduling* [12], or dynamic coscheduling [29, 30]. Both of these approaches appear to have many of the advantages of explicit coscheduling, without the disadvantages. Unlike explicit coscheduling, both are completely distributed, requiring no global coordination; both have potential for working well with a mix of parallel and interactive jobs and parallel jobs performing I/O; finally, neither requires that communicating processes be statically identified.

Simulations have shown that *implicit coscheduling* [12] effectively schedules both coarse- and fine-grain bulk-synchronous parallel applications. Implicit coscheduling avoids the problems of traditional coscheduling by using communication and synchronization occurring naturally within the application to coordinate scheduling across workstations.

There are two events that must be observed and acted upon for processes to implicitly coschedule themselves. First, if a process sends a message and then waits longer than the round-trip time of the network plus the time to switch to the destination process, then the sender can infer that the other processes in this job are probably not currently scheduled; consequently, this process should relinquish the processor by going to sleep. Second, if a process receives a message, it can infer that other processes in this application are currently scheduled; consequently, in some circumstances, this process should also be scheduled. The previous simulations [12] found that waking up processes waiting for message responses was sufficient; it was not necessary to schedule processes waiting in the ready queue.

An important assumption in previous simulations was that each node was running the Solaris time-sharing scheduler. Time-sharing schedulers reward jobs that frequently sleep by raising their priority when they awake, on the assumption that they will sleep again. In Solaris 2.5, this priority raise occurs if the process was sleeping when a one-second periodic timer expired. Thus, as desired, these mechanisms usually schedule a blocked destination process when a message arrives.

In contrast, processes are not scheduled with any additional probability after sleeping with the default stride-scheduler. Therefore, basic stride-scheduling does not lead to robust implicit coscheduling: simulations in [12] showed that a round-robin scheduler (which is the same as the basic stride scheduler when all processes have equal tickets) led to substantially reduced performance compared to the time-shared priority scheduler.

Fortunately, the extensions to stride-scheduling presented in Section 3 for giving credit to sleeping jobs are applicable to communicating jobs as well. When a parallel job wakes up due to the arrival of a message, the job is given exhaustible tickets and is thus more likely to be scheduled. Preliminary simulations of implicit coscheduling for a range of communication patterns and computation granularities indicate that the stride-scheduler with *system credit* performs similarly to the Solaris time-sharing scheduler.

Partial traces from one of those simulations, shown in Figures 8 and 9, illustrate that both local schedulers do implicitly coschedule the jobs across workstations; currently, the time-sharing scheduler tends to schedule jobs for longer time-slices, leading to fewer context-switches and slightly better throughput. Increasing the default time-slice of the stride-scheduler does not fix this inefficiency; we are currently in the process of understanding these results more thoroughly.

The advantage of using a stride-scheduler as the building-block in the cluster is that a proportional-share of resources can be allocated to parallel jobs as well. Our initial simulations indicate that the achieved level of control is good, but not as precise as when scheduling sequential jobs. For example, when tickets in a $1:2:3$ ratio are allocated to three competing parallel jobs that synchronize every 10 ms, an actual ratio of $1:1.8:2.7$ is achieved. We believe these preliminary results are very promising.

5.3 Job Placement

The previous discussion centered around scheduling parallel applications once they have been placed on nodes in the cluster. Of course, first the system must place the processes of the parallel job on different workstations. As when placing sequential jobs, the system should select nodes with the fewest competing tickets. However, placing parallel jobs involves additional considerations. Because fine-grain parallel jobs run at the rate of the process on the slowest machine [4], the job receives the same amount of resources whether all machines have exactly T competing tickets, or one machine has T tickets and all others have fewer. Therefore, the system should be extra careful when placing parallel jobs as to not pick one machine with many more competing tickets than the others.

As shown in Figure 10, while the parallel job executes at the same rate whether all nodes have the same number of tickets or not, the impact on the competing jobs differs. From

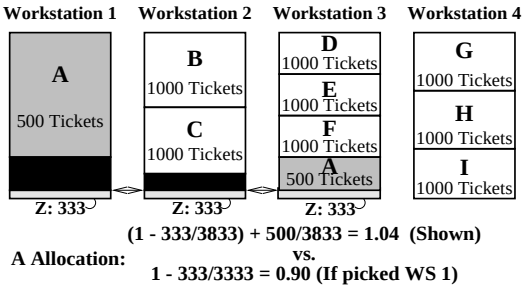

Figure 11: **Impact of a Greedy User on a Parallel Job.** *User A is able to obtain more processing time on workstation 1 by placing another job on workstation 3, the machine with most tickets competing against the parallel job, Z.*

the viewpoint of the system, it is best to allocate parallel jobs starting with the machines with the least number of tickets; in this way, tickets are balanced as closely as possible and more clients receive their proportional-share.

Parallel jobs also complicate the behavior of users who control the placement of their own jobs. With implicit coscheduling, when a parallel job on a faster node waits for a slower node, the faster process sleeps and relinquishes the processor; with stride-scheduling, the unused time is then divided between the other clients competing on this workstation. Therefore, a greedy user with a sequential job in competition with a parallel job can receive more resources on that machine by placing new jobs on the most heavily-loaded node running the parallel job. This counter-intuitive result is illustrated in Figure 11. However, as was the case when we considered only sequential jobs, later arriving users balance the allocations as they are placed on nodes with fewer tickets.

6 Conclusions and Future Work

The shared resources in a Network of Workstations have enabled a new set of workloads to coexist: sequential, interactive, multimedia, client/server, and parallel jobs. This new workload and this environment require new approaches for fairly and efficiently allocating resources to competing users. Previous work for providing fairness in clusters has made a number of assumptions that are no longer acceptable: e.g, providing only a coarse-level of control over allocations, requiring a centralized master, forcing users to accurately estimate the lifetime of their jobs, and/or only supporting space-sharing.

In this paper, we have shown that we can extend stride-scheduling [38] to provide a proportional-share of resources to users running a mixed workload in a distributed environment. First, we found that giving credit (i.e., exhaustible tickets [37]) to jobs that relinquish the processor, rewards jobs that periodically sleep while not harming other competing clients. Second, we demonstrated one way to extend proportional-share scheduling to a distributed environment: the number of tickets issued by each client must be known to each local scheduler and the number of tickets must be balanced across workstations.

By combining these two extensions, a proportional-share of resources can be allocated to parallel jobs as well. Preliminary simulation results indicate that stride-scheduling can implicitly coschedule [12] a proportional-share of resources to competing parallel jobs.

We have begun to explore the fundamental trade-offs between achieving fairness across users and optimizing throughput in a NOW. The system must often choose between balancing tickets (*i.e.*, fairness) and balancing load (*i.e.*, throughput) when placing jobs. More work needs to be performed to better understand this trade-off, as well as the impact of job and ticket placement by "greedy" individuals. We also expect that the presence of migration and the ability to dynamically readjust ticket allocations will dramatically impact the appropriate policies.

The next steps of this work involve incorporating the described policies into the U.C. Berkeley NOW [3, 10]. Key components of the implementation work include extending the parallel ticket server so it is robust to node failures and placing jobs and tickets without the current centralized master. We will also soon measure our implementation of implicit coscheduling on Active Messages [24, 35] with both the Solaris time-sharing scheduler and our modified stride-schedulers.

Acknowledgments

We would like to thank Tom Anderson, Remzi Arpaci-Dusseau, and Amin Vahdat for their many helpful comments on the presentation of this work. Andrea Arpaci-Dusseau is currently supported by an Intel Foundation Graduate Fellowship. This work is also supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C0014), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Hewlett Packard, Microsoft, and Mitsubishi.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The Utility of Exploiting Idle Workstations for Parallel Computation. In *Proceedings of 1997 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Seattle, June 1997.

[2] R. Agrawal and A. Ezzat. Processor Sharing in Nest: A Network of Computer Workstations. In *Proceedings of 1st International Conference on Computer Workstations*, Nov. 1985.

[3] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, Feb. 1994.

[4] R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of ACM SIGMETRICS'95/PERFORMANCE'95 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. P. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference*, 1997.

[6] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, Dec. 1995.

[7] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the International Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

[8] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Sietz, J. Seizovic, and W. Su. Myrinet – A Gigabit-perSecond Local-Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[9] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 12–25, Dec. 1995.

[10] D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel Computing on the Berkeley NOW. In *Ninth Joint Symposium on Parallel Processing*, Kobe, Japan, May 1997.

[11] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–85, Aug. 1991.

[12] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of 1996 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, 1996.

[13] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[14] D. Ferguson, Y. Yemini, and C. Nikolaou. Microeconomic Algorithms for Load Balancing in Distributed Computer Systems. In *International Conference on Distributed Computer Systems*, pages 491–499, 1988.

[15] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[16] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[17] M. Harchol-Balter and A. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. In *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 13–24, May 1996.

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

[19] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, Oct. 1984.

[20] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–18, Nov. 1993.

[21] J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 31(1):44–55, Jan. 1988.

[22] W. Lee, M. Frank, V. Lee, K. Mackenzi, and L. Rudolph. Implications of I/O for Gang Scheduled Workloads. In *Proceedings of the IPPS '97 Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[23] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[24] A. M. Mainwaring. Active Message Application Programming Interface and Communication Subsystem Organization. Master's thesis, University of California, Berkeley, 1995.

[25] T. Malone, R. Fikes, K. Grant, and M. Howard. *Enterprise: A market-like task scheduler for distributed computing environments*, pages 177–205. North-Holland, 1988.

[26] M. Mitzenmacher. Load balancing and density dependent jump Markov processes. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 213–222, Oct. 1996.

[27] M. Mutka and M. Livny. Scheduling Remote Processing Capacity In A Workstation-Processor Bank Network. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 2–9, Sept. 1987.

[28] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[29] P. G. Sobalvarro, S. Pakin, W. E. Weihl, and A. A. Chien. Dynamic Coscheduling on Workstation Clusters. Available from http://www.research.digital.com/SRC/scheduling, 1997.

[30] P. G. Sobalvarro and W. E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 63–75, Apr. 1995.

[31] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. G. Plaxton. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *IEEE Real-Time Systems Symposium*, Dec. 1996.

[32] I. Stoica, H. Abdel-Wahab, and A. Pothen. A Microeconomic Scheduler for Parallel Computers. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–135, Apr. 1995.

[33] M. Theimer, K. Landtz, and D. Cheriton. Preemptable Remote Execution Facilities for the V System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 2–12, Dec. 1985.

[34] M. M. Theimer and K. A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444–57, Nov. 1989.

[35] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[36] C. Waldspurger, T. Hogg, B. Huberman, J. Kephart, and S. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, Feb. 1992.

[37] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–11. USENIX Association, 1995.

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

[39] C. A. Waldspurger and W. E. Weihl. An Object-Oriented Framework for Modular Resource Management. In *5th Workshop on Object-Orientation in Operating Systems (IWOOOS '96)*, Oct. 1996.

[40] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. Technical Report UCB/CSD 87/305, Computer Science Division, Univerisity of California, Berkeley, Sept. 1986.

[41] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computing Systems. Technical Report CSRI-257, University of Toronto, 1992.