Market-based Proportional Resource Sharing for Clusters

Brent N. Chun and David E. Culler

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

Abstract

Enabling technologies in high speed communication and global process scheduling have pushed clusters of computers into the mainstream as general-purpose high-performance computing systems. More generality, however, implies more sharing and this raises new questions in the area of cluster resource management. In particular, in systems where the aggregate demand for computing resources can exceed the aggregate supply, how to allocate resources amongst competing applications is an important problem. Traditional solutions to this problem have focused mainly on global optimization with respect to system-centric performance metrics, metrics which ignore higher level user intent. In this paper, we propose an alternative market-based approach based on the notion of a computational economy which optimizes for *user value*. Starting with fundamental requirements, we describe an abstract architecture for market-based cluster resource management based on the idea of proportional resource sharing of basic computing resources. Using this architecture, we have implemented a 32-node (64 processors) prototype system that provides a market for time-shared CPU usage for sequential and parallel programs. To begin evaluating our ideas, we are currently in the process of studying how users respond to the system by collecting data on real day-to-day usage of the cluster.

1 Introduction

Enabling technologies in high speed communication [3, 4, 7, 23, 37] and global process scheduling [2] have pushed clusters of computers [1, 6, 8, 27] into the mainstream as general-purpose high-performance computing systems. No longer restricted to strict space-sparing as personal supercomputers, clusters today are capable of supporting multiple parallel and sequential jobs concurrently and delivering high performance as well. More generality in the system, however, implies more sharing and this raises new questions in the area of cluster resource management. In particular, in systems where the aggregate demand for computing resources can exceed the aggregate supply, how to allocate resources amongst competing applications is an important problem.

Traditional solutions to cluster resource management [5, 10, 13, 15, 18, 39] have mainly focused on global optimization with respect to system-centric performance metrics (e.g, mean job completion time, average system utilization). For systems with significant sharing, though, these approaches present two problems. First, global optimizations are not targeted to be consistent with the individual resource valuations of the users. Optimizations are performed as though all applications are equally important[1] while ignoring individual user value of the resources which vary based

[1]Some schedulers, such as the Unix priority scheduler, make implicit assumptions about user value and thus do not treat all applications equally. For example, in observing the fraction of a job's CPU scheduling quantum used, priority schedulers attempt to infer whether a job is "interactive" or "compute-bound" and modifies the job's priority in accordance with what it thinks will result in "better" performance.

on the immediacy, importance, and the resource demands of the user's computing needs. Thus, in allocating resources to competing applications, these solutions are unlikely to deliver the greatest value to the users for a given set of resources. Second, traditional solutions lack proper incentives to encourage users to back off the system during periods of high contention. In systems where aggregate demand can exceed aggregate supply, the probability of overcommitting cluster resources is non-negligible. With users pursuing selfish, locally-optimal behavior, this can be a problem during periods of high contention as users have no incentive to perform socially desirable behavior to improve global goodness.

To address these issues, we propose a market-based approach to cluster resource management based on the notion of a computational economy which optimizes for *user value*. In this scheme, clusters are organized as economies of independent sellers and buyers. Cluster nodes act as independent sellers of computing resources and user applications act as buyers who purchase resources based on the personal value delivered to users. Users express value using a currency which can be traded for resources and are charged for resource use based on the the rate they are willing to pay and the rate other users are willing to pay on competing applications. Starting with fundamental requirements, we develop the idea of computational economies by presenting an abstract architecture for market-based cluster resource management based on the idea of proportional resource sharing of basic computing resources. We then describe an implementation of this architecture, a 32-node (64 processors) prototype system that provides a market for time-shared CPU usage for sequential and parallel programs. To begin evaluating our ideas, we are currently in the process of studying how users respond to the system by collecting data on real day-to-day usage of the cluster.

The rest of this paper is organized as follows. In Section 2, we present the three fundamental functional requirements for market-based systems. In Section 3, we present an abstract architecture for market-based cluster resource management based on proportional resource sharing. Using this architecture, in Section 4, we describe an implementation of an initial 32-node (64 processors) market-based cluster which provides time-shared CPU usage for sequential and parallel programs. Section 5 discusses related work and in Section 6, we present some conclusions.

2 Three Fundamental Requirements

There are three fundamental functional requirements for market-based resource management on a cluster: (i) a means for users to express value, (ii) policies to translate value into resource allocations, and (iii) mechanisms to enforce resource allocations. Variations in how these requirements are met defines the design space of market-based systems.

2.1 Expressing Value

For a market-based system to optimize for user value, it must first determine what is the personal user value for each application competing for shared resources. Without this knowledge, the system has no meaningful way of inferring which applications are more valuable when deciding how to allocate resources. Thus, the first component needed is a means to concretely express a user's personal value in executing each application on a system. There are two parts to this: the medium of expression used to express value and the entity (or entities) to which value is being assigned.

When running an application, each user has some implicit notion of how important the execution of an application at some instant in time is to them. For example, a user running Netscape is likely to value its execution less than the execution of an important simulation whose results need to be

included in a paper due later the same day. To make this notion of value concrete to the system, there needs to be a mapping from personal user value to some common medium of expression for value. A common solution to this problem is to use a common currency (e.g., US$). As long as users can form reasonably accurate mappings from personal value to a currency, this solution is adequate.

The second part of expressing value concerns to what entity or entities value is being assigned. For example, when a user says that a simulation is important, perhaps the thing that is valued is the simulation's wall-clock completion time. On the other hand, when a user runs a server, perhaps the user values the average requests per second it can sustain. Ideally, the entities to which the user assigns value would be expressed in the vocabulary of the application in terms the user deals with.

Forming the mapping between application-specific performance metrics and low-level resources can be solved, in some cases quite accurately, for some applications. However, in many cases, this will not be the case. In cases where those mappings cannot be made, users can approximate this in terms of basic computing resources. For example, a user running a scientific simulation might want to speak in terms of grid points evaluated per second, but percentage of the CPU obtained over time may be just as effective. For many applications, we hypothesize that the basic resources critical for performance are well-known.

2.2 Translating Value

Given competing users' values for applications using a shared resource, a market-based system then needs a policy for translating those values into specific resource allocations with appropriate performance guarantees. How the policy goes about doing this involves a critical trade-off between human-computer interaction (HCI) design goals and criteria in the design of economic mechanisms.

On one hand, from the HCI perspective, because the expression of value and the policy to translate value is directly exposed to and observable by users (the policy may not actually be known, but the effects of the policy in terms of resource allocations can certainly be observed), this favors use of a policy that is simple, intuitive, and which users can form a crisp mental model of how the system behaves in response to their choices. On the other hand, from a game theory perspective, a common design goal is for policies to be *incentive compatible*, meaning that in deciding how to assign a value to an application the optimum utility-maximizing behavior is to reveal one's true value; thus users need not be concerned with other users' behavior [14]. Different policies in real systems will reflect different trade-offs between these two perspectives.

Another issue associated with the translation of value to resource allocations is the nature of the resource allocations themselves. This relates to the point made earlier regarding what are the entities to which value is being assigned. When value is translated to resource allocations, those allocations should reflect a performance guarantee, hard or statistical, that can be ultimately meaningful to the user after zero or more layers of translation. For example, assigning value to a fraction of a resource that an application obtains over time is well-defined and meaningful. This would be a case with zero layers of translation. As another example, suppose a video application wanted to run 30 frames per second and the maximum achievable frame rate was directly proportional to CPU utilization, then providing a mechanism to control the CPU very precisely (e.g., using a stride scheduler) would be a reasonable way to provide meaningful resource allocations, after one level of translation.

2.3 Enforcing Value

Finally, once value has been translated into resource allocations with performance guarantees, operating system mechanisms are needed to enforce those allocations so that users get the resources they are paying for. That is, we need appropriate quality of service (QoS) mechanisms on the resources to which value is being assigned (and users are being charged!).

Numerous papers have been published on novel quality of service mechanisms for CPU time, networking, physical memory, I/O, spin-locks, etc. Unfortunately, despite a plethora of mechanisms, relatively little work has been done in the area of policies which can make effective use of these mechanisms in real user environments. There are a number of possible reasons for this: no incentives to prevent unrestricted, socially unacceptable use of the mechanisms (i.e., what's to discourage a user from turning the service knob to the maximum setting?), lack of inclusion in popular operating systems, etc.

Perhaps the most likely reason is that on personal computers and workstations, there simply is not a compelling need to have quality of service mechanisms to control resource usage of competing applications. (On the networking side, the absence of QoS is no doubt partly due to difficulty in implementing and deploying end-to-end QoS.) On single node systems, more often than not, typical applications being run execute reasonably well in the presence of competition. On the other hand, on a cluster system, whose intended use is high performance computing, the situation is quite the opposite. Very demanding applications are common and the effects of and frequency in which non-negligible contention occurs for shared resources are significant.

3 Architecture

To address the issues raised in Section 2, in this section, we define an abstract architecture for market-based resource management on clusters. We take specific positions on certain key issues (e.g., the entities to which value is being assigned). These positions are based largely on our goal of defining an architecture which can be used to realize real systems with end-to-end properties that can be measured in real settings with users running a wide range of applications. This goal stands in contrast to much of the previous work on market-based systems which was highly theoretical in nature and focused on fairly narrow slices of the problem (e.g., extensive game theoretic analyses of policies to translate value to allocations). The architecture consists of five layers (Figure 1): *resources*, computational resources such as CPU, physical memory, and disk; *resource managers*, operating system entities which enforce resource allocations; *the economic front end*, the entity which translates user value into resource allocations; *access modules*, the means by which users assign value and run applications on the system; and *users and applications*, the entities which use the system and ultimately drive the design decisions in all other layers.

3.1 Resources

Resources are the computational resources that are controlled by the resource managers and allocated to applications to deliver value to users. Basic time-shared resources such as CPU time, I/O bandwidth, and networking bandwidth and shared-spaced resources such as physical memory and disk space are typical examples. More unconventional examples include spin-locks on SMPs and time-shared access to special-purpose devices, such as a wall-of-monitors.

In the context of the computational economy, we focus primarily on resources that exhibit scarcity and which have significant application-level performance implications. In other words, we focus

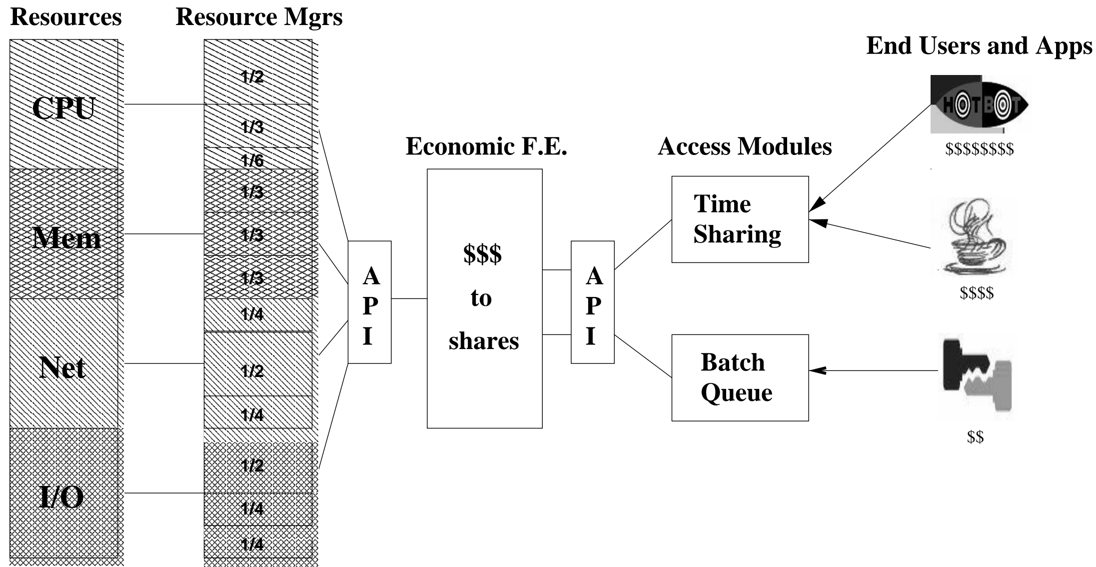

Figure 1: **Abstract end-to-end system architecture.** End users, running a wide range of applications, assign values to their applications and run them on the cluster using the appropriate access module. Access modules run jobs on the cluster and obtain resources for competing jobs using an economic front end. Given a mode of access and a set of competing jobs, an economic front end translates values expressed in a common currency to resource shares. Resource share allocations are enforced using resource managers, one per resource type.

on resources where the demand for the resource can, with non-negligible probability, exceed supply in a way that application performance suffers. These are the resources whose allocation should be controlled and priced through the computational economy.

3.2 Resource Managers

Resource managers are the operating system entities which provide resource allocations with associated performance guarantees. Examples of possible design choices for resource managers include priority schedulers [24, 29, 31], real-time schedulers [21, 28], fair-share schedulers [19, 20], proportional-share schedulers [22, 33, 35, 36], and rate-based schedulers [9, 38]. Which resource manager to choose for each resource in a computational economy depends on several factors including resource allocation precision, flexibility, and overhead. Ultimately, for resources whose allocation is under control of the economy, the resource managers for those resources should be capable of exposing and enforcing resource allocations that are, after zero or more layers of translation, meaningful to the user.

In our architecture, we have chosen to use proportional-share schedulers as the resource managers for all basic computational resources (CPU, memory, network, I/O[2]). Our reasons for doing so are threefold: (i) proportional-share schedulers provide an intuitive model of resource allocation, (ii) there exist efficient algorithms for implementing them, and (iii) they provide flexibility in exposing different entities to which to assign value (e.g., in addition to assigning value simply

[2]We currently have a CPU stride scheduler implemented on Linux 2.2.5 running on Dell Poweredge 2-way SMPs. We next plan to implement a minimum-funding revocation algorithm for proportional-share memory management

based on shares, given a CPU stride scheduler, reasonable user estimates of CPU time needed, and admission control one could potentially build a market-based system based on deadlines).

Proposed and articulated in detail by Waldspurger [33, 35, 36], proportional-share scheduling provides a simple, intuitive model of resource allocation. In this model, resource rights for a shared resource are encapsulated as tickets. A resource is represented by a total of T tickets. An application holding t tickets competing for use of that resource obtains t/T of the resource. For example, with CPU stride scheduling, an application holding t tickets would obtain an accurate t/T allocation of the CPU over time. This model of allocation is simple to reason about and affords a simple mapping from value to resource allocations by mapping credits (credits is the name of the currency used in our system that can traded for resources) per minute to a ticket allocation over time.

3.3 Economic Front End

The economic front end is the entity which translates value into resource allocations in terms of shares (i.e., fractions) of a resource as enforced by the resource managers. What policy the front end implements is largely a function of which access modules we use to access the system. For example, different policies are appropriate if users are accessing the system for interactive use in a time-shared environment as opposed to, say, submitting jobs to a batch queue which runs jobs on a cluster. Vickrey auctions [32], for instance, may be appropriate for a batch queue since there is a delay between the time a job is submitted and the time it is executed in which the auction has time to clear. On the other hand, an auction in an interactive system makes less sense as the time to clear becomes an issue (e.g., does the user have to wait after typing a command for an auction to clear?).

Policies will also depend on the entity or entities to which value is being assigned and associated performance guarantees. Whether the system performs admission control is a key issue here as it determines whether the resource allocations that applications receive can represent absolute performance guarantees or whether they represent relative performance guarantees. With admission control, for example, absolute guarantees could be made to manage a futures market in computational resources [17]. Futures could be sold and enforced using proportional-share schedulers and an admission control policy that prevents overselling of resources. (This combination of proportional-share scheduling and admission control would, in effect, be trying to emulate what real-time schedulers accomplish.) However, in having absolute performance guarantees, this implies using admission control which, depending on the access module may or may not be desired.

The choice of policy also reflects trade-offs between usability concerns and game theoretic design points. From the abstract architecture point of view, the appropriate trade-offs to be made in the front end's policy are still very much open. To date, there have been no reported user studies on user behavior on market-based clusters. (There have been few real implementations at all, certainly none on the scale at which we wish to examine this problem.) Without real user behavior data, it is impossible to know, for example, whether the often criticized assumption of hyperrational human behavior assumed in game theory even matters in practice. If it does not and users tend not to try and exploit the properties of the game, knowing this would provide a lot of freedom in trying to make the system usable, for example. In designing the system, we take the position of giving usability concerns higher priority than making the policy absolutely incentive compatible. Usability will always be an important design point. Ensuring that a policy is incentive compatible may not.

3.4 Access Modules

Access modules provide the means by which users access cluster resources to run applications. Common examples include telnet/rlogin/rsh/ssh for interactive remote execution on a single node, custom remote execution environments [13, 16] or multiple rsh/ssh instances for interactive remote execution of parallel programs on multiple nodes, and batch queue systems for large numbers of sequential and/or parallel jobs.

To support market-based resource management, access modules will need to be augmented so users can assign value to their applications. Assigning value may be as simple as specifying a scalar value that is passed interactively on the command line, or may involve more complex specifications (e.g., specifying a utility function). They close the loop, in that future workload demand and valuation depends on past performance. In keeping with our goal of usability, we have initially chosen to use simple scalar valuations for CPU time. In addition, because interactive, time-shared access to resources is so common, the first access module we have implemented (Section 4.4) provides interactive remote execution for parallel and sequential jobs. To prevent users from bypassing access through the computational economy, we currently disallow general-purpose use of access modules which have yet to modified (e.g., ssh access is allowed only for administrative purposes).

3.5 End Users and Applications

Although not explicitly part of the system's actual implementation, end users and applications are an important part of the end-to-end architecture. It is the modes of usage and resource demands of the users and their applications that ultimately drive many design choices in the system. It is the users' actual usage of the system that will determine whether market-based resource management is effective or not. Keeping the users in mind requires thinking about how they use clusters (e.g., batch mode or interactive), how sophisticated they can be assumed to be (e.g., how much do they understand about how computer systems work), what the resource demands of their applications are (e.g., what resources do they care about), etc. These factors have significant implications for the design of the other layers of the system.

4 Implementation

To investigate how users behave when using market-based systems and how these systems perform in practice, we have implemented a computational economy for CPU time on a cluster of PCs as part of the UCB Millennium Project. The system allows users to run sequential and parallel programs while assigning a value which specifies the maximum credits per minute the user's application is willing to spend purchasing CPU time. Using a simple, intuitive algorithm, each node independently computes and charges for CPU allocations for competing applications and enforces these allocations using stride scheduling, an efficient, deterministic proportional-share scheduling algorithm.

4.1 Resources: UCB Millennium Cluster

The system consists of 32 2-way Dell Poweredge 2300 multiprocessors running a modified version of the Linux 2.2.5 operating system. Each node has two 500 MHz Intel Pentium III processors, 512 MB of memory, 17.4 GB of local storage, and has network connections to Myrinet [3], a switched, 1.28 Gb/s wormhole-routed network, and a switched 100 Mb/s Ethernet. Resources in our system

are the four basic computational resources on each node (CPU time, physical memory, I/O bandwidth, and network bandwidth), each of which is broken up into time or space-shared shares. At present, our system supports economic resource allocation of CPU shares. Memory, I/O, and network bandwidth are allocated using the default operating system policies.

4.2 Resource Managers: CPU Stride Schedulers

To enforce CPU resource allocations based on value, we modified the Linux 2.2.5 CPU scheduler by replacing the standard priority scheduling algorithm with stride scheduling. Using stride scheduling, our system is able to achieve much better precision when controlling CPU allocations for different jobs compared with priority scheduling. Such precision is necessary since allocations are being driven by user-specified values which result in users being charged credits, of which they have a finite supply to get their work done. Building on the basic stride scheduling algorithm, we have also added a few extensions in order to support stride scheduling on SMPs and to deal with multithreaded and multiprocess jobs.

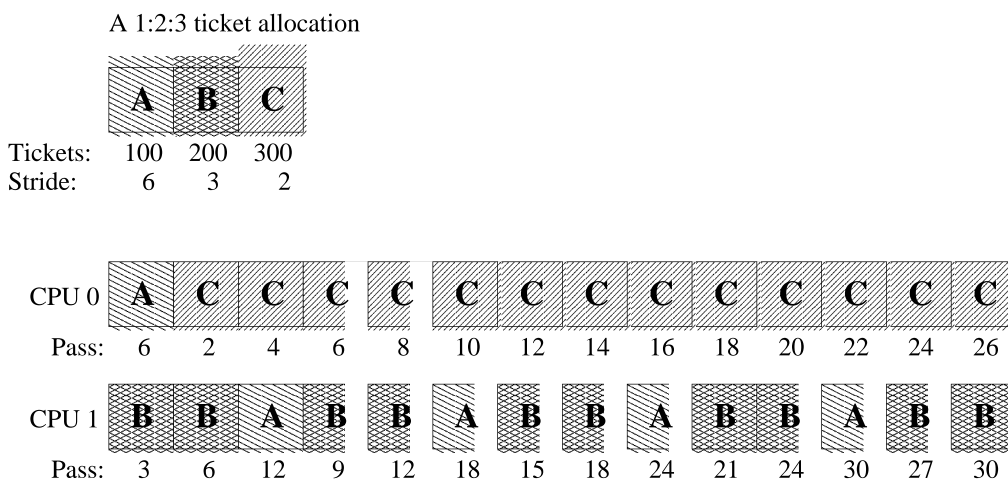

Figure 2: **Stride scheduling algorithm.** In this example, jobs A, B, and C are competing for CPU time on a 2-way SMP. Resource rights to the SMP's 2 CPUs are encapsulated in a pool of 600 tickets. A, B, and C have 100, 200, and 300 tickets respectively, thus they should receive a 1:2:3 allocation of the machine CPUs over time. (Note that with a k-way SMP, having the rights to more than 1/k-th the tickets is really the same as having exactly 1/k unless the job is multithreaded.) A job's stride is the interval of virtual time, measured in CPU quantums, between that job's being selected to run. (It is inversely proportional to the ticket allocation.) A job's pass is the virtual time when the job will be scheduled. It is incremented by the job's stride each time the job is scheduled. For each CPU quantum on each processor, the stride scheduler chooses the job with the lowest pass that is not already scheduled on another processor, runs it for one scheduling quantum, and increments its pass by its stride.

In stride scheduling, resource rights to a shared resource are represented by a pool of T tickets. Let n be the number of jobs competing for the resource. Associated with each job i, we have t_i (tickets), the number of tickets assigned to job i; s_i (stride), the interval of virtual time (measured in units of scheduling quantums), inversely proportional to t_i, job i has to wait before being scheduled to use the resource; and p_i (pass), the virtual time at which job i will next be scheduled. ($\sum_{i=1}^{n} t_i =$

T.) Jobs with more tickets have shorter strides and hence are scheduled more frequently and obtain a greater fraction of the resource. The algorithm itself is simple: the scheduler picks the job with lowest pass, runs it for a scheduling quantum, increments its pass by its stride, and iterates (see Figure 2). It is deterministic and within resolution of the scheduling quantum schedules jobs in exact proportions based on their ticket allocations.

We extended the basic stride scheduling algorithm to support SMPs and to contain and control the resource usage within a collection of related entities (e.g., all the threads in a process, a collection of processes working together). The former was necessary since our basic processing nodes have multiple processors and the Linux kernel is organized around a centralized run queue. The latter was necessary to prevent a user from running a process, then having that process create replicas of itself by forking or cloning, each of which would then be scheduled with the same value as the original process. Global pass, ticket, and stride values represent aggregate information for all competing jobs and are necessary to support a system where jobs can join and leave the system dynamically. (See [36] for more details.) Making the stride scheduling algorithm SMP-aware simply required a small change in the way the global pass is computed. In particular, when k processes are running concurrently on k CPUs, the global pass must be incremented k times as fast.

To support multithreaded and multiprocess jobs, we isolate all related entities using their own currencies. Each job in the system is scheduled on the basis of the number of base tickets it has. To deal with a multithreaded or multiprocess job i with base ticket allocation t_i, we simply ensure that the aggregate base ticket assignment for all entities forked or cloned from a root process, including the root itself is t_i. We do this by maintaining a linked list of currencies, each of which has an associated linked list of processes/threads that use that currency. The default policy for ticket allocation within a multithreaded or multiprocess job is to perform fair share across all entities. For applications that desire more control, a system call interface is provided that allows applications to manage tickets in their own applications-specific currency.

4.3 Economic Front End: Initial Economic Policy

To translate values to shares of a CPU (i.e., tickets), we have elected to start with a simple, intuitive policy based on opportunity-cost charging[3] that is easy for users to reason about. In this scheme, n jobs are competing for some shared resource r, in this case CPUs on a node. Job i has a stated value b_i, which may or may not be the same as true value v_i (e.g., suppose a user is trying to undervalue the value of an application to save credits or perhaps the user does not realize how important a particular run is). The system allocates shares and charges as follows. If $n = 1$, there is only one user demanding the resource. Thus, that user is imposing no burden on the system and not denying anyone else an opportunity since there is no competition for r. In this case, that user is charged nothing and is given all of r. If $n > 1$, multiple jobs are competing for r and jobs receive shares as they would if the b_i's were tickets. That is, job i with stated value b_i obtains $b_i / \sum_{j=1}^{n} b_j$ of r over time. When $n > 1$, job i is charged b_i credits per minute. (If multiple jobs are competing, each with $b_i = 0$, each job receives fair share and is not charged.) The computing of shares and charging is computed dynamically as jobs join and leave the system.

A key issue addressed by this policy is the distinction between selling and sharing. Our policy charges users based on the burden they place on other users sharing the system. Under no con-

[3] Opportunity-cost charging occurs when charging is done based on the opportunity to use a resource. A user who opts to use a resource is charged based on the value of the opportunity to use that same resource that is being denied to other competing users.

tention, this means users use the system for free. Conversely, under high contention, users pay more to use the system. Contrast this approach to that of a profit-center, charging to maximize seller revenue. Selling to maximum seller revenue produces different user incentives, incentives which may result in suboptimal sharing of resources. Under monopoly pricing, for example, it is common for a seller to maximize revenue while underutilizing the underlying resources being sold. For a shared computer system, pricing schemes such as this make little sense when considering our original motivation for computational economies.

Addressing sharing directly, this policy has a number of other desirable properties as well, including simplicity, intuitiveness, and low computational overhead. From a usability perspective, it makes it very easy for users to reason about what the system's response will be to assigning some value to a computation. This property is very important in order to reduce the amount of noise in our experiments. If users are unable to understand what it means to assign value given a particular means of doing so, trying to meaningfully interpret the results of users using such a mechanism will be difficult if not impossible. By making the policy and interface extremely simple (scalars), we increase the chances that most users will have a clear picture of what it means to assign a value to a computation and what the system will do in the face of competition when assigning resources. Another nice property of this policy is that it is computationally simple to implement. The overhead required to compute allocations and charging are straightforward and are even simpler than those needed to implement stride scheduling. Furthermore, because we do not anticipate users using the cluster to run large numbers of very short jobs (e.g., less than a second), the dynamic updating of allocations, which require system calls, should incur fairly minimal overhead.

4.4 Access Modules: REXEC Remote Execution Environment

To allow users to interactively run jobs on the cluster through the economy, we have implemented REXEC, a secure, decentralized remote execution environment for parallel and sequential jobs. Compared to previous remote execution systems (e.g., GLUnix [13], Score-D [16], LSF [39]), REXEC provides a unique set of features including decoupling of node discovery and selection, emulation of local execution on multiple nodes, decentralized control, dynamic discovery and configuration, well-defined failure and cleanup models, and authentication and encryption. In Section 4.4.1, we discuss each of features in more detail. In Section 4.4.2, we briefly describe the REXEC implementation.

4.4.1 Features

Decoupling discovery and selection means that the process of discovering which nodes are available in the system is separated from the process of selecting which nodes the system should run a user's application on. By decoupling these two processes, users are given the flexibility to choose which selection policy they want to use based on their personal preferences as opposed to having the system implement a global policy that everyone is forced to use. For a computational economy, this property is important because we expect users will require multiple selection policies depending on their needs.

Emulation of local execution on multiple nodes means that a program can be run on multiple nodes with propagation of the user's local environment, signals, stdin, stdout, and stderr. This provides users with a familiar execution environment and also allows them to apply standard job control mechanisms (e.g., C-c, C-z) in controlling their jobs. Clearly, not all local behavior will make sense when extended to multiple remote instances of an application (e.g., running top on four nodes

from an xterm). REXEC provides sufficient emulation in the common cases. It does not, however, attempt to address these types of mismatches.

For scalability and availability reasons, REXEC is decentralized. A position we take with REXEC is that any node capable of running user programs should always be available to do so. There should be no false dependencies between artifacts of the remote execution environment and nodes which are perfectly healthy to do useful work. Decentralized control is achieved in two ways: (i) by having each client directly manage the remote execution of a job on multiple nodes and (ii) through replication of the node discovery and selection service. Besides increasing availability and reducing performance bottlenecks, decentralized control also works well with the economy, which is also decentralized (i.e., each entity acts based on local information).

REXEC does not depend on static configuration files or a fixed set of cluster nodes. Instead, all entities in REXEC dynamically discover each other through use of well-known IP multicast channel. In a large cluster of machines, adding or removing a node from the cluster is a fairly common event. Thus, it should not be the case that doing this requires restarting the system or manually editing configuration files. REXEC avoids this completely. Cluster nodes, the discovery and selection servers, and even the account service (for managing user credits in the economy) can all be dynamically discovered.

REXEC detects failures and provides well-defined failure and cleanup models. Remote processes, machines, networks, and so forth will fail from time to time. REXEC detects such failures and performs appropriate actions. A significant shortcoming of many remote execution systems is the lack of a precise failure model. For example, when a user types C-c while running a remote parallel program, what that actually means in terms of cleaning up remote resources is usually not well-defined. REXEC currently provides a single failure and cleanup model which states that if anything in the remote execution fails at any time, the entire program is aborted and all remote resources are freed. For example, freeing resources on a remote node involves termination of an arbitrary tree of child processes (and child processes which might have broken off from the tree and been inherited by the init process) and freeing REXEC resources on that node dedicated to that job.

Lastly, REXEC implements client-side authentication and encrypts all relevant data exchanged between clients and servers using the SSLeay implementation of the Secure Sockets Layer (SSL) Protocol [12]. To use the system, a user presents a certificate, signed by a trusted certificate authority, that establishes the user's identity and uses public key cryptography to prove himself. Using public-key cryptography, this scheme provides significant improvements against attacks compared to traditional password-based schemes. In addition to authentication, REXEC also performs encryption of all relevant data (stdin, stdout, stderr, signals flowing to and from remote programs). This prevents potentially eavesdroppers from snooping on the network and observing input and output to programs users are running on the system.

4.4.2 Implementation

The REXEC system consists of three main programs: *rexecd*, a daemon which runs on each cluster node; *rexec*, the client program that users run to execute jobs using REXEC; and *vexecd*, a replicated daemon which provides node discovery and selection services (Figure 3). Logically, it consists of two layers: REXEC, the physical layer which provides low-level mechanisms for remote execution, failure detection and cleanup, and authentication and encryption on a set of named nodes; and VEXEC, the virtual layer which provides a set of node names based on some selection criteria (e.g.,

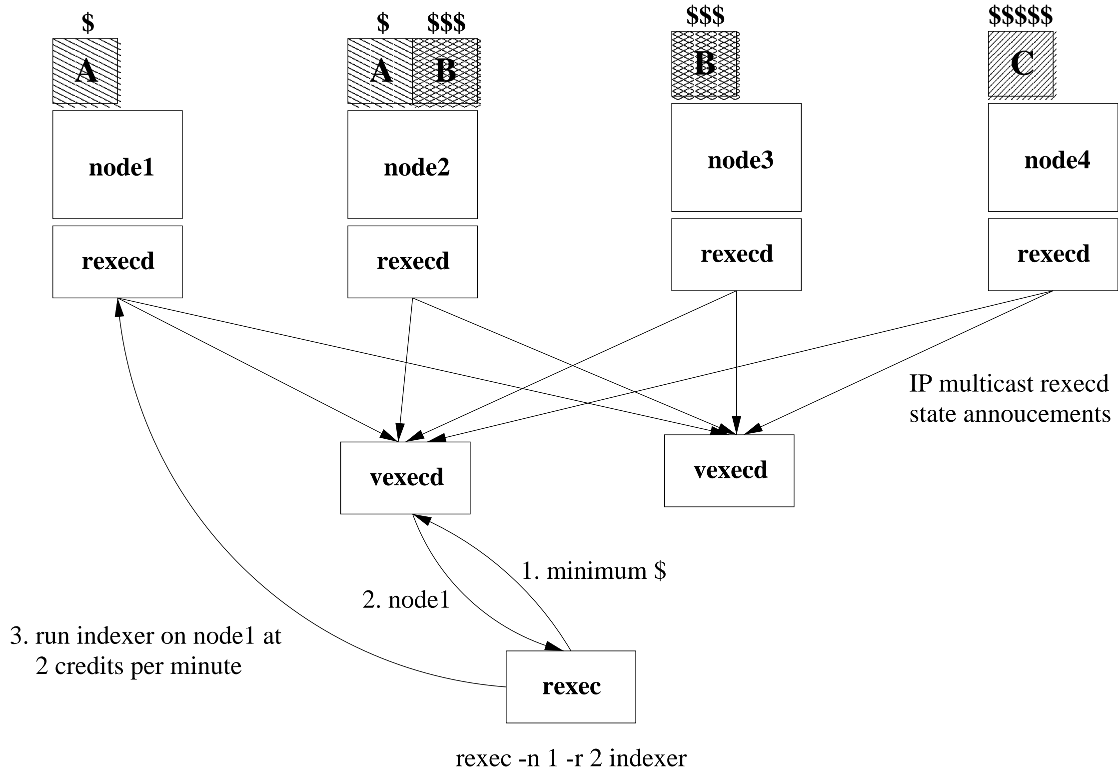

Figure 3: **REXEC remote execution environment.** The REXEC system consists of three main programs: *rexecd*, a daemon which runs on each cluster node; *rexec*, the client program that users run to execute jobs using REXEC; and *vexecd*, a replicated daemon which provides node discovery and selection services (Figure 3). Users run parallel and sequential applications in REXEC using the rexec client, which performs two functions: (i) selection of which nodes to run on based on user preferences and (ii) remote execution of the user's application on those nodes through direct SSL/TCP connections to the node rexecd daemons. In this example, there are four nodes in the system: node1, node2, node3, and node4 and two instances of vexecd, each of which implements a lowest CPU price policy. A user wishes to run a program called indexer at rate two credits per minute and wishes to do this on the machine with the lowest CPU price. Contacting a vexecd daemon, rexec obtains the name of the machine with lowest CPU price (node1). rexec then establishes an SSL/TCP connection directly to node1 to run indexer at a rate of two credits per minute. indexer will then be started on node1 and compete for resources with job A.

cheapest CPUs). Users run parallel and sequential applications in REXEC using the rexec client, which performs two functions: (i) selection of which nodes to run on based on user preferences (e.g., using a user-specified vexecd which implements the user's preferred policy) and (ii) remote execution of the user's application on those nodes through direct SSL/TCP connections to the node rexecd daemons. The REXEC system is implemented mainly at user-level with a few small kernel changes for properly freeing resources for a tree of processes.

To extend REXEC to operate within the framework of the computational economy, we made four changes to the system. First, a vexecd daemon was written that implements a minimum price policy on all the nodes it knows about. Each rexecd in the cluster sends state announcements on a well-known IP multicast channel, periodically and whenever a state change occurs (similar to V [5]). vexecd daemons listen on this channel and maintain a list of nodes they have heard from over a specific window of time. A selection policy is simply an ordering on this list. Second, the rexec client was modified to allow specification of CPU valuation for the user's application. Using a new command line switch (-r maxrate), the user specifies the maximum credits per minute the application can spend purchasing CPU time. Third, rexecd was modified to use an economic front end (a collection of functions that implement the CPU market) to translate user-specified rates into CPU allocations under the stride scheduler whenever the state of the system changes (i.e., whenever a job starts/exits on a node). Finally, with applications competing for resources and being charged for usage, an account service was needed to manage credit usage for each of the users. This service provides simple credit/debit management of user credit accounts. rexecd daemons obtain the IP address and port of the account service at runtime using IP multicast then establish secure SSL/TCP connections to perform transactions against its database.

4.5 End Users and Applications: UCB Millennium Project

As part of the UCB Millennium Project, our system has a user community that spans 17 academic departments on the UC Berkeley campus. These users work on a wide range of problems in areas such as multimedia, civil engineering, computational astrophysics, digital libraries, large-scale Internet systems, physics, and computer-aided design. Computational workloads generated by these users typically take the form of large numbers of sequential jobs (e.g., in exploring a parameter space) or large parallel jobs written using MPI. In addition to Millennium users, there is also a significant opportunity to obtain participation in user studies through a number of the undergraduate and graduate courses in the Berkeley computer science department. Taken together, the Millennium Project and computer science users will form a large, diverse user community. This community will be used to drive the empirical evaluation of our prototype.

5 Related Work

Ferguson's microeconomic load balancing algorithm is one of the earliest examples of applying market-based ideas to cluster resource management [11]. In his algorithm, multiple sequential jobs compete for dedicated slices of CPU time on a collection of heterogenous (in speed) machines interconnected by point-to-point links. At the end of each CPU slice, each machine holds an auction to determine which, of a set of competing jobs, should be granted exclusive use of its CPU for a user-specified slice of time. The winner of the auction is the job with the highest bid amount per unit of time. This winning rate, also referred to as the current machine price, is posted to both the machine's local Bulletin Board of prices and the Bulletin Boards of a local neighborhood of ma-

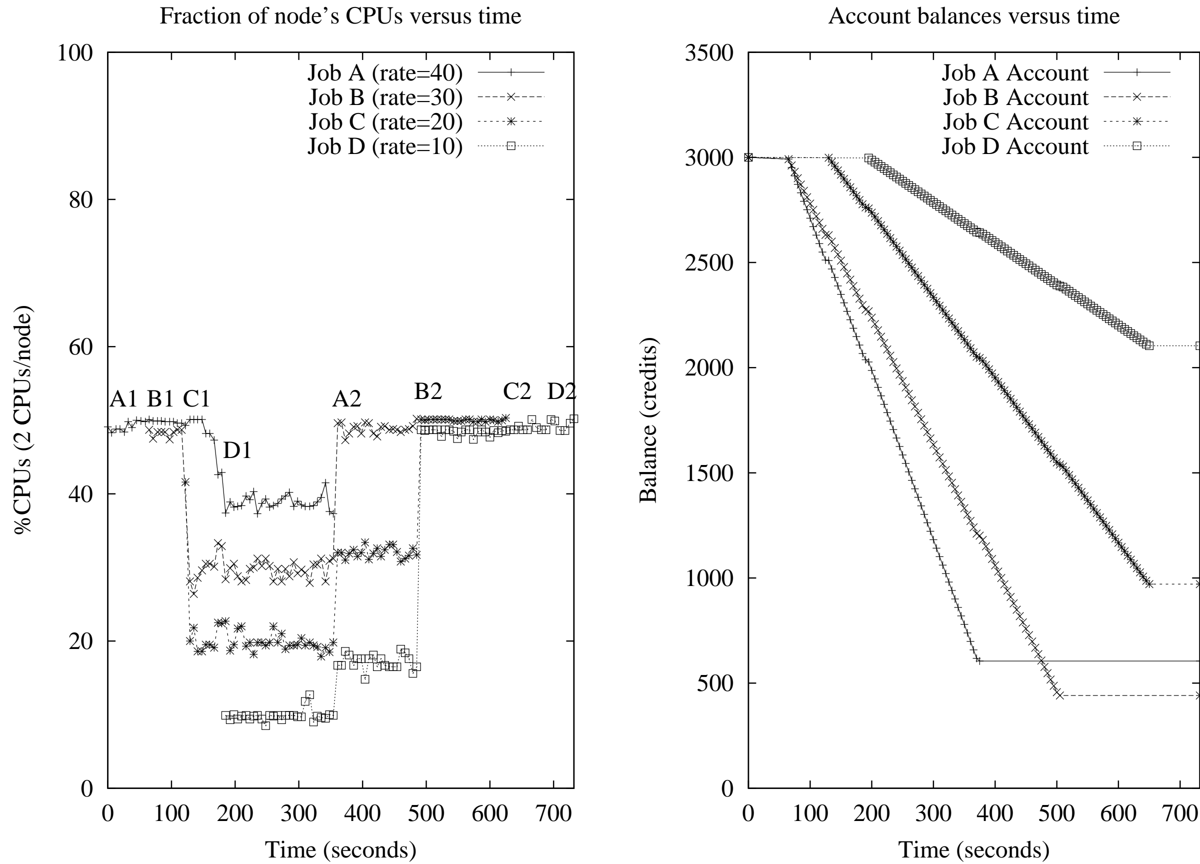

Figure 4: **Example of measured CPU allocation and charging on a single node.** In this scenario, jobs A, B, C, and D are willing to pay rates of 40, 30, 20, and 10 respectively to compete for CPU resources on a single node. Each node has two CPUs. Job arrival times are labeled A1, B1, C1, and D1. Job departure departure times are labeled A2, B2, C2, and D2. At any instant in time (e.g., interval C1-B2), we see that the relative CPU allocation of any two jobs is given by the ratio of the rates they are willing to pay. (Note that since all the jobs are single threaded, when two jobs compete, each receives 50% of the node's CPUs, i.e., one CPU.) When multiple jobs compete, each job is charged based on the rate it is willing to pay and is given an allocation in proportion to its rate. When a single job demands the resources, it is given all of the resources and is charged nothing (e.g, intervals A1-B1, C2-D2).

chines. In Ferguson's model, jobs may be submitted on any machine, migrate to different machines for execution, but must always complete on the machine from which they originated. Where a job gets executed is a function of which machine the job was submitted on, Bulletin Board prices in the neighborhood of that machine, and user preferences (e.g., lowest cost, fastest execution).

Extending Ferguson's work to parallel programs, Spawn is a market-based system for scheduling concurrent tree-based applications on a collection of heterogeneous workstations [34]. In Spawn, each application has a user-specified funding rate, which specifies the aggregate funding rate for the entire application, and is organized as a tree of *worker modules*, the computational tasks of the concurrent application, and *manager modules*, entities which control worker creation and relative funding of workers under their control. Using their funding, applications compete for resources with each other by bidding on dedicated slices of CPU time on multiple workstations, each of which periodically runs a sealed-bid second-price auction. Upon winning a workstation's auction, an application is granted exclusive use of that workstation's CPU until its slice expires. Upon using its slice, the application can continue by requesting an extension and continuing to pay market price (i.e., the price to win the next auction) or is terminated.

System	Resources	Resource Mgrs	Economic F.E.	Access Modules	Users/Apps
Ferguson	CPU	Admission control	Sealed bid and Dutch auctions	None	None
Spawn	CPU	Admission control	Vickrey auction	Application manager	None
Popcorn	CPU	Admission control	Vickrey, double, and Clearinghouse auctions	Java API	None
Mariposa	CPU, I/O, disk	N/A	Contract Net; pricing based on load, expected resource usage, history	None	None
SC centers	CPU, memory, disk	Priority scheduler[4]	nice, priority weighted usage charges	Batch queue with priority, ssh	Numerous

Table 1: **Related work summary.** This table summarizes related work in the framework of the abstract architecture presented in Section 3. Looking at the data, we find that most of the work so far has focused mainly on the economic front end layer. Very little attention has been paid to the end-to-end problem, how real users and applications use market-based systems in practice, and the implications real usage has on other layers of the system. (Supercomputing centers, though listed here, are a bit of a stretch in being categorized as market-based.) Lack of real users and applications in four of the systems and either lack of or highly specialized access modules clearly shows this (e.g., Popcorn requires each program to be written with the economy in mind using a special API.). As further evidence, notice that Ferguson's algorithm, Spawn, and Popcorn all assume dedicated use of the system. However, in many systems in use today, multiple users sharing resources is an important, common mode of usage.

[4]This is the case for clusters. Some supercomputing centers have specialized vector machines that support more

Popcorn is a wide area infrastructure for market-based CPU scheduling of distributed computations written in Java [26]. In this system, users write distributed Java applications by decomposing their application into a collection of coarse-grain tasks called computelets. Each computelet requires some number of Java Operations (JOPs) to complete. Buyers (users) assign desired prices to computelet or JOP-based contracts based on the computation's value and enter a well-known, centralized market to seek sellers. Sellers are the users selling CPU time on their machines to perform JOPs on behalf of buyers. In the market, a matching of a buy and sell can occur in three different ways, each constituting a different internal market. These ways include a repeated Vickrey auction (sealed-bid second-price auction), a double auction, or a Clearinghouse double auction. It is unclear from the paper what the duration of a contract is. Like Ferguson's algorithm, it is possible that Popcorn also uses user-specified contract lengths and performs all economic policies using rates.

Mariposa is a distributed database and storage system that uses market-based mechanisms for resource management to execute queries [30]. In this system, a client submits a query to a broker with a time-dependent budget $B(t)$, which specifies the value delivered to the client as a function of the query's completion time. A broker, upon receiving a query, computes a parallel, pipelined execution plan to efficiently execute the query as a collection of subqueries. Following that, it solicits bids from multiple servers on contracts to execute these subqueries and checks whether it can execute the client's query within the specified budget. If successful, the query plan is executed. Servers provide the processing and storage capabilities to perform client query processing to execute subqueries. Like brokers, they also perform selfish, local optimizations, in particular by pricing contacts based on the resources required to execute it, long-term revenue collected per storage fragment, and the current load.

Supercomputing centers often use pricing algorithms to charge for resource usage and to give different levels of CPU priority to batch queue jobs submitted to high-end machines, such as vector supercomputers and MPPs. For example, on machines managed by the National Partnership for Advanced Computing Infrastructure (NPACI) [25], each job is charged based on its resource usage, measured in service units (SU), and the queue rate of the queue the job was submitted to. Each SU is roughly equivalent to a CPU hour on the machine the job runs on. The queue rate is a priority that the user assigns to a job which determines how quickly the user's job is run. The idea here is that users assign higher priority to jobs which have higher personal value and that these priorities result in jobs running with greater priority in whatever native CPU scheduling mechanism the underlying machine supports. One nice property of supercomputing center charging is that since charges change very slowly (e.g., different fixed pricing schemes during the day and during the night), users can potentially get a better handle on how to manage their funds since prices are predictable. On the other hand, because prices are not responsive to immediate demand, it is clear that prices during periods of high demand will be underpriced (so demand exceeds supply) and that during periods of low demand, users will be overcharged. Finally, how the prices and constants used in pricing and computing SUs is not obvious and does not appear to change over time (e.g., to reflect long-term averages in observed supply and demand).

6 Conclusion

Enabling technologies in high speed communication and global process scheduling have pushed clusters of computers into the mainstream as general-purpose high-performance computing systems.

---

sophisticated resource allocation mechanisms.

More generality, however, implies more sharing and this raises new issues in the area of cluster resource management. To address these issues, we proposed a decentralized, market-based approach to resource management based on the idea of proportional resource sharing of basic computing resources. Here, a cluster of computers is organized as a computational economy which optimizes for user value. Cluster nodes act as independent sellers of computing resources and user applications act as buyers who purchase resources based on the personal value delivered to users. We described the three fundamental functional requirements for a market based system: expressing value, translating value, and enforcing value. We then proposed an abstract architecture for market-based cluster resource management based on proportional resource sharing. Using this architecture, we have implemented a 32-node (64 processors) prototype system that provides a market for time-shared CPU usage for sequential and parallel programs. To begin evaluating our ideas, we are currently in the process of studying how users respond to the system by collecting data on real day-to-day usage of the cluster.

References

[1] ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A., AND THE NOW TEAM. A case for now (networks of workstations). *IEEE Micro* (Feb. 1995).

[2] ARPACI-DUSSEAU, A. C., CULLER, D. E., AND MAINWARING, A. Scheduling with implicit information in distributed systems. In *Proceedings of the 1998 ACM SIGMETRICS Conference* (1998).

[3] BODEN, N. J., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., SEIZOVIC, J. N., AND SU, W.-K. Myrinet—a gigabit-per-second local-area network. *IEEE Micro 15*, 1 (Feb. 1995), 29–38.

[4] BUONADONNA, P., GEWEKE, A., AND CULLER, D. E. An implementation and analysis of the virtual interface architecture. In *Proceedings of Supercomputing '98* (1998).

[5] CHERITON, D. The v distributed system. *Communications of the ACM 31*, 3 (March 1988).

[6] CHIEN, A., PAKIN, S., LAURIA, M., BUCHANON, M., HANE, K., AND GIANNINI, L. High performance virtual machines (hpvm): Clusters with supercomputing apis and performance. In *Proceedings of 8thSIAM Conference on Parallel Processing for Scientific Computing (PP97)* (1997).

[7] CHUN, B. N., MAINWARING, A. M., AND CULLER, D. E. Virtual network transport protocols for myrinet. In *Proceedings of the 5th Hot Interconnects Conference* (Aug. 1997).

[8] CULLER, D., ARPACI-DUSSEAU, A., ARPACI-DUSSEAU, R., CHUN, B., LUMETTA, S., MAINWARING, A., MARTIN, R., YOSHIKAWA, C., AND WONG, F. Parallel computing on the berkeley now. In *Proceedings of of 9th Joint Symposium on Parallel Processing* (Kobe, Japan, 1997).

[9] DEMERS, A., KESHAV, S., AND SHENKER, S. Anaylsis and simulation of a fair queueing algorithm. In *Proceedings of 35th IEEE Computer Society International Conference (COMPCON)* (March 1990), pp. 380–386.

[10] DUKE, D. W., GREEN, T. P., AND PASKO, J. L. Research toward a heterogeneous networked computing cluster: The distributed queuing system version 3.0 (http://www.scri.fsu.edu/ pasko/dqs/dqs.html). 1996.

[11] FERGUSON, D., YEMIMI, Y., AND NIKOLAOU, C. Microeconomic algorithms for load balancing in distributed computer systems. In *International Conference on Distributed Computer Systems* (1988).

[12] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The ssl protocol version 3.0 (internet-draft). 1996.

[13] GHORMLEY, D. P., PETROU, D., RODRIGUES, S. H., VAHDAT, A. M., AND ANDERSON, T. E. Glunix: a global layer unix for a network of workstations. *Software—Practice and Experience* (Apr. 1998).

[14] GIBBONS, R. *Game Theory for Applied Economists.* Princeton University Press, 1992.

[15] GMBH, G. S. Codine 4.2: Technical description (http://www.genias.de/products/codine/tech_desc.html). 1998.

[16] HORI, A., TEZUKA, H., , AND ISHIKAWA, Y. An implementation of parallel operating system for clustered commodity computers. In *Proceedings of Cluster Computing Conference '97* (March 1997).

[17] HULL, J. C. *Futures and Options Markets.* Prentice Hall, 1998.

[18] IBM. Ibm loadleveler: General information. September 1993.

[19] KAY, J., AND LAUDER, P. A fair share scheduler. *Communications of the ACM 31*, 1 (January 1988), 44–55.

[20] LARMOUTH, J. Scheduling for a share of the machine. *Software—Practice and Experience 5*, 1 (January 1975), 29–49.

[21] LIU, C. L., AND LAYLAND, L. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1 (1973), 45–61.

[22] MAHESHWARI, U. Charge-based proportional scheduling. Tech. Rep. MIT/LCS/TM-529, Massachusetts Institute of Technology, April 1995.

[23] MAINWARING, A. M., AND CULLER, D. E. Design challenges of virtual networks: Fast, general-purpose communication. In *Proceedings of 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)* (1999).

[24] MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1996.

[25] NPACI. National partnership for advanced computing infrastructure web page (http://www.npaci.edu).

[26] REGEV, O., AND NISAN, N. The popcorn market – an online market for computational resources. In *Proceedings of the 1st International Conference on Information and Computation Economies* (1998).

[27] RIDGE, D., BECKER, D., MERKEY, P., , AND STERLING, T. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings of IEEE Aerospace* (1997).

[28] SHA, L., KLEIN, M. H., AND GOODENOUGH, J. B. *Rate Monotonic Analysis for Real-time Systems. In Andre M. van Tilborg and Gary M. Koob, editors, Foundations of Real-time Computing: Scheduling and Resource Management.* Kluwer Academic Publishers, 1991.

[29] SILBERSCHATZ, A., AND GALVIN, P. B. *Operating System Concepts.* Addison Wesley, 1994.

[30] STONEBRAKER, M., DEVINE, R., KORNACKER, M., LITWIN, W., PFEFFER, A., SAH, A., AND STAELIN, C. An economic paradigm for query processing and data migration in mariposa. In *3rd International Conference on Parallel and Distributed Information Systems* (September 1994), pp. 58–67.

[31] TANENBAUM, A. S. *Modern Operating Systems.* Prentice Hall, 1992.

[32] VICKREY, W. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance 16* (1961), 8–37.

[33] WALDSPURGER, C. A. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Massachusetts Institute of Technology, 1995.

[34] WALDSPURGER, C. A., HOGG, T., HUBERMAN, B. A., KEPHART, J. O., AND STORNETTA, S. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering 18*, 2 (February 1992), 103–177.

[35] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation* (1994), USENIX Association, pp. 1–11.

[36] WALDSPURGER, C. A., AND WEIHL, W. E. Stride scheduling: Deterministic proportional-share resource management. Tech. Rep. MIT/LCS/TM-528, Massachusetts Institute of Technology, 1995.

[37] WELSH, M., BASU, A., AND VON EICKEN, T. Incorporating memory management into user-level network interfaces. In *Proceedings of the 5th Hot Interconnects Conference* (Aug. 1997).

[38] ZHANG, L. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems 90*, 2 (May 1991), 101–124.

[39] ZHOU, S., WANG, J., ZHENG, X., AND DELISLE, P. Utopia: A load sharing facility for large, heterogenous distributed computer systems. *Software—Practice and Experience* (1992).