A System for Batch-Mode Economic Scheduling of a Cluster of Workstations

by

Andrew R. Geweke

B.S. (Michigan State University) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David E. Culler, Chair
Professor Joseph M. Hellerstein

Fall 2000

The dissertation of Andrew R. Geweke is approved:

Chair Date

Date

University of California at Berkeley

Fall 2000

A System for Batch-Mode Economic Scheduling of a Cluster of Workstations

By

Andrew R. Geweke

Abstract

A System for Batch-Mode Economic Scheduling of a Cluster of Workstations

by

Andrew R. Geweke

Master of Science in Computer Science

University of California at Berkeley

Professor David E. Culler, Chair

Clusters of commodity workstations are becoming an increasingly dominant solution for achieving very high computational performance. Their growing popularity and the increasing breadth of applications running on these clusters creates a new set of stresses on clusters' resources. Existing resource-management algorithms, designed for stand-alone computers or monolithic supercomputers, may not be suitable for managing the resources of a large, distributed cluster.

We present our experience managing such a large cluster in an academic environment, and analyze the stresses placed upon it. We then introduce a *computational economy* for managing the cluster's resources, in which users trade a valuable resource ("money") for processing time upon the cluster, with the price set dynamically using a well-known economic model, the Vickrey auction. We analyze users' behavior under this computational economy and compare it with prior behavior. We also apply past data traces to a simulator to deduce what the effect of such a system would have been at prior times. Finally, we contrast and analyze the data obtained under all three regimens and draw conclusions about the effectiveness of a computational economy.

I count life just a stuff
To try the soul's strength on.

Robert Browning

Contents

1 Introduction 1

2 Background, Motivation, and Related Work 3
2.1 Background 3
2.1.1 Key Metrics for Scheduling Decisions 3
2.1.2 Schedulers for standalone systems 4
2.1.3 Schedulers for Clusters and Distributed Systems 6
2.1.4 Economic Systems 7
2.1.5 The Synthesis of Economic Systems and High-Performance Computing 9
2.2 Motivation 9
2.2.1 The Pre-existing System 9
2.2.2 A Unique Opportunity 12
2.2.3 A Large-Scale, Real-World System 13
2.3 Related Work 13

3 The Existing System: GLUnix on the NOW 16
3.1 GLUnix Goals and Design Point 16
3.1.1 Time-shared, immediate-execution 16
3.1.2 Issues with Time-Sharing and Implicit Coscheduling 16
3.1.3 Reservations and Privileges 17
3.2 Resources 17
3.2.1 Computers 18
3.2.2 Network 18
3.2.3 Infrastructure 18
3.3 User Behavior and Collected Data 18
3.3.1 General behavior 19
3.3.2 User Conflict and Conflict Resolution 20
3.4 The Coming Millennium 20
3.4.1 Cluster Resources 21
3.4.2 New Challenges and Opportunities 21

4 Choosing an Economic Model 23
4.1 Requirements 23
4.1.1 Low user interaction 23
4.1.2 Transparent 23
4.2 Supply and Demand 24
4.2.1 Aggregate demand of the system 25
4.2.2 Variance of demand over time 26
4.2.3 Reaction of the system to changing demand 26
4.3 Basic Auction Theory 27
4.3.1 Why auctions? 27
4.3.2 Types of Auctions 29
4.3.3 Auctions as used in the real world 30
4.4 Pure Economic Theory vs. Pragmatism 31

4.4.1 Pure Auction Theory 31
4.4.2 A Pragmatic Approach 31
4.5 Vickrey Auctions 33
4.5.1 Previous Results 33
4.5.2 Motivation 33
4.5.3 Tradeoffs 34

5 Implementing an Economy with PBS 35
5.1 Background 35
5.1.1 About PBS 35
5.1.2 Structure 36
5.1.2.1 User Perspective 36
5.1.2.2 Developer's Perspective 38
5.1.2.3 System Components 40
5.1.2.3.1 Server 41
5.1.2.3.2 Machine-Oriented Miniserver (MOM) 41
5.1.2.3.3 Scheduler 41
5.1.2.4 Parallel Execution 43
5.2 The Economic Scheduler 44
5.2.1 Overview 44
5.2.1.1 JNI 44
5.2.1.2 JDBC 45
5.2.1.3 Performance and Reliability Concerns 45
5.2.2 Structure 46
5.2.2.1 Wrapper process 47
5.2.2.2 Economic Scheduler Classes 47
5.2.2.3 JNI 47
5.2.2.4 JDBC 48
5.2.3 Scheduling Algorithm 48
5.2.3.1 Event Queuing 49
5.2.3.2 Obtaining Status Information 50
5.2.3.3 State Strategy 50
5.2.3.4 Vickrey Auction 51
5.2.3.5 All Done 52
5.2.4 Challenges 52
5.2.4.1 Communication with GLUnix (signaling issues) 52
5.2.4.2 Suspend/resume 52
5.2.4.2.1 Runtime Systems 53
5.2.4.2.2 Signalling via GLUnix 53
5.2.4.2.3 Benchmarking 54
5.2.4.3 GLUnix Reliability 58

6 Simulating an Economy with *econsim* 59
6.1 Motivation 59
6.2 System Design 60
6.2.1 Block Structure 60
6.2.2 Cycle 62
6.2.3 Performance 64

6.2.4 Plug-in models ... 65
6.2.4.1 Simulator ... 65
6.2.4.2 User ... 66
6.2.4.2.1 Current User modules ... 67
6.2.4.3 Job ... 68
6.2.4.4 Status ... 68
6.2.5 Data Sources ... 69

7 Results/Analysis ... 71
7.1 Real Systems ... 71
7.1.1 GLUnix ... 71
7.1.1.1 Workload ... 72
7.1.1.2 Scheduling Behavior ... 75
7.1.1.3 Analysis ... 77
7.1.1.4 Anecdotal Evidence ... 78
7.1.2 PBS Economic Queue ... 81
7.1.2.1 Data ... 82
7.1.2.2 Analysis ... 85
7.2 Simulator ... 87
7.2.1 Data Set ... 87
7.2.1.1 Jobs ... 87
7.2.1.2 Users ... 89
7.2.1.3 Graphs ... 90
7.2.1.4 User Modules ... 94
7.2.1.5 User Modules Studied ... 96
7.2.1.6 Scheduler Modules ... 96
7.2.2 Results and Graphs ... 97
7.2.2.1 Definitions ... 97
7.2.2.2 Graphs ... 98
7.2.2.3 Values ... 101
7.2.2.4 User-Invariant Graphs ... 102
7.2.2.5 Analyzing Job Delay and Job Dilation vs. Bid ... 107
7.2.2.6 Constant-Total-Bid User ... 111
7.2.2.7 Random-Bid User ... 115
7.2.2.8 Proportional-Bid User ... 119
7.2.2.9 Center of Mass ... 123

8 Conclusions ... 125
8.1 Make the user resource-aware ... 125
8.2 FIFO can cause very large latencies ... 126
8.3 Economic queuing gives control and predictability ... 126
8.4 Bidding patterns matter ... 127
8.5 Bids' currency matters ... 128
8.6 Traditional metrics don't capture user desires ... 128

9 Issues and Future Work ... 130
9.1 Users understanding the system ... 130
9.2 Combination with Time-Shared System ... 131

9.3 Production-system issues 131
9.4 Intercluster issues 132
9.5 Technical issues 133

10 Bibliography 134

List of Figures

Figure 2.2-A. Mean number of running jobs on a node, Berkeley NOW................................ 10

Figure 2.2-B. Total node-hours used by each user (sorted by node-hours used). 11

Figure 5.1-A. Overview of PBS from a user's perspective... 37

Figure 5.1-B. Overview of PBS from a developer's perspective. 39

Figure 5.2-A. Internal structure of the PBS Economic Scheduler. 46

Figure 5.2-B. Economic scheduling algorithm. 49

Figure 5.2-C. Sequence of events upon suspension or resumption of a distributed PBS job. 56

Figure 6.2-A. Internal class structure of the simulator. 61

Figure 6.2-B. Sequence of events on each scheduling run. 63

Figure 7.1-A. Histogram of job durations for the existing GLUnix system. 72

Figure 7.1-B. Histogram of job parallelism for the existing GLUnix system. 73

Figure 7.1-C. Average duration of each user's jobs. 74

Figure 7.1-D. Average parallelism of each user's jobs. 75

Figure 7.1-E. Mean running jobs on each node (approximate node load average) over time, existing GLUnix scheduler. 76

Figure 7.1-F. Histogram of job delays, GLUnix scheduler. 77

Figure 7.1-G. Sum total of nodes requested in the queue, over time, PBS economic scheduler. 82

Figure 7.1-H. Total distinct users with jobs in the PBS system, over time. 83

Figure 7.1-I. Median bid in the PBS system, over time. 84

Figure 7.1-J. Maximum bid in the PBS system, over time. 84

Figure 7.2-A. Histogram of all job durations. 91

Figure 7.2-B. Histogram of all job durations (zoomed). 91

Figure 7.2-C. Degree of parallelism of all jobs. 92

Figure 7.2-D. Number of jobs run by each user. 93

Figure 7.2-E. Node-hours consumed by each user. 93

Figure 7.2-F. Mean delay over time, GLUnix scheduler. 103

Figure 7.2-G. Mean delay over time, FIFO scheduler. 103

Figure 7.2-H. Mean dilation over time, GLUnix scheduler. 104

Figure 7.2-I. Mean dilation over time, FIFO scheduler. 105

Figure 7.2-J. Histogram of job dilation, GLUnix scheduler. 106

Figure 7.2-K. Histogram of job dilation, FIFO scheduler. 106

Figure 7.2-L. Job delay vs. job bid, Economic scheduler, Constant-Total-Bid users. 108

Figure 7.2-M. Job delay vs. job bid, Economic scheduler, Constant-Total-Bid users: constructed set of line segments. 109

Figure 7.2-N. Job delay vs. job bid, FIFO scheduler, Constant-Total-Bid users: constructed set of line segments. 110

Figure 7.2-O. Piecewise linear division of Figure 7.2-M by Figure 7.2-N. 110

Figure 7.2-P. Job delay vs. bid, all jobs, GLUnix scheduler, Constant-Total-Bid User module. 112

Figure 7.2-Q. Job delay vs. bid, all jobs, FIFO scheduler, Constant-Total-Bid User module. 112

Figure 7.2-R. Job delay vs. bid, all jobs, Vickrey economic scheduler, Constant-Total-Bid User module. 113

Figure 7.2-S. Ratio of delays, Vickrey economic scheduler and FIFO scheduler, for users' jobs bid using the Constant-Total-Bid bidding scheme. 114

Figure 7.2-T. Ratio of delays, Vickrey economic scheduler and GLUnix scheduler, for users' jobs bid using the Constant-Total-Bid bidding scheme. 114

Figure 7.2-U. Job delay vs. bid, all jobs, GLUnix scheduler, Random-Bid User module. 116

Figure 7.2-V. Job delay vs. bid, all jobs, FIFO scheduler, Random-Bid User module........... 116

Figure 7.2-W. Job delay vs. bid, all jobs, Vickrey scheduler, Random-Bid User module....... 117

Figure 7.2-X. Ratio of delays, Vickrey economic scheduler and FIFO scheduler, for users' jobs bid using the Random bidding scheme.. 118

Figure 7.2-Y. Ratio of delays, Vickrey economic scheduler and GLUnix scheduler, for users' jobs bid using the Random bidding scheme.. 118

Figure 7.2-Z. Job delay vs. bid, all jobs, GLUnix scheduler, Proportional-Bid User module. 120

Figure 7.2-AA. Job delay vs. bid, all jobs, FIFO scheduler, Proportional-Bid User module. 120

Figure 7.2-BB. Job delay vs. bid, all jobs, Vickrey scheduler, Proportional-Bid User module. 121

Figure 7.2-CC. Ratio of delays, Vickrey economic scheduler and FIFO scheduler, for users' jobs bid using the Proportional bidding scheme. 122

Figure 7.2-DD. Ratio of delays, Vickrey economic scheduler and GLUnix scheduler, for users' jobs bid using the Proportional bidding scheme 122

Acknowledgements

I'd like to thank my advisor, David Culler, for his support, exceptional patience, and insights; Joe Hellerstein, who gave me a great deal of helpful feedback on a prior draft of this Master's report; Steve Gribble, David Wagner, and Ian Goldberg, for being such excellent role models, even when they didn't know it, and Alan Mainwaring, for his advice, support, and insight into what it really means to be a graduate student.

I'd also like to thank my parents and family, whose constant support has meant so much to me over the years, and specifically my grandfather, for encouraging me so strongly to pursue these endeavors. Finally, I'd like to thank my closest friends, who have been with me through so much and have shown me support and encouragement beyond any reasonable expectations. Thank you.

1 Introduction

Networks of workstations (NOWs) [1] have become an increasingly popular and scalable high-performance computing platform in recent years, driven by the availability of high-bandwidth, low-latency switched networks [2, 3]. As the popularity of these clusters grow, so, too, do the demands placed upon them, from processor-intensive scientific simulations to data search and processing tasks [4]. Of specific interest are growing sets of Internet-scale applications [5] that demand real-time responsiveness and use internal architectures that are much more similar to those of high-scale database systems [6] than traditional supercomputing tasks.

The supercomputer of yesteryear, running a relatively specific task or set of tasks, is being replaced by general-purpose clusters, ready to run any one — or several — of a wide range of tasks. This shift in tasking demands a parallel shift in resource management: the algorithms we have used to manage supercomputers in the past will not provide the flexibility and performance required by these new applications.

A great deal is known about managing the resources of traditional, time-shared computers and personal workstations [7]; the literature is rich with various algorithms, each of which obtains one of a wide variety of desired results. Similarly, much work has been done on scheduling large, parallel supercomputers running traditional "supercomputing" tasks; while still not as well understood as smaller machines, analysis and optimization of scheduling algorithms for various tasks is also plentiful [8], suggesting that such machines are also well-understood.

An ongoing research project at the University of California, Berkeley is considering the new "third class" of system: high-performance clusters of workstations, built and used to run — often simultaneously — a variety of tasks, from traditional supercomputing jobs to new applications that arise from the growing trend towards ubiquitous computing and Internet-scale services [5]. Such systems present new challenges in resource management due to the wide application domain and disparate user demands on the system; traditional techniques for managing small-scale systems can fall apart at such large scale, and algorithms for handling traditional supercomputer workloads cannot account for the new types of applications being developed for such systems.

In this paper, we consider the use of *computational economies* for managing the resources of such a system; specifically, we describe our experiences designing, implementing, using, and analyzing a system that uses economic principles — in this case, "money", in the form of credits — to expose the underlying supply and demand of the system to users and applications directly. Our system is directed towards those users running "batch-mode" jobs: jobs that have a significant runtime and do not, in general, interact with the users; this subsumes a large portion of traditional "supercomputer" tasks and provides us with a known, understood workload.

Given this workload as a target, we describe various economic models that may be used to build a computational economy, and describe our reasons for choosing a specific model. We then detail our specific implementation of this model, paying particular attention to decisions that affect end users. We present results of this model over time on production clustered systems, analyzing the behavior of users and the system; we also apply a simulator to historical user data and present the results that our economic system would have produced given this input data. Finally, we analyze the behavior and results of users and the system, reflecting on our model, implementation, and possible future directions.

This thesis is organized as follows. Section 2 gives further background for the ideas contained within and describes the motivation for our work, as well as describing related work. Section 3 describes the pre-existing system in our test environment. Section 4 provides the economic theory leading us to choose a particular model; section 5 describes the implementation of this model in our system. Section 6 describes the simulator we have constructed to analyze historical data as if it were placed under the control of our economic scheduler. Section 7 provides the results of our work and analysis of these results; section 8 describes the conclusions we have drawn from our work. Finally, section 9 critiques our work and provides possible future directions.

2 Background, Motivation, and Related Work

2.1 Background

The idea of applying economic principles to resource-allocation algorithms in computer systems, while not new, is also not yet well understood. We first consider the basic principles underlying nearly all resource-allocation algorithms in computing systems today, and explain the fundamental rationale behind current schedulers. We then consider the well understood case of single-computer schedulers; observing that schedulers on these machines have reached a point of relative stability, we offer a potential explanation and consider whether or not the same reasons can apply to clustered systems. We then take a more direct look at clustered systems, characterizing their resource-allocation needs more firmly and describing how their varied workloads may cause the same techniques used in single-computer schedulers to fail. Next, economic systems in general are described; we consider how the general properties of economic systems may be of benefit (and hindrance) to resource-allocation decisions in a computing system. Finally, we consider how economic and computational systems may be fused, describing the benefits of such a fusion and why such a policy may be the ideal solution for clustered computing systems.

2.1.1 Key Metrics for Scheduling Decisions

As previously noted, the literature is extraordinarily rich in resource-allocation policies for the most common computer systems: for processor scheduling on time-shared computers alone, there are first-in–first-out schedulers, shortest-job-first schedulers, round-robin schedulers, priority schedulers, lottery schedulers [9], stride schedulers [10], and so on — and so on. Each of these schedulers provides different behavior; while some differ slightly and some differ a great deal, they all provide distinct functions for determining which job to run at a given time.

Characterizing such schedulers is typically done via reference to their actual scheduling functions, *i.e.*, a description — whether mathematical, algorithmic, or in natural language — of how the scheduler chooses which job to run at a given point in time. This is a useful characterization when attempting to implement or simulate schedulers, or when trying to compare widely-varied schedulers.

However, we offer here a second characterization, one which is more germane to our work: characterizing a scheduler by "what it optimizes for" — the *metric* that it attempts to optimize. That is, a FIFO scheduler tries to keep the computer system as busy as possible by minimizing context switches, thus optimizing for system throughput; a round-robin scheduler tries to minimize the variation among processes' fraction of CPU times, optimizing for scheduling "fairness". Indeed, we suggest that in many cases, given a highly precise description of the metric for which a scheduler optimizes, the scheduler implementation itself inevitably follows: there is, or nearly is, a one-to-one relationship between schedulers and metrics.

In other words, once the metric that is to be optimized has been chosen precisely, the scheduler itself is already determined. If the metric is chosen precisely enough, a single scheduler results. Instead, then, of discussing the details of particular scheduling algorithms, we can simply discuss the metrics that are to be optimized. This ability will be of special interest to us in the following discussion.

We note here that certain classes of scheduler — "parameterized" schedulers — are exceptions to the preceding discussion: for example, stride schedulers, lottery schedulers, and weighted round-robin/priority schedulers do not inherently optimize any particular metric. Instead, these schedulers are used as a platform on which to implement other, higher-level scheduling decisions, ones which either have their own algorithm and own metric for optimization or are simply manually adjusted to produce the desired results.

2.1.2 Schedulers for standalone systems

Traditional, "standalone" computer systems — designed to serve a single user or be shared between a number of users — have had by far the most extensive research with respect to scheduling. We consider here just a few of the various CPU resource-allocation algorithms that have become well known over the years; with each algorithm, we list the particular metric that it attempts to optimize:

FIFO — system throughput
Round robin — equal time to all processes; interactivity
Shortest-job-first — job latency

Shortest-remaining-time — job latency

Priority — minimize product of priority and delay

Lottery/stride — deviation from manually-set processor fractions

Of particular interest in this chart is a question posed by the list of metrics: *where*, exactly, did these metrics come from? That is, why did the designers of these schedulers choose to optimize these particular quantities?

Inevitably, these designers chose these particular metrics because they believed the metrics came closest to approximating the desires of users (and/or administrators) using the systems at which the algorithms were targeted. That is, the designers observed the system in question, considered the priorities of the user(s), came up with a metric to express those priorities, and then designed a scheduler to optimize that metric.

This strategy is effective precisely because, on a standalone computer system, the priorities of a user or a few users typically *can* be expressed in a single, fairly straightforward metric; if there are more than a few users, there is nearly inevitably a system administrator, whose priorities are the *real* desires that must be properly expressed. A user may desire interactive response time, or throughput; a system administrator may desire to rank jobs according to priority, and have them executed strictly that way.

Simply by matching the users' — or administrator's — desires with algorithm metrics, an acceptably good (and often excellent) scheduler can be selected for any given task. This is, indeed, the method by which nearly all computer systems have operated up to the current time.

As an example, the vast majority of personal computers, workstations, and even large time-shared systems today use one variant or another of the priority-based, round-robin scheduler. This scheduler optimizes for "fairness" among processes, giving multiple processes with the same priority roughly equivalent amounts of CPU time; priorities can be used to bias the system (usually very heavily or entirely) in favor of certain processes. Why is this a good metric for current systems?

Essentially, we find this scheduler useful because its artificial metric aligns well with the expectations users have of the system: it gives all processes roughly the same level of service, and splits its attentions evenly among a user's various requests. Heuristics are used to ensure that no program is left entirely unattended; priorities are nearly always used simply to indicate that a particular ("background") task is of little importance or, by the administrator, to boost a particular process's importance.

However, there remains one critical point: these schedulers do not *directly* attempt to behave in a way consistent with users' desires at any given point in time — that is, they do not *directly* take into account a user's particular requests over time; rather, they work well because the metrics of the algorithm *happen* to coincide well with the desires of the user(s) or administrator (who can act as an "intermediary", manipulating the scheduler to bring about the desired results). While this is nearly always the case in traditional, small-scale systems, a more interesting area appears when such coincidence does not exist.

2.1.3 Schedulers for Clusters and Distributed Systems

In stark contrast to the incredible ubiquity of the priority-based, round-robin scheduler among small-scale systems is the wide variety of customized schedulers in use at very-large-scale computing facilities. Supercomputers are typically scheduled using batch queues, and the popular software packages that implement these queues typically provide a great deal of flexibility in individual site configuration — sometimes to the extent of allowing a site to easily define a fully-custom scheduler. While many sites use some variant of a first-in, first-out queue scheduler, there is certainly a great deal more variance among schedulers than in the small-computer case. Further, large-scale computing sites nearly always involve a great deal more administrative manipulation of the scheduler and individual jobs than particular computer systems.

The reasons for this wide variance among schedulers — and the need for more administrative involvement — are many. Part of this variety is simply the desire to squeeze every drop of work from these large, expensive systems; another part is that they simply have much more supply of and demand for resources than smaller systems. However, these two facts together do not preclude the existence of an efficient scheduler for such systems; indeed, we argue that there is a third factor explaining why such schedulers tend to be heavily customized and still must be "tweaked" to perform satisfactorily.

Specifically, we argue that there is as of yet no metric similar to the ones used for small-scale systems that can adequately capture the wide variety of demands placed on large-scale systems by their many users and administrators. Moreover, this lack of an adequate metric will only become more apparent as high-performance computing platforms gain new, more varied applications, driven by the era of ubiquitous computing and Internet-scale services. As the number and variety of applications continues to increase, no single "manufactured" metric can provide systems with sufficiently optimal scheduling to satisfy their needs.

Instead, these systems would ideally use a more direct method of determining their optimal scheduling: instead of developing a metric that happens to coincide with users' (or administrators') desires, they would directly *use* the desires of users or administrators in the algorithm, optimizing directly for something like the sum of each individual user's satisfaction with the system coupled with low variance among users' satisfaction levels. We do not argue that such a system is necessarily easy to build, only that it would be desirable; with each user satisfied to the greatest degree possible given the other constraints of the system, the system would, in some sense, be completely optimal.

2.1.4 Economic Systems

Fortunately, a model exists in human society for how to come up with this sort of optimized solution in a *distributed* fashion: the normal mechanisms of human economies perform this sort of distributed resource allocation in a highly optimal fashion each and every day. By devolving the issue from a centralized, command-and-control approach — which has been demonstrated to fail in human economies — to individual economic decisions, resource-allocation problems that could not otherwise be solved are dealt with efficiently and effectively. The key to this approach is the use of a *currency* in resource-allocation decisions.

By expressing demand for and supply of resources in terms of a common intermediary — *money* — economies no longer require a central policy for resource allocation. Without money, users are reduced to arguing about "who needs the resource more" — which is an impossible question to resolve, because interpersonal comparisons of utility cannot be resolved objectively. By introducing money and the concept of a limited budget to a user, users instead translate their desires into a common currency and thus provide systems with the necessary information to make optimal allocation decisions.

We believe that this is a crucial design point for considering next-generation resource-allocation techniques: if resources are scarce, it becomes absurd in many cases to simply *ask* users "who needs the resources more"; when demand is very high, such exercises inevitably come down to statements of "I need it more" and must be mediated. (We ask the reader to consider what would happen if luxury automobiles were to be given to "whoever needs them the most".) However, by introducing money, users *can* compete for the resources with an interpersonal medium: we endeavor to create a system in which, if one user offers $10 for a resource and the other $20, it is because it truly *is* more valuable to the second user.

The advantages of building a resource-allocation algorithm around such a mechanism are numerous. Most importantly, such a resource-allocation mechanism can be fully distributed and far more robust than a traditional algorithm: if set up properly, users' demands, expressed via money, are the sole input the system needs, and no administrative overrides should be required. Each component of the system should be able to allocate itself properly by simply acting in its own economic best interest; if the system is structured properly, components following this model will derive allocations most beneficial to their individual users (whether those users are humans or other computer systems). By allowing users to translate their own preferred system-optimization metrics into a "global metric" of money, the system can attempt to optimize for the combination of all users' individual metrics.

There are, however, some disadvantages to such economic systems. Most importantly, economic systems in the real world are *complex*: despite the combined efforts of millions of the brightest humans worldwide, it can still easily be argued that nobody truly understands why the global human economy behaves in the ways that it does. When stable, such complexity does not pose a problem to the end users of the system; however, when the system goes through sudden periods of rapid change, the complexity of such systems is often a serious problem as there is no way to predict nor to explain such change. Another issue is *inequity*: to what extent can end users be certain that they are each given a "fair share" of the resources, and what share is considered "fair"? In general, however, economic systems lack a "global system view": there is no way to gain visibility into the system as a whole and explain its inner workings; this lack of a global system view is a major issue in any system implementing an economy and can profoundly affect participation from users and administrative acceptance of such a system.

2.1.5 The Synthesis of Economic Systems and High-Performance Computing

We believe that the inherent properties of economic systems and the difficulties faced by resource allocation in the large "interclusters" of workstations in tomorrow's computing world are a natural match. By using economic ideas to construct scheduling systems for such large interclusters, the natural distributed decision-making of economic systems can be exploited to form fault-tolerant resource allocators. More importantly, however, economic systems allow the users of such systems to specify their demands on the system *on their own terms*, allowing them to propagate their own decision-making metrics about the importance of jobs to the system's own scheduler, rather than having to rely upon the system's own defining idea of "good behavior" matching their own. Further, the enforcement of limited budgets of money on end users requires them to allocate their priorities rationally and accept the resource limitations of the underlying system.

Due to the distributed nature of the Berkeley Millennium intercluster, and due to the widely varied user population that places demands upon it, we believe that an economic resource-allocation mechanism therefore provides substantial opportunity to improve on existing scheduling policies while potentially providing a distributed, fault-tolerant policy for a wide-area cluster.

2.2 Motivation

We chose to implement a computational economy on our research clusters of workstations and PCs due to our unique environment and opportunity. Patterns of use in our cluster in the past demonstrated a direct need for a better resource-allocation algorithm than we had been using in the past, and our research environment is unusually receptive to such changes. Further, the distributed intercluster, or "cluster of clusters", that we are currently building almost *necessitates* such a solution: it is decentralized to the point that a single, centralized resource allocator would become a bottleneck and likely do more harm than good.

2.2.1 The Pre-existing System

We consider now the Berkeley NOW cluster, our primary parallel-computing platform for several years and one still under substantial use. The cluster is composed of approximately 100 Sun Ultra 1 workstations, each containing a single Sun UltraSPARC 1 microprocessor operating at 167 MHz, a moderate amount of local disk space (ranging from one to four gigabytes), 128 MBytes of RAM, and both standard Ethernet and high-performance Myrinet [3] network interfaces.

Use of the cluster is widespread: it has been used by various departments around the campus, especially electrical engineering, to conduct scientific simulations of various physical phenomena, model proposed circuits, test new network protocols and hardware, provide a platform for many new cluster software systems and languages, and so on. Of particular interest to us are the usage patterns of the cluster over time. Unlike a commercial or governmental supercomputing center, we do not sell or rent time, and thus usage of the cluster tends to vary widely over time.

We delay a thorough analysis of cluster usage data until section 7.1.1; however, we give here some basic data and anecdotal examples of cluster usage to demonstrate one part of our motivation for implementing a computational economy.

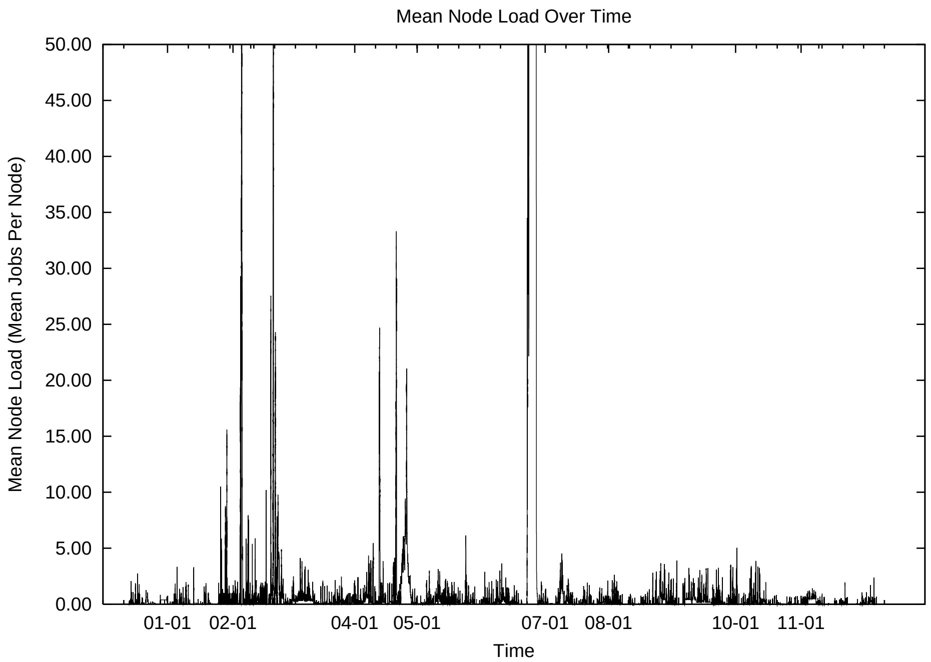


Figure 2.2-A. Mean number of running jobs on a node, Berkeley NOW.

Figure 2.2-A demonstrates the mean number of jobs, per processor, on the Berkeley NOW over the half year January 1, 1999 – July 1, 1999. As is evident from the graph, the NOW experienced very long periods of underutilization, followed by very short periods of extreme overload: at times, there were as many as twelve competing jobs on each node. This degree of time-sharing inevitably exhausts the NOW's physical memory, causing the nodes to "thrash" and essentially stop all useful work.

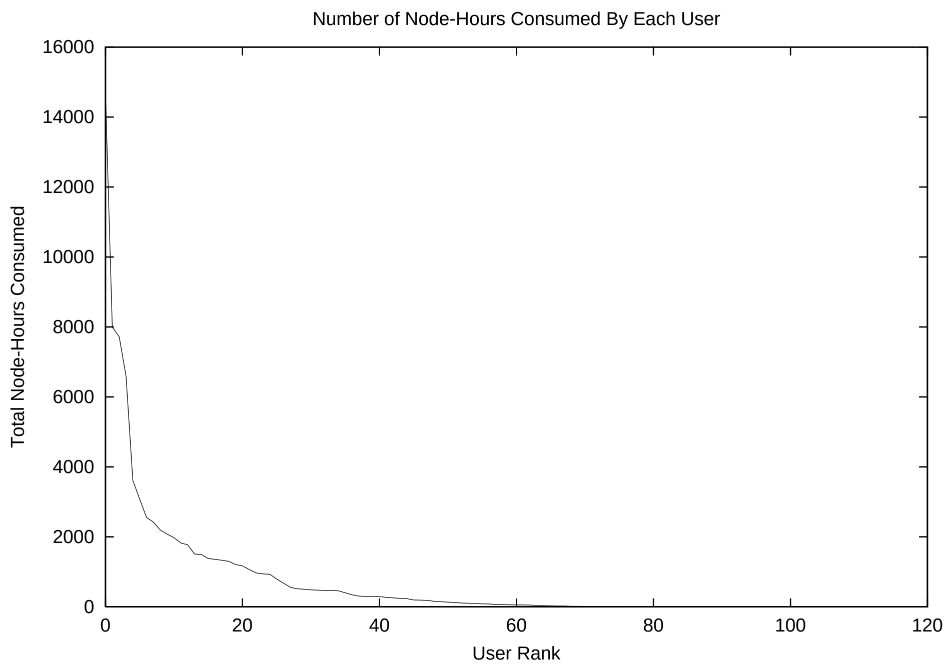


Figure 2.2-B. Total node-hours used by each user (sorted by node-hours used).

Figure 2.2-B shows the total number of node-hours of processing time consumed by each user of the Berkeley NOW over the time period studied. The graph roughly fits a Zipfian distribution (exponential decay), showing that while some users used many thousands of node-hours of cluster time over the year, other used very little. In fact, the top six users (5.6%) used as much cluster time (the equivalent of the entire cluster for 17.16 days straight) together as all the other users, *combined*; the top user alone used as much cluster time as the bottom 97 users (81.5%) combined. In general, the data fits a typical "80/20" pattern: the top 20 users (16.8%) used 79.9% of all cluster time. While the objective of a cluster scheduler is certainly not to ensure that each participant uses exactly the same amount of resources as each other participant, these figures clearly demonstrate the potential for unfairness and user dissatisfaction when the cluster is scheduled with standard resource allocators.

Of perhaps even more interest are the subjective observations made of user usage patterns and behavior with respect to each other. In general, we found that users tended to use the system with very little or no consideration given to its current load. Even during periods where the NOW was clearly overloaded, users continued to add jobs to the system even though they could not have reasonably expected them to get any serious work done; the basic cluster status monitoring tools were not even invoked often. In other words, demand was not at all responsive to supply. Certainly part of this was due to users' needs; much of this work was done as class assignments with specific due dates, or for conference deadlines, but the demand still remained remarkably inflexible.

Conversely, long periods existed in this data during which virtually no work was done on the cluster. Given that use of the cluster was essentially given, *gratis*, upon demand to anyone in the entire Berkeley community, this also is a problem: our cluster was greatly overloaded at some points, and sat entirely idle at other points.

These two situations, taken together, make a strong case for introducing additional feedback into the Berkeley NOW cluster, and also suggest that such feedback would be of use if incorporated into the forthcoming Berkeley Millennium "intercluster".

2.2.2 A Unique Opportunity

Given this need for *some* better form of feedback in the environment today, we also considered our unique environment here as an advantage when motivating development of this system. First, and foremost, our clusters are dedicated to research, and this gives us the flexibility to try different resource-allocation models without fear of disrupting a production system. Second, due to the confluence of work arising from shared homework and conference deadlines, our cluster generates substantial periods when demand outstrips supply. Finally, and of great import to this experiment, our users tend to have widely varying demands: some use the cluster for large-scale scientific simulation, some for research into new networking protocols, some for on-demand "services" that vary based on user demand, and so forth.

Additionally, our research goals to construct a very-large-scale campus network of clusters — an "intercluster", where individual small clusters are connected into one large one— motivated much of this work into computational economies. Such a large, distributed cluster system has several characteristics that make a computational economy appealing. It has highly distributed administration — as each campus department administrates their own particular cluster, maintaining a central scheduler of any sort becomes politically difficult or impossible. Also, engaging a centralized scheduling algorithm impacts the robustness and fault-tolerance of such a system, as individual segments of the network of clusters become useless if disconnected from the central scheduler.

Finally, we consider the rapidly-approaching world of truly ubiquitous computing: as computing power and presence becomes as assumed a part of life as, say, modern plumbing or electrical work, the idea of "reserving" large pieces of such an infrastructure becomes increasingly absurd. Just as a company cannot simply reserve a power plant or two directly from Pacific Gas & Electric "just in case they need it", one cannot reasonably expect to reserve a large piece of core computing infrastructure. Instead, an entirely different model is needed.

2.2.3 A Large-Scale, Real-World System

Most importantly, however, our research was motivated by the need for *real-world* results of systems that involve real users and computational economies. As we will see in the next section, a great deal of theoretical work governing computational economies has been executed; a fair number of computer systems use economic models internally; yet very few systems to date have involved users with computational economies directly. We aim to gain experience with computational economies that expose the supply and demand of the underlying system to users directly.

2.3 Related Work

Economic systems themselves have, of course, been around since almost the beginning of human civilization. In general, these systems evolved around the notion of the transfer of fixed-unit goods — food, clothing, or shelter; today, Palm Pilots. As simple bartering progressed, currencies were invented as a common intermediary representing the value of goods. Systems of barter, sale, and auction evolved to meet the needs of those wishing to acquire or dispose of these goods. Trading resources such as time on a cluster of computer workstations, however, is significantly more complex: demand is highly variable, often the exact requirements of a user are somewhat unknown, and the good itself — the computer time — is constantly expiring and being renewed over time. Computer time that passes unused can never be recovered, but there is an essentially infinite future supply yet to be provided.

This places cluster resource allocation in a similar vein to allocation of electrical power at electrical power plants, scheduling of job shops and airports, and the allocation of natural gas pipelines. Banks, Ledyard and Porter, in [11], give an excellent overview of some of the challenges posed by such systems and present a computer-assisted mechanism for using auctions to meet some of the challenges therein. Study of particular systems that bear resemblance to ours extends far back in history; even by the end of the nineteenth century, studies of how to price the nascent electricity industry were underway [12]. Of course, not all decisions regarding these systems are made on the basis of efficiency alone, as we have assumed here; Hillman and Riley, in [13], present a model for the analysis of these systems when political influence can be exerted over the results.

The direct application of economic principles to computer systems has been fairly substantially studied in the literature, although relatively few systems expose such economies to the user. In [14], Cocchi, Shenker, Estrin and Zhang demonstrate the application of economic models to computer networks and the resulting classes of service that can be developed, including simulation of such networks. Ferguson, Nikolaou, Sairamesh, and Yemini, in [15], present a cogent argument for the application of economic principles to large, decentralized economic systems, although they study the application of such principles at a low level, invisible to the user. In [16], Walsh, Wellman, Wurman, and MacKie–Mason present a detailed analysis of the applicability of economic principles to classic scheduling problems and consider two auction protocols for using economic methods to solve scheduling problems.

A recent direct application of economic principles to a distributed computer system is found in Spawn [17]; it shares a great deal in common with our own work, although its concentration was much more on the architecture and mechanisms of economic resource allocation than the eventual interaction with human end-users. Similarly, Mariposa [18] provides users with the ability to determine individual bid curves that are then combined using economic resource-allocation policies. By comparison, Popcorn [19] uses economies internally to allow resource allocation over wide scales, but does not in general expose this economy directly to the end user. Mariposa is a distributed database system that uses microeconomic principles to allocate resources for and process queries. Popcorn presents the entire Internet as an enormous potential distributed system; it allows programmers to use the portable Java programming language to create distributed programs and use its built-in mechanisms to compete in economic markets for various resources.

Similar to Popcorn, Wellman and Wurman's study of mobile programs, or "agents", and their behavior in an economically-scheduled environment [20] presents a case for enabling agents to understand an economically-modeled environment. Chislenko and Ramakrishnan, in [21], provide a proposal for a framework of semantic elements that can be used by such agents to communicate amongst each other regarding an economically-modeled environment.

Finally, REXEC [22], concurrent work at the University of California, Berkeley, exposes computational economies directly to the end user on a strictly time-shared basis; rather than winning or losing an auction outright, users can simply adjust the share of the CPU that they receive by varying their bids. At the time of this writing, work is underway to extend this software to a batch-computing environment [23].

3 The Existing System: GLUnix on the NOW

As a point of comparison, we now examine the execution environment of the Berkeley NOW prior to introduction of the computational economy. The execution layer of the system is Global Layer UNIX (GLUnix) [24].

3.1 GLUnix Goals and Design Point

3.1.1 Time-shared, immediate-execution

GLUnix is, first and foremost, a remote-execution layer designed to share a cluster of workstations through *time*: although jobs run on the nodes in the cluster that are the least loaded, no job is ever prevented from running. Rather, if the various users demand more nodes than are actually available, some (or all) nodes will end up running multiple processes at a time. The underlying, single-node operating system's scheduler is given the responsibility of scheduling the CPU among the various jobs.

This method of scheduling the cluster provides an interface familiar to all users: in many ways, the cluster behaves as a very-large-scale, traditional time-shared system. When the system is idle or underloaded, users' jobs run immediately and effectively; as system load increases, the performance of the cluster slowly deteriorates until a saturation point is reached at which, typically, the memory demands of the cluster jobs are too great to fit into an individual node's physical memory. At this point, the cluster begins thrashing, and essentially no further progress occurs.

3.1.2 Issues with Time-Sharing and Implicit Coscheduling

Time sharing of a cluster is at odds with the traditional space-sharing and queued environment of many supercomputers, and thus presents some immediate challenges to a cluster toolset. Of particular interest is the behavior of fine-grained, highly parallel large-scale programs when time-shared; many find their performance drops by a much larger degree than the time-sharing factor due to the lack of global gang scheduling. Implicit coscheduling [25] has been developed to resolve this problem and, in general, works very well.

3.1.3 Reservations and Privileges

Because GLUnix executes all jobs immediately, providing no queue to users, problems arise when exclusive use of a node or nodes is required for a particular research project. Without a queue, users are unable to guarantee themselves uninterrupted execution of jobs and have difficulty scheduling jobs to run non-interactively — *e.g.*, late at night or on the weekends, when cluster use is relatively low. While the problems that arise are not terribly common, we find that these problems occur with some frequency in our environment; typical causes are a need to test new messaging software, requiring reboot of individual nodes, very precise benchmarking of codes that require no daemons or other processes to be present, or similar situations. GLUnix does provide a facility for this, the *reservation model.*

A tool called `glupart` can be used by users with sufficient privileges to create a *reservation* on the NOW. A reservation consists of a designated set of machines for a designated (possibly non-contiguous) time period; during that period, GLUnix will refuse to place jobs on the nodes contained in the reservation unless the user starting those jobs is named in the reservation's set of users.

Of particular note is the "sufficient privileges" part of the above stipulation: in our environment, "sufficient privileges" are granted by being a member of a UNIX group called `now`. The members of this group are typically those students and faculty actively doing research on the cluster itself, and the system administration staff.

3.2 Resources

We now turn to a brief description of the actual cluster infrastructure for the Berkeley NOW [26].

3.2.1 Computers

The Berkeley NOW consists of approximately 100 Sun Ultra 1 computers; each conforms to the following specifications:

Model	Sun Ultra 1 Model 170
CPU	1 Sun UltraSPARC I at 167MHz
Memory	128MB DRAM
Virtual Memory	1GB swap space provided
Disk Storage	2GB for operating system and applications; 2GB user scratch space.
Infrastructure Network	100Mbit/s Fast Ethernet

3.2.2 Network

The cluster is also supported by a high-speed switched network, Myricom's Myrinet [3] system-area network. The raw network hardware supports link rates of 1.260 Gbit/s; the Active Messages message-passing interface supported over it achieves an end-to-end latency of about 14 microseconds and a half-power bandwidth point of 31.7 MByte/s with 8.7–KByte messages [27].

3.2.3 Infrastructure

There is also some miscellaneous support infrastructure to handle the administrative and storage needs of the cluster. An administrative front-end machine provides centralized services for GLUnix (and, later, an economic queuing environment); users have a choice of a single front-end Ultra 1 server or several multi-processor Sun Ultra Enterprise 5000 servers to run sequential jobs and dispatch parallel job requests to the actual NOW nodes. Further, departmental infrastructure provides a shared filesystem space for the entire cluster.

3.3 User Behavior and Collected Data

Next, we describe general usage patterns of the cluster in the past as run under the existing GLUnix execution environment. We delay precise analysis of this data until section 7, and consider here a generalized description of observed user behavior, both from personal experiences and as a summary of the data presented in section 7.

3.3.1 General behavior

In general, we find that users tend to almost entirely ignore supply from the cluster and consider only their own personal demands: users start jobs when they need or want to with little attention to how many other users are simultaneously running jobs or how much of the cluster remains free. While sufficient cluster monitoring tools are provided to determine the load on each node in the cluster and the total workload placed on the cluster at any given moment, we find that they are rarely invoked (or, if invoked, seemingly ignored): very long periods of no or little cluster use exist, while during busy periods users continue to submit jobs to an overloaded cluster — even though no useful work can be accomplished.

In some sense, this behavior is encouraged or exacerbated by the very nature of the tools: rather than impose some sort of admission control or queuing environment on the cluster, they will happily accept and run far more jobs at once than the cluster can handle, causing rapid deterioration of performance for all users simultaneously. Further, this lack of a queue makes it difficult at times for users to take best advantage of the cluster's resources: since jobs are started interactively, it is very difficult to start jobs during holidays or late at night, when the cluster is otherwise idle.

Fundamentally, the GLUnix tools encourage relatively reckless user behavior because they offer little feedback to the user on the current state of the cluster and because they provide no *budget* for the users whatsoever: a single user may submit as many jobs as he or she desires, as often as he or she desires, with absolutely no decrease in performance or penalty whatsoever.

Yet another contributor to this unbalanced use of the cluster was the way in which privileges were granted to peremptorily reserve cluster resources for their own use. Essentially, those students actively pursuing research regarding the cluster itself were granted an unlimited power to reserve arbitrary sections of the cluster at will; these reservations were irrevocable and inviolable. Because there was no feedback regarding these reservations, they tended to be overused; worried users would reserve far more resources than they actually used or needed, just to ensure that the resources they *did* need would be available upon demand. (A reasonably common occurrence would see a user reserve 16 nodes — over ten percent of the cluster — for several weeks at a time, although less than thirty percent of this time — and often far less — was ever actually used.) This, in turn, only exacerbated the supply-demand problem during periods of intense use of the cluster.

3.3.2 User Conflict and Conflict Resolution

Naturally, these patterns of use of the cluster have created various sources of conflict among the users: while the cluster is very busy running one person's long data-analysis task that easily could withstand a full day of delay, another user has a homework or paper deadline a few hours away and very urgently needs full use of the cluster.

Generally, this sort of resource-allocation conflict has been resolved on the Berkeley NOW simply by interpersonal means: users would use the cluster status tools to determine *who* was in conflict with their desires, find that person, and come to a resolution verbally. While, in general, this system was effective, it was not efficient. Sometimes users refused to yield, and significant interpersonal conflict ensued; more importantly, a large amount of valuable time was consumed just trying to resolve fairly simple resource-allocation conflicts: graduate students are not always known for being highly accessible or consistent in their hours, and so hours or days could be wasted trying to track down the owner of a reservation or debating the relative merits of one user's work versus another's. Administrative tools and overrides were available in very urgent situations, but, due to the academic environment, users and system administrators were quite reluctant to force the issue, and so many valuable resources were simply wasted.

Overall, however, the most important issue was simply this: users did *not* pay much attention to either how many resources they were using *nor* how many resources others were using. Users couldn't tell what was happening on the cluster, nor did they have any incentive to behave nicely (in a manner compatible with a large shared resource). In the end, the cluster would go through long periods of little use and short periods of overload, managing to achieve the worst of both worlds.

3.4 The Coming Millennium

We present now the new cluster available at Berkeley, and its unique opportunities and challenges in the realm of resource-allocation techniques.

3.4.1 Cluster Resources

The Berkeley Millennium "intercluster" contains several hundred processors, distributed non-uniformly across the Berkeley campus. The intercluster is distributed as follows: the largest portion by far is composed of a single, several-hundred-processor cluster located in the computer science research facilities; about ten smaller departmental clusters are located in individual departments across campus; and many individual workstations (single-processor or low-scale symmetric multi-processor) are distributed into various offices and facilities in departments across campus. Cluster nodes themselves typically consist of two-way or four-way Intel Pentium II Xeon or Pentium III Xeon computers, running at somewhere between 500 MHz and 1 GHz, and with 1–2 GBytes of RAM.

The individual clusters that compose the "intercluster" are linked internally via Myricom's Myrinet [3] system-area network; the clusters themselves are networked together by Gigabit Ethernet links spanning the campus and feeding into multi-gigabit, wide-area network links across the country.

3.4.2 New Challenges and Opportunities

Of particular interest is the entirely federated nature of this cluster: although the bulk of the processors will remain within the computer science research division, administration of each individual department's cluster is delegated to that department (with assistance from the computer science staff, if necessary). This means that administrative control over the resources and allocation policy of a particular department's cluster can vary widely, from a GLUnix-like model of "anything, any time" to a very strict, department-users-only policy. Because of this federated administration, it is clear that traditional, centralized resource-allocation policies are not just inefficient but quite infeasible.

We believe that this federated nature of the intercluster will provide a unique opportunity for economic models of resource allocation: by causing resource allocation policies and decisions to be distributed (or at least giving them that potential), we not only enable such a large intercluster to function correctly and robustly, but provide vastly greater flexibility — and thus, we hope, efficiency — to the workings of such a system.

It is also clear that the reservation and privilege model previously employed by the Berkeley NOW becomes completely inappropriate here. The Berkeley NOW essentially divided users into two groups: ordinary users were forced to simply run their jobs along with other users, and hope there wasn't too much demand on the cluster; on the other hand, researchers that were working on the NOW itself had full power to reserve arbitrary swaths of resources for as long as they desired, with no feedback (other than interpersonal communication when things got entirely out of hand) to prohibit them from causing frustration for other users. Were this model applied to the intercluster, it seems clear that total chaos would result: it is first totally unclear as to who would be granted this reservation power (computer science researchers only? Administrative staff in any department? Researchers in any department?); further, the "walk down the hall" method of resolving conflicts becomes unworkable if several hundred people, many of whom do not know each other, are given this power; finally, the model is simply inappropriate for such a large, important, shared resource. It is inappropriate for users of one department to be able to reserve such a large shared resource just like it is inappropriate for such users to be able to reserve the campus wide plumbing, or heat, or copy center.

Due to these two major factors — the problems we were having with the Berkeley NOW and the impossibility of maintaining its approach with the new Berkeley Millennium — we decided to pursue implementation of a real-world, full-scale computational economy. Because of the paucity of research that has gone into implementing systems such as this, we started small: we are conducting two parallel research experiments, into a simple time-shared economic model (presented in [22]) and a simple economic batch queue.

4 Choosing an Economic Model

Our first challenge in this system was the choice of a theoretical model on which to base our real-world computational economy. Economic interaction in the external world takes on a great number of forms — simple fixed-price purchases, auctions, interpersonal bargaining, and a practically infinite number of derivatives of these — and so we must choose a form of economic interaction on which to model our computational economy.

4.1 Requirements

We first consider some basic requirements of our computational economy, induced by the nature of our system and the applications and behavior of our users:

4.1.1 Low user interaction

First, and foremost, we believe our economy must impose a *low user burden*, meaning that it must require very little interaction from the users. Notably, it should not require much more interaction than the existing system (GLUnix), which requires, essentially, only that users issue a command to start their job running. At a practical extreme, we cannot require users to modify their programs or write code to interact with the economy. The reasons for this requirement are practical and political: first, imposing additional burden on users would cause a number of them to simply give up on using our cluster, which is an unacceptable result; second, most users of our cluster are interested in the results of their particular program, not in how the cluster works, and would quickly rebel if we attempted to impose a great burden upon them.

4.1.2 Transparent

The second major requirement that we considered for our system is *transparency*: using such a new, radical model for resource allocation requires that we reveal as many details as possible about its inner workings to users and present the entire system as "having no secrets". Because our system may deny or delay service to users, users will want to know *why* their jobs are not being serviced; this requires the system to be transparent.

Transparency is a relative, fuzzy concept: certainly, users will not want to know the *very* deepest details of the entire system, such as network communications protocols, internal data structures, and so forth. Rather, we consider “transparency” here to simply mean exposure of the general, high-level workings of the internal job-scheduling algorithm(s), and enough information so that a user can determine why his or her job is or isn’t running (and, presumably, change that status if desired). We shall see later that this becomes a significant issue: some economic algorithms, by their very nature, require secrecy in the actual scheduling process and thus cannot provide full transparency.

4.2 Supply and Demand

We step back now for a moment and consider the essential factors underlying our — and, indeed, *any* — economy: supply and demand. Although many resource-allocation systems, computational and otherwise, are not viewed in economic terms, the factors of supply and demand still apply and looking at these systems in such terms can be highly informative.

The various components of large-scale clustered systems have differing means of allocation. Processor cycles, for example, can be shared in time, but not in space — nearly all CPUs only execute thread(s) from a single process at once. Disk space can be shared both in time and space; temporary files can be shared in time (existing while a process is running, then later being deleted), while permanent storage can only be shared in space (disks can store files from multiple processes). Network links are actually shared in time, but, to a user program, appear shared in space.

These different means of sharing various resources have profound effects on the economic means used to allocate these resources; space-shared resources are fundamentally allocated in a different manner (and, possibly, using different algorithms) than time-shared resources. For our purposes, we began by considering only CPU cycles as the resource to be allocated. While CPU cycles are hardly the only resource in contention, they are the most fundamental resource for a vast proportion of the existing scientific applications on our cluster — users worry about allocating CPU cycles first, and then worry about allocating memory, disk space, *etc.* in our environment. And, fundamentally, CPU cycles are shared over time, but not space.

Typically, computational systems have a (relatively) *fixed supply* in space: the number of potential CPU cycles, or amount of network bandwidth, or megabytes of disc storage, is fixed, and it is only the demand over time of users that varies. Although this is not strictly true — CPUs and disks can be added, additional network links brought online — we here make the simplification of assuming that supply is fixed. Over short time periods (less than a few weeks, or, more likely, a few months), this holds true: if a cluster of workstations becomes particularly busy at 8 am one morning, administrators are not likely to have purchased additional hardware nodes and added them by 5 pm that afternoon.

If supply, then, is fixed in space, the only varying factor is demand; and, indeed, we can characterize many existing computer systems based on three major factors, in increasing order of importance:

- Aggregate demand, as it compares to aggregate supply;
- Variance of this demand over time;
- Reaction of the system to changing demand.

4.2.1 Aggregate demand of the system

The aggregate demand of a system is important only in the broadest sense, but here it can make a great deal of difference. If a system is generally under-loaded — that is to say, its aggregate demand rarely exceeds, and is typically substantially less than, supply — then it need not provide any sort of complex or very efficient resource-allocation algorithms, because even the simplest algorithms will serve to soak up all demand presented rapidly and efficiently. On the other hand, if a system is often over-loaded, and thus has aggregate demand substantially exceeding supply for some periods, resource allocation algorithms become increasingly important.

If true demand typically is less than supply, there will be a zero "price of admission" to the system; users will be able to submit and run jobs with few or no conditions, and the jobs will run immediately. While this situation will seem strange to administrators of most supercomputers or very-large-scale computing systems, this is because such systems nearly always have demand exceeding supply: if supercomputer time were free to members of the general public, the aggregate demand would surely overwhelm supply. However, if the typical workstation or personal computer is considered, we see that this is not the case: a user can typically start and run programs on their own personal workstation at any time, with no delay and no preconditions.

If true demand *exceeds* supply, however, as is typically the case in most supercomputer installations or high-performance computing environments, there will be a positive "price of admission" to the computing environment. While this price may take one of a great number of forms, it is important to note that it *will* exist in one form or another: users will have to wait in a queue before they are admitted to the system, only certain users will be allowed to access the system, users will be allowed only a particular fraction of the system's resources, users will be charged a certain amount of money for their time on the system, some other restriction on admission to the system will invariably develop, or an external market — the equivalent of "ticket scalping" — will be created that controls access to the system's resources.

4.2.2 Variance of demand over time

The degree to which the aggregate demand upon a system varies over time is sometimes of greater importance in many resource-allocation decisions than the total demand itself. If aggregate demand is high but relatively steady, the price of admission to the system will remain relatively constant, and can be expressed to users simply: "there is a waiting time of three weeks to run on the system", "only faculty get to run on the system", "each user is allocated five hours per week". If demand varies a great deal, however, a constant price of admission no longer makes sense: leaving it set as if the aggregate demand were a constant will result in either under-use of the system (if the price is set for the maximum demand), or a secondary price developing (if the price is set for the minimum demand — for example, if the price is expressed in dollars and is set too low, a queue of users waiting for the system will develop, and the price of admission will naturally expand to include *both* the monetary price *and* the price in time required of the users).

4.2.3 Reaction of the system to changing demand

Finally, and most importantly, the system's reaction to changing demand over time defines its interactions with users. As demand to the system decreases to and past the point of oversupply, certainly all systems will begin to accept any and all user requests. However, as demand on the system increases, the behavior of various systems can vary widely. Some systems (*e.g.*, a traditional round-robin scheduler) degrade performance of all users' jobs equally, while others (*e.g.*, traditional first-in-first-out — FIFO — schedulers) cause any further requests to the system to wait in a queue. Still others (*e.g.*, real-time scheduling systems) cause new requests to the system to be rejected entirely, requiring them to be repeated at a later point in time when the system is not in a period of high demand.

It is this very "price of admission" to a system that defines its interactions with users the most comprehensively: when the system has more request for its services than it can supply directly, how does it give feedback to users? Traditional systems typically either degrade performance smoothly, cause new submissions to wait, or reject new submissions entirely. We consider here, however, a new model, one that allows for greater flexibility in responding to users' demands when the system has more demand than supply.

4.3 Basic Auction Theory

4.3.1 Why auctions?

The chosen economic model for our experimental computational economy is the simple auction: in its most general form, users enter competing bids for the various resources available; users with higher bids are allocated generally more resources than those users with lower bids, but must pay more for the privilege.

We considered several potential economic models for our system, but several factors conspire to make the auction the tool of choice. The first model that was considered, and the leading competitor to the chosen auction model, is a simple set-price policy: a given quantity of resources would be available for a certain price of admission (whether expressed in actual money, or in a waiting time for jobs to be run, or a quota on time consumed by each user per week), with the level set by the administrator. This policy has great advantages in predictability and simplicity; users always know how much it will cost to run a particular job, and essentially no interaction is required from users. Indeed, this policy is nearly identical to that used by many supercomputing centers as of this writing.

While this approach is used quite effectively at supercomputing centers, the unique environment of the Berkeley NOW makes such an approach difficult at best and impossible at worst. While supercomputing centers typically have variation in demand over periods of months or years, our cluster environment experiences vast variations in demand from day to day (and, sometimes, from hour to hour; see section 7.1.1 for details). A fixed-price policy would require the administrators to adjust prices rapidly and responsively in such situations, perhaps with greater accuracy in estimation than is really possible — or risk misbalancing supply and demand, undermining the essential benefit of an economic system in the first place. Further, in an environment such as ours, rapidly manipulating the price to keep up with supply and demand places an unworkable burden on those who must administer the system.

Even more importantly, however, the problem is this: the price of our resources is fundamentally *unknown*. Price is determined at the point where supply and demand are equal; if prices are to be adjusted manually, by humans, that point is determined by experience; we have no experience with pricing in our (fairly unique) environment. It seems unlikely that we could dedicate sufficient resources to administrating and analyzing our cluster to make such a policy worthwhile. Further, we have ample evidence that the price of our resources fluctuates wildly over time, and thus cannot be determined statically; trying to determine a fixed-price policy in the face of a system whose demand can swing by an order of magnitude in a few hours due to essentially unpredictable external events (*e.g.*, the assignment of homework in a particular class in another department) becomes an exercise in futility.

By contrast, the model of a simple auction — where users set the price themselves, by the confluence of their individual bidding decisions — has few of the drawbacks that a simple fixed-price model does. Perhaps most importantly, the simple auction model requires no direct involvement by administrative staff; prices are set only by the individual decisions of the users and thus no guesswork is required. Prices may be fundamentally unknown to the administrators, but the auction process discovers these prices in the process.

The most acute disadvantage of the simple auction is its lack of predictability: users would like a model in which prices are fixed or nearly fixed, so that they can have a reasonable expectation of the cost of accomplishing a specific amount of work. Simple auctions can cause prices to fluctuate wildly, from zero (when there is no demand) to very high levels (when there is a great deal more demand than the amount of resources the cluster can supply).

4.3.2 Types of Auctions

Simple auctions can be distinguished according to several critical characteristics; we present them below. While certain types of auction (*e.g.*, the traditional "English" auction — open-bid, first-price, ascending, single auction) are more common than others, nevertheless, all the different types of auction discussed have valuable properties that make them useful in certain situations.

Sealed *vs.* Open Bid. The distinguishing feature of a *sealed-bid* auction is that only a participant's own bids are visible to that participant — that is, bids are secret, to be told to other participants either only after the auction is complete or not at all. An *open-bid* auction, then, allows all participants to see all other participants' bids during the auction itself, and adjust their own bids accordingly. (We note, however, that this does not imply that all participants are able to see all other participants' *potential* bids; a certain degree of secrecy is what makes an auction successful, after all.)

Ideally, an open-bid auction is preferred for economic models used for resource allocation, because it makes the entire bidding process transparent to all users. Participants are thus all able to inspect the resource-allocation process, both to understand it better and to convince themselves that it is fair.

First *vs.* Second Price. A first-price auction indicates that the winning participant pays the highest price bid for the goods won; a second-price auction, similarly, indicates that the winning participant pays not the highest price bid, but the next-lowest price. Second-price auctions often have a degree of stability that first-price auctions do not — that is, users bid (sometimes much) less frequently, as they are unafraid that by bidding too high they will only drive up their own eventual price [29].

Ascending *vs.* Descending. An ascending-bid auction is a "traditional" auction in which bids start at some fixed low price (ideally, zero), and increase until no bidders are willing to bid higher. A descending-bid auction, by comparison, starts at a price considered so high that no participant will possibly pay it, and lets the price fall until the goods are sold.

Single *vs.* Double. A single auction is one in which only one party (typically, the buyer) offers bids for the goods to be sold; this is the most common type of traditional auction. A double auction, however, is one in which the buyer offers bids for the goods to be sold, and the seller offers various selling bids; the point at which a seller and a buyer of the goods agree on a price is the auction price. (A double auction, therefore, is what is used in stock markets.)

4.3.3 Auctions as used in the real world

We consider now several types of auctions as used in the real world.

English. The traditional "English" auction is probably what most people think of when they speak or hear of a generic "auction". In an English auction, bidding for a certain good starts at a very low price (ideally, zero, but practical concerns often create an "opening bid"); participants offer ascending bids by open outcry, thus informing all other participants of their bid. Bidding stops (and the goods are sold) when no participants bid higher than the previous bid offered.

Dutch. The "Dutch" auction is often misconstrued; a true Dutch auction is an open-bid, first-price, descending single auction. That is, bidding starts at some fixed high price — assumed to be so high no participant would ever actually pay it — and descends, by open outcry of the seller, at a steady pace. When a participant is willing to pay the price indicated, that participant offers an outcry; bidding stops, and goods are awarded to that participant at the current price. Thus only one bidder's bid is ever revealed.

Double. A double auction is typically open-bid, first-price, and both ascending (from the buyer's side) and descending (from the seller's side) at once. The sellers offer lower and lower bids, and the buyers higher and higher bids, until a buyer and a seller meet at a price in the middle; the transaction is then performed at this middle price. Double auctions are often used when there are many buyers and sellers.

Vickrey. A Vickrey auction (named after William Vickrey [28]) is the type of auction often mistakenly called "Dutch": in a Vickrey auction, participants all offer their bid secretly to the seller (a closed-bid auction); the seller selects the participant who offers the highest bid and awards this participant the goods, but only requires payment of the next-lowest bid (a second-price auction).

4.4 Pure Economic Theory vs. Pragmatism

We consider now the task of selecting a type of auction upon which to model our system's economic behavior. A very fundamental choice presented itself here: were we to adhere strictly to the rules of a traditional type of auction, thus allowing direct application of the results of economic theory, or were we to allow ourselves to modify the auction model as we saw fit, thus allowing much more freedom in implementation and adaptation to the changing behavior of the system?

4.4.1 Pure Auction Theory

An ideal approach to the problem would involve using an economic model that was a direct implementation of a traditional auction type: by implementing an auction directly, all of the results of economic theory that apply to that auction could be applied directly to our model. For example, the Vickrey auction is provably "game-free" [29], meaning that it can be proved that users are unable to manipulate the auction system to, for example, require other users to pay more for the goods delivered than would otherwise be possible. Further, very extensive analysis of the theory of many types of auction has been carried out over time, leaving the literature rich in results that can be directly applied to our economic system.

4.4.2 A Pragmatic Approach

Nevertheless, modeling a certain type of auction faithfully enough that derived results for that auction are applicable to the resulting economic system poses several problems. In general, adhering strictly enough to the set of preconditions for a particular type of auction's derived results to be applied limits system flexibility a great deal and greatly constrains the ways in which the system may be altered or "tweaked" in order to better suit users. By making a conscious decision to instead *not* limit a system's behavior strictly to that required by a particular type of auction, much greater applicability may be achieved.

Apropos to our own system, we considered the following issues when deciding whether to strictly follow an auction's rules or not:

- Simply scheduling jobs according to an auction's rules may not always be possible. In particular, the runtime system used (GLUnix) supports neither suspend/resume nor migration of jobs; while suspend/resume was added for this project (see section 5.2.4.2), migration of jobs could not be due to the sheer amount of work involved. Jobs may therefore combine in ways that either require nodes of the system to go idle even when a job is available to fill them, or that requires the system to suspend a job when jobs of lower bids are running.

- We believe that users eventually will want *guarantees*: promises from the system to run their job by a certain time, or that their job will not be interrupted, or that their job will run at least a certain fraction of the time. Traditional auctions provide no framework in which to provide these guarantees.
- We believed (and, later, were confirmed in our belief; see section 8) that the exact economic model used was not of paramount importance; rather, we believed that users would adapt and react to any reasonable economic model provided, and that users would not attempt to induce games in the system or otherwise manipulate it.

Given these issues, we concluded that it made more sense to modify an existing type of auction to suit our needs, rather than trying to faithfully reproduce the exact parameters required for a formal implementation of a given type of auction. By doing this, however, we incurred the following disadvantages:

- All economic results from the auction type adapted for use were no longer directly applicable. In particular, the results from the literature regarding the Vickrey auction's optimality and game-free status [29] could no longer be applied.
- Rather than being able to demonstrate that our economic model was fair in the strictest mathematical sense, we had to rely upon users' judgment and perceptions of "fairness".

In the end, it seems that the choice to rely upon a pragmatic, rather than a pure, economic model was a successful one; although we cannot know for certain, users did not seem to try to game the system, nor did they appear to have any doubt in the system's overall fairness. (Indeed, section 7.1.2 presents data that suggests that once the system was in place, users were so careful about submitting jobs that they had no motivation to try to game the system — it was essentially never overloaded.) The economic model chosen appeared fair and simple enough to them, and thus was accepted. It remains an open question, however, whether a system with greater use and much greater contention would continue to support this decision.

4.5 Vickrey Auctions

We chose the Vickrey auction — a sealed-bid, second-price, ascending single auction — as the model for our economic system. We consider now some of the reasons that choice was made, as well as the advantages and disadvantages of that choice.

4.5.1 Previous Results

A major reason for choosing the Vickrey auction as our economic model was the extensive research and analysis into this algorithm that already existed in the literature. The Vickrey auction is provably optimal [29]: the bidder who truly values the goods under auction the most will win the auction, and the seller is certain to obtain the highest possible price for those goods. The Vickrey auction is also provably game-free [29], meaning that users are unable to manipulate the auction in order to gain an unfair advantage over other users. These characteristics meant that the Vickrey auction was likely to both converge on an optimum price for resources rapidly, and prevent users from attempting to manipulate the system to their own advantage. While these characteristics could no longer be *proved* due to our emphasis on pragmatism over purity of economic theory, starting our model with an auction that has these characteristics was nevertheless desired.

We also chose the Vickrey auction because it is extremely well understood. Above and beyond the theoretical literature on the subject, the Vickrey auction model is used for significant auctions in the real world — including the issuance of debt by the United States government [30, 31], an auction market virtually sure to invite manipulation if at all possible (and which did, in fact, experience manipulation prior to its switch [32]). Certain initial public offerings of companies are now issued using the Vickrey auction, too, and even eBay [33], a popular online auction site, uses an auction model nearly identical to the Vickrey auction to allow greater efficiency for its users.

4.5.2 Motivation

A more direct set of motivations for our use of the Vickrey auction consists of its sheer simplicity. The Vickrey auction requires very low user interaction when compared to other types of auction; users ideally will set their bids, once, at the true value that the resources have to them, and then leave their bids untouched. Further, the Vickrey auction is a conceptually simple model, something of paramount importance when introducing economic resource allocation to an environment not used to such policies. Finally, the growing popularity of auctions in the real world — witness the enormous number of visitors to eBay each day — and their shift to more-efficient auction models convinced us that modeling our system on the Vickrey auction was a wise choice.

4.5.3 Tradeoffs

However, no choice of auction model was without its disadvantages, including the Vickrey auction. The major tradeoff we made in choosing this model was one of sheer system transparency. Because the Vickrey auction requires, of necessity, that users' bids are kept secret, users can no longer see the entire state of the auction system, and thus must simply trust that the system is performing as specified; as well, if users find that they have lost an auction (and thus find their highly-valued jobs sitting in a queue rather than running), users have little recourse other than to try adjusting their bids essentially at random until some change in the status of their jobs results.

Unfortunately, there was little compromise that could be made on this point: if a Vickrey auction is made open-bid, users can manipulate the system by driving up their own bids to be equal to the highest bid minus the minimum bid increment, thus ensuring that the highest-bidding user always pays, essentially, his or her actual bid. This, in turn, causes users to cease bidding their true value for resources and instead causes them to bid only what they think other users will bid, plus some minimum increment; the entire auction degenerates into a traditional English auction, with associated problems [34].

5 Implementing an Economy with PBS

5.1 Background

Once a proper economic model was chosen, a full-scale implementation of that model was the next obvious step. Here we faced a decision: create a new batch-scheduling system from scratch, using it to implement our economic policy, or try to retrofit an economic scheduling policy into an existing batch-queuing system? Both posed problems: implementing a system from scratch would take a substantial effort and yield a result not likely to be as robust as systems that had been in existence for years, but retrofitting an economic scheduling policy into an existing system could prove very difficult and introduce bugs, since no widely-used existing queuing systems were designed with economic scheduling policies as a goal.

Indeed, most existing queuing systems failed on this second point: most systems were designed to use some sort of traditional first-in-first-out (FIFO) or priority algorithm. In some systems [35, 36], such a scheduling algorithm was so completely ingrained in the design of the system as a whole that it would be nearly impossible to retrofit an economic scheduling algorithm; in others, the scheduling algorithm was more versatile — but the interfaces and data structures required to implement a new scheduling algorithm were undocumented, esoteric, or both.

One system, however, provided an environment expressly designed to accommodate any sort of scheduler, and provided enough flexibility to accommodate a scheduling algorithm unlike those imagined when the system was designed. The Portable Batch System (PBS) [37] is a second-generation batch queuing system designed to operate in a wide variety of environments, scalable from a single time-shared system up to groups of large dedicated parallel supercomputers and clusters.

5.1.1 About PBS

PBS is a relatively recent batch-queuing environment that is intended to be portable to a wide number of platforms, robust, and exceedingly flexible. It has ancestral roots in NQS (the New Queueing System [35]) and other similar systems, but was created from scratch to fulfill the requirements of modern queuing environments. The architecture of PBS is oriented at once both toward more traditional large-scale computing facilities, such as IBM's SP2 series [38] and SGI's Origin 2000 series [39], *and* toward the coming generation of clustered workstations or PCs, such as Linux Beowulf clusters [40]. This dual orientation forces PBS to be very flexible in its architecture and be forgiving of use in new or untested environments; this, thus, was an ideal candidate for the economic queuing system.

It became clear that PBS would be a useful, stable platform for experiments in economic scheduling for several reasons. First, it supports — with substantial flexibility — the traditional model of a batch queue; queues can be created, deleted, enabled or disabled, and prioritized at will. Second, it is growing rapidly in popularity among high-performance computing (HPC) installations and thus is well-supported and maintained. Finally, and most importantly, it was designed as a second-generation queuing system: after it became apparent that queuing environments tended to vary a great deal from site to site and machine to machine, PBS was architected with flexibility as a primary goal. It thus supports, with ease, internal modification to support new environments, systems, and scheduling algorithms. Indeed, by simply replacing a single process in the system, it was possible to provide the PBS system with an economic scheduler that worked robustly and scaled to many thousands of jobs.

5.1.2 Structure

The architecture of PBS can be seen either from a user's point of view or a developer's. We consider first the user's perspective on a working PBS system, showing the various components with which he or she may interact and how they respond to queries, commands, and so forth; next, we consider the developer's perspective on the same system, viewing the same transactions internally. Finally, we discuss the component structure of the system, discussing the individual daemons and processes required.

5.1.2.1 User Perspective

From a user's point of view, the PBS system is fairly simple and quite straightforward. Figure 5.1-A shows the overall structure of the PBS system as seen by a user who runs a job; we trace the path of the job through its course, starting at the upper left of the figure and moving in a counterclockwise direction.

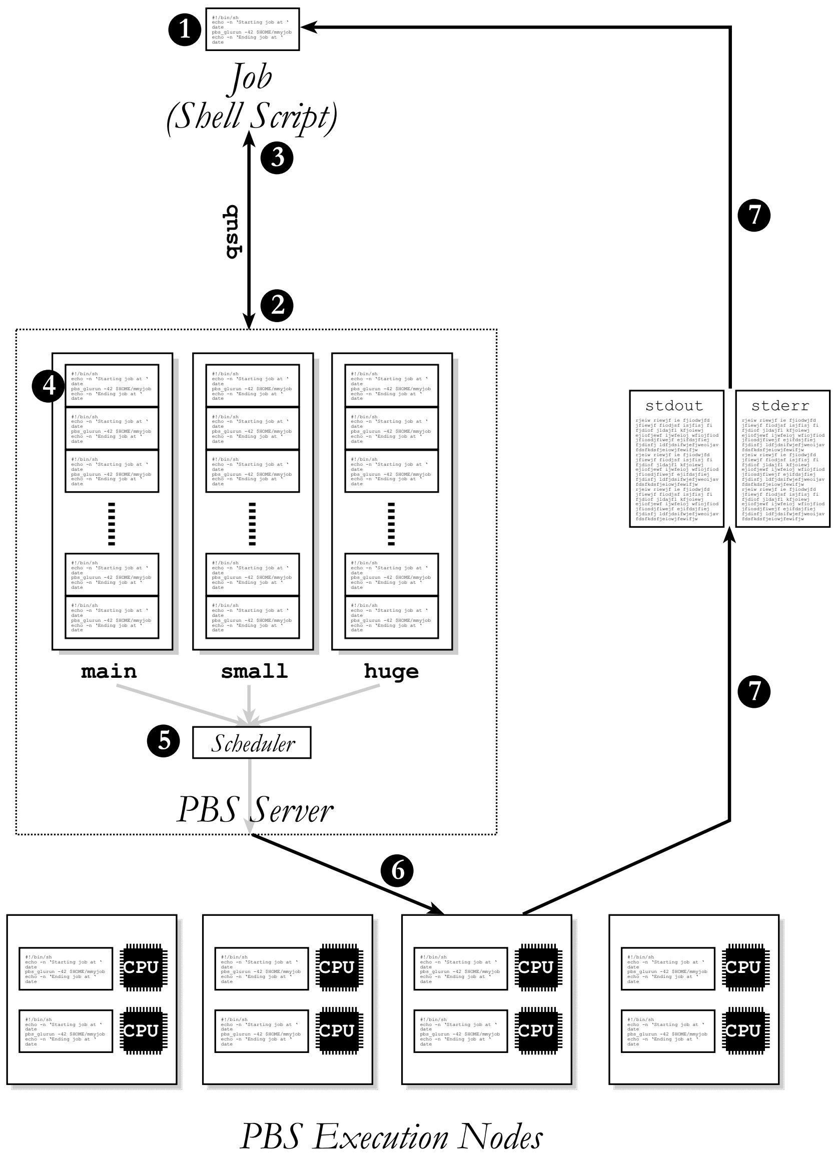


Figure 5.1-A. Overview of PBS from a user's perspective.

1. A PBS job is simply a shell script, prepared by a user. Obviously, this shell script can be simple or arbitrarily complex. The user creates such a shell script and stores it in a file accessible to the PBS system — for example, `/home/jsmith/myjob`.

2. Once this file has been stored, the user submits the job to the system using the `qsub` command:

```
% qsub $HOME/myjob.sh
142.u.CS.Berkeley.EDU
%
```

The job is accepted by the PBS server, which, to the user, is simply an entity that "makes the system work".

3. The user is returned a job ID, an opaque handle with which the user can retrieve the status of the job or modify it at any point in the future.

4. The server now adds the job to one of its queues, which are created by the PBS administrator. By default the job is added to the `main` queue, but the user can specify a specific queue as an option to `qsub`. The meaning of various queues is defined entirely by the scheduling algorithm; with our economic scheduler, there is only one queue, `main`.

5. The server now runs the user's job when the scheduling algorithm indicates it should. The user can query the server for the status of his or her job at any time (using `qstat`), and watch it move through the queuing system.

6. At some point in the future, as determined by the scheduling algorithm, the user's job will be scheduled on one or more (depending on its degree of parallelism, as specified on the `qsub` command line) *execution nodes*. The user is not notified of this fact other than by a notation in the output of a status command to the system.

7. Eventually, the user's job will complete execution. When this occurs, the standard output and error streams of the job — which have been collected from the running job by PBS — are copied back to the original directory of the job (in this case, as `myjob.o142` and `myjob.e142`). At this point, the cycle is complete; the user's job has executed and produced output. Of course, the user may have as many jobs as desired in the system at any given time.

5.1.2.2 Developer's Perspective

We turn, now, to a view of the same process, as seen from the point of view internal to PBS — the view experienced by a PBS developer. Figure 5.1-B shows this view of the system. We proceed from top to bottom through the diagram, indicating the various components of PBS and how they interact to properly execute a user's job.

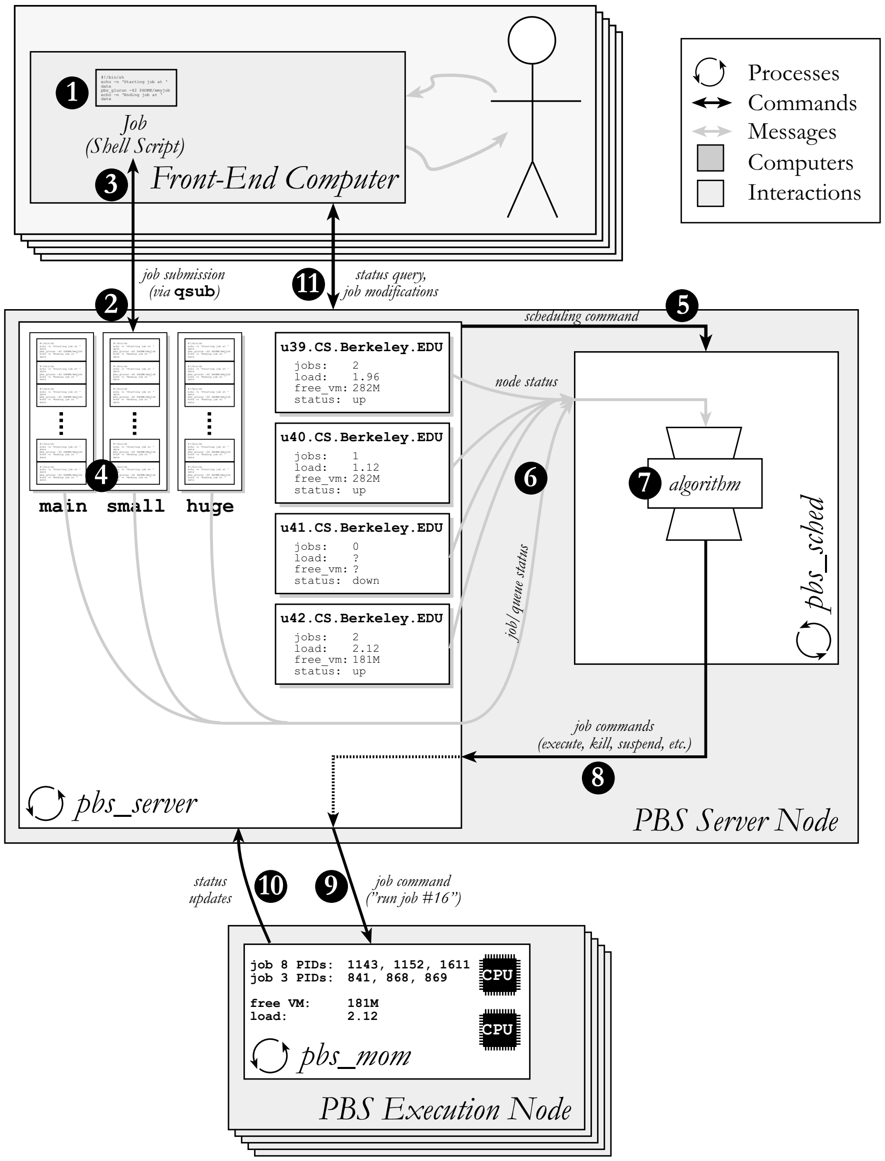


Figure 5.1-B. Overview of PBS from a developer's perspective.

1. The entire process is begun by a user who executes the `qsub` command on a front-end computer, specifying the shell script that the user wishes run and any of a fairly wide variety of options (most of which do not alter the resulting internal processes).

2. `qsub` performs some basic error-checking, and then opens a connection to the PBS server. It transmits the location of the shell script (if a distributed file system is present) or the script itself (if not) and any of the various options the user has passed to the server, which stores the data in its internal database of jobs.

3. The server returns a *job identifier* to the `qsub` process, which prints it on the screen

for the user's reference. This unique identifier, which can also be retrieved later using a "list all jobs" command (`qstat`), allows the user to manipulate and query the status of the job later.

4. The server adds the job to one of its internal queues. The server can maintain an arbitrary number of queues, each with limits on execution time, job size (parallel degree), *etc.*; in the economic system, only one queue is used. Queues are only used in scheduling decisions to the degree determined by the scheduling code itself.
5. Whenever the state of the system is perturbed — such as an added or completed job, a user's modification of a job, or so forth — and on a regular basis, the server directs the scheduler (a separate process) to begin a new scheduling run. The nature of the cause of the scheduling run is transmitted to the scheduler, although typically this is inconsequential.
6. The scheduler, in turn, queries the server concerning the current state of the system: it reads the status of all jobs and all nodes from the server.
7. The scheduling algorithm uses this information to make decisions about what actions are necessary on the system. The scheduler can decide to terminate, suspend, resume, enqueue, dequeue, kill, or run any or all of the jobs in the system.
8. The scheduler issues the required commands to the server.
9. The server receives the commands from the scheduler and forwards them to the appropriate execution nodes. These nodes, which maintain a mapping from jobs to operating system resources, make the required requests of the underlying operating system (primarily, starting or terminating processes) and reply with confirmation to the server.
10. As the job executes, updates of its status (runtime, resources consumed, *etc.*) are returned to the server.
11. The user can query the server at any time for status updates on the job, as well as to manipulate the job as he/she sees fit.

5.1.2.3 System Components

We next consider the individual processes that compose the entire PBS system and describe their actions.

5.1.2.3.1 Server

The PBS *server*, implemented by the process `pbs_server`, essentially acts as a persistent database of jobs, nodes, and configuration information: when users submit jobs to the system, they are stored in the server, and the current status of all nodes is also stored in the server. The server is responsible for storing all state associated with a job, including its status (queued, suspended, running — for how long, where, and so forth), and for caching the latest status update received from each node. It communicates with the node execution daemons (`pbs_mom`) to actually execute a job.

(PBS uses the term "MOM" to stand for *m*achine-*o*riented *m*iniserver; we call the reader's attention to the distinction between this and the more-common use of "MOM" to stand for "message-oriented middleware", which is an entirely different concept.)

Of particular note is that `pbs_server` does *not* make any scheduling policy decisions itself: the server will never execute a job without being specifically told to by an administrator or the scheduler. The server is merely a repository for all the state of the system; it does not by itself initiate any actions other than to gather and store status.

5.1.2.3.2 Machine-Oriented Miniserver (MOM)

Each node of a PBS system that is to be used for execution runs the *m*achine-*o*riented *m*iniserver (`pbs_mom`), a daemon that runs on each node in the cluster and provides the mechanisms by which the PBS system actually processes work and manages individual nodes' resources. It is responsible for accepting two general categories of requests: status requests (load average, free memory, job count, *etc.*) from any client so authorized (though typically this is the `pbs_server`), and execution requests (run job, kill job, suspend job, *etc.*) from the server.

The PBS "MOM" is thus the point of direct contact between PBS and the underlying operating system itself: it manages resources on the system, starting, stopping, and maintaining the status of the various jobs that are executing under the node's native operating system. It is the most machine-specific component of PBS — but, typically, also the simplest.

5.1.2.3.3 Scheduler

Of the various PBS components, however, the scheduler — `pbs_sched` — is the most significant and the most powerful. Part of PBS's flexibility is that essentially all policy decisions are concentrated here, in the scheduler, which interacts with the rest of the system via a very well-defined and powerful interface.

The PBS scheduler, in fact, interacts with the rest of the system simply as yet another client: it uses the same fully-documented native library interface (`libpbs`) as all user-space programs (including the supplied programs such as `qsub`, `qstat`, *etc.*). The scheduler (by nature of its connecting host and port) is granted full privileges to act on any job or node in the PBS system — but, aside from that, is no different from any other client of the `pbs_server`, except that it receives a notification signal whenever a significant event, such as a new job submission, occurs on the server.

By using this status as a "privileged client", the PBS scheduler can efficiently and directly impose any desired scheduling policy upon the system. Rather than a more traditional model of coercing the scheduler into one of a few traditional modes (FIFO, priority-based, shortest-job-first, *etc.*), the scheduler is welcome to use any information it has access to, whether internal to PBS or external, in making its job-scheduling decisions. This format also isolates the scheduler from any internal server data structures and entirely isolates it from interaction with the execution nodes; this isolation makes the scheduler more portable, flexible, and much easier to write and debug. Unfortunately, this architecture of the scheduler as a "privileged client" also means that the PBS scheduler is something of a single point of failure for the system: if the scheduler fails, no further jobs will be started by the system; while the server will finish existing jobs, accept new submissions, allow removal of old submissions, and so forth, no new jobs will ever be scheduled.

A typical PBS scheduler remains in an infinite loop:. First, it listens for notification of some new event from the PBS server. Upon receipt of this notification, it queries the server for all the information it will need to make scheduling decisions and stores this information locally. Next, it processes this information via some scheduling algorithm, determining the actions that need to be executed on the system due to the scheduling policy. Finally, the scheduler issues commands to the server to cause these actions to actually be carried out.

The great flexibility provided by the scheduler is the single most attractive feature of PBS for our work. It is exceedingly powerful and allows great flexibility in the implementation of an economic scheduler; further, due to the scheduler's separation from the server, a bug in the scheduler typically means only that new jobs in the system will not begin running — existing jobs will continue running happily, and users can still submit new jobs to the system.

There is a slight disadvantage to the way PBS manages its scheduler: because schedulers are so powerful, they are also complicated. The scheduler must handle all possible cases in the system, and must handle them gracefully; this means that schedulers can be difficult to write properly and debug in a production system. Put another way, the power of being able to redefine absolutely the scheduling policy of a PBS system comes at the price of requiring a substantial amount of careful coding before the scheduler will be truly robust.

5.1.2.4 Parallel Execution

Finally, we consider the mechanism by which PBS handles the notion of a parallel program and arranges for it to be executed on various nodes. PBS understands, via the `nodect` parameter passed upon submission of a job, the parallel degree of a submission. This information is retained by the server, and is used by the scheduler to make decisions about how many and which nodes to allocate to the job. The supplied PBS schedulers create a *node file* containing the list of nodes allocated to a job by the scheduler when that job is run; user-provided schedulers may, of course, use any mechanism they see fit.

However, when a job is spawned under PBS, *only one node* begins running the process as a direct result of the PBS run directive, no matter how many nodes are requested in the job submission. Responsibility for actually spawning the parallel tasks across all nodes allocated to the job (and determining which nodes are allocated to that job, by use of a node file or other mechanism) is entirely the responsibility of the job itself. That is, if a job requests sixteen nodes, it is allocated sixteen nodes by the scheduler; however, it is only spawned on *one* of those nodes, and the job itself must arrange to be run across all sixteen nodes — typically via a runtime parallel-execution mechanism, such as `mpirun` or GLUnix.

This particular mechanism provides some flexibility to the submitted job; set-up commands can precede the call to the parallel-execution mechanism in the supplied script, thus allowing for sequential set-up and tear-down tasks to run before and after the actual parallel code. However, as is expressed later, this mechanism can also produce difficulties: because PBS does not spawn the parallel processes itself, it cannot manipulate them either. (Notably, as of this writing, a lively discussion was ongoing on the `pbs-users` mailing list concerning ways of integrating this manipulation into PBS.)

5.2 The Economic Scheduler

Once PBS was chosen for its flexibility and power in implementing a fairly radically new type of scheduler — an economic scheduler — implementation of the scheduler itself began. Based on early investigations, the scheduler was implemented in the Java programming language [41] to lend it flexibility and robustness; the Java Native Interface (JNI) [42] and Java Database Connectivity (JDBC) [43] APIs provided connections between the scheduler itself and, respectively, the external PBS system and databases used for logging and accounting. Here we explore the structure of the scheduler a bit more deeply and explain the ways in which it connects to the rest of the system.

5.2.1 Overview

PBS natively provides the ability to implement a scheduler in the C, Tcl, or BASL programming languages. (BASL is a custom language unique to PBS intended to make scheduler creation simple.) Each has its weaknesses: Tcl and BASL were, generally speaking, not structured enough to produce a relatively complex scheduler, and their lack of easy connectivity to databases (for logging users' actions and maintaining bank accounts) provided another strike against them. C, meanwhile, was certainly sufficiently powerful, but it was obvious (after study of several PBS-provided C schedulers) that creating a native C scheduler would take a great deal of groundwork before the important parts of the scheduler (the algorithm itself) could be created — and, more importantly, that flexibility would be lost once the scheduler was created.

5.2.1.1 JNI

Because of these considerations, and prior positive experience, the Java language was chosen to implement the economic scheduler. This created a difficulty, however: a native C PBS scheduler uses a library, called `pbs_ifl`, to interact with the server; this library provides calls that, in turn, take care of all handling of network messages to and from the server. Rather than re-implement `pbs_ifl` in Java, the Java Native Interface (JNI) was used to build a "bridge" from Java code, executing in a Java Virtual Machine (JVM), to the underlying `pbs_ifl` C library.

This strategy rapidly proved itself to be a good choice. The `pbs_ifl` library, like most of PBS, has an implicit object-oriented structure, and modeling this structure in Java objects (whose methods called through to `pbs_ifl` using JNI) was quick and straightforward. Once this work was done, the resulting Java PBS interface made the JNI and `pbs_ifl` connection invisible from the main program: all the benefits of using Java for the scheduler were realized, from much faster development to easier debugging to much easier modification of the scheduler, without creating contortions in the program code just to interface with PBS.

5.2.1.2 JDBC

The use of Java provided another significant benefit to the scheduler. Collection of all data pertinent to users' job submissions, when those jobs ran, and when they terminated (as well as a number of other factors) was required for later analysis; further, a persistent, reliable banking service with full accounting trails was necessary to ensure that users would trust the system. By using the Java Database Connectivity (JDBC) API, the scheduler rapidly gained connection to a MySQL [44] database running on a remote node that provided both these facilities with the reliability and performance of a full relational database.

5.2.1.3 Performance and Reliability Concerns

Unfortunately, none of JDBC, JNI, or MySQL are typically known for being exceptionally fast layers of software; further, MySQL, while a fairly high-performance database given its status as free software, contains no support for transactions and is not known for being exceptionally robust. These concerns would be of increased (even paramount) importance in a high-availability, performance-oriented scheduling environment; however, in our environment, these choices proved effective. Actual runtime of the scheduler, while sometimes as long as thirty seconds, was entirely dominated by calls to the various PBS MOM daemons on individual runtime nodes; JDBC and JNI calls never appeared as significant bottlenecks.

Further, MySQL's ease of installation, use, and cost (zero) were enormous advantages, outweighing the risk we ran of losing some data. Again, however, our cluster's status as a research resource allowed us to take the risk of losing some users' jobs or account information; this risk may not be acceptable in a production clustering environment. Especially evident is the risk we ran of billing a user twice for the same job and/or never billing a user for a job; while these risks were minimized by careful coding practices, such "guarantees" of safety are hardly tolerated by the commercial banking industry, and certainly would not be tolerated in many production economic systems. Instead, full production commercial RDBMS systems, message-oriented middleware, and transaction processing systems would be required for a high-performance, high-reliability, robust implementation.

5.2.2 Structure

Figure 5.2-A presents a block overview of the entire PBS economic scheduler, as implemented with Java. Each component of this picture is now described, and its relationship to the rest of the economic scheduler is discussed.

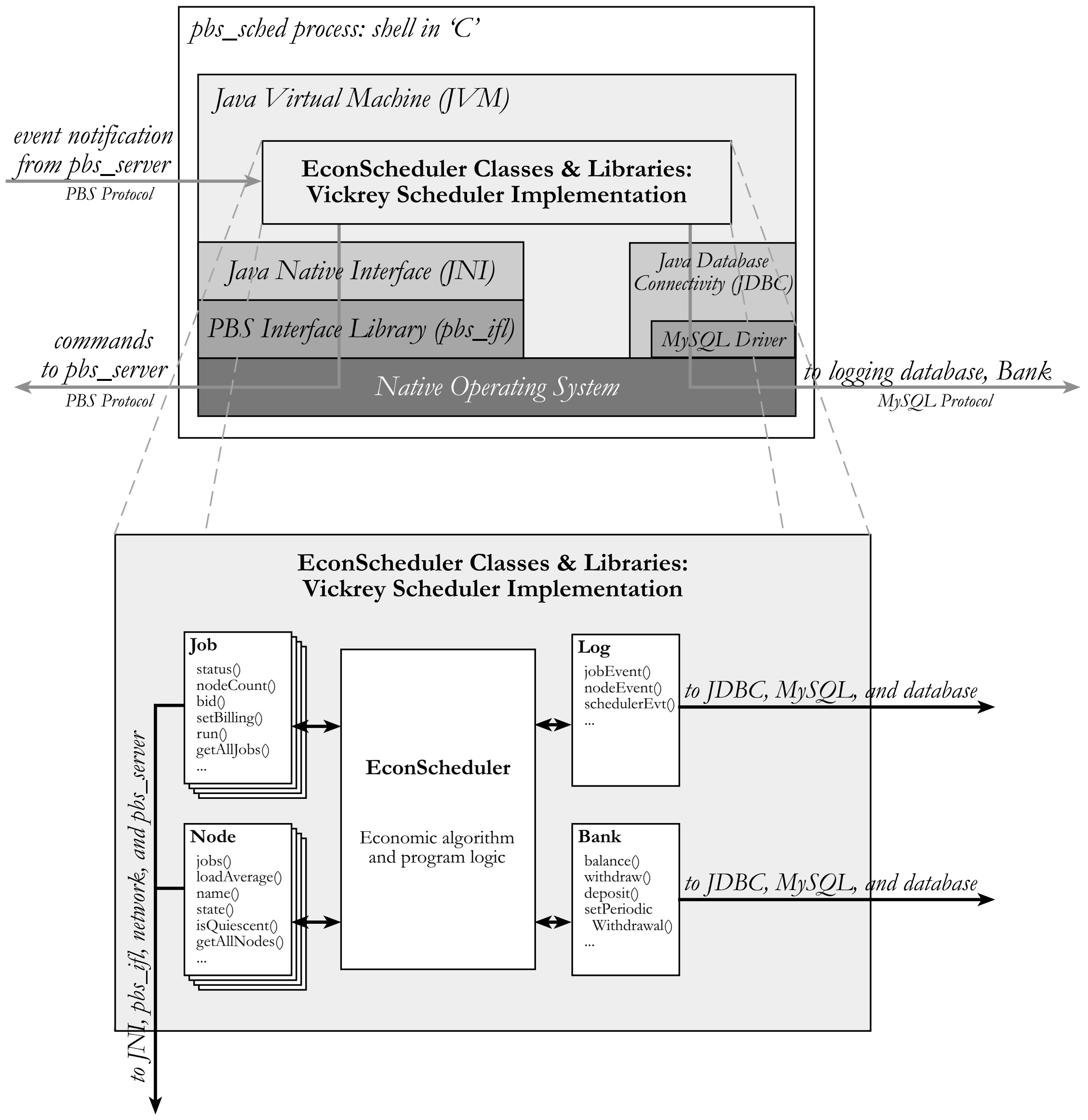

pbs_sched process: shell in 'C'
Java Virtual Machine (JVM)
event notification from pbs_server
PBS Protocol
EconScheduler Classes & Libraries: Vickrey Scheduler Implementation
Java Native Interface (JNI)
Java Database Connectivity (JDBC)
PBS Interface Library (pbs_ifl)
MySQL Driver
commands to pbs_server
PBS Protocol
Native Operating System
to logging database, Bank
MySQL Protocol
EconScheduler Classes & Libraries: Vickrey Scheduler Implementation
Job
status()
nodeCount()
bid()
setBilling()
run()
getAllJobs()
...
EconScheduler
Economic algorithm and program logic
Log
jobEvent()
nodeEvent()
schedulerEvt()
...
to JDBC, MySQL, and database
Node
jobs()
loadAverage()
name()
state()
isQuiescent()
getAllNodes()
...
Bank
balance()
withdraw()
deposit()
setPeriodic Withdrawal()
...
to JDBC, MySQL, and database
to JNI, pbs_ifl, network, and pbs_server

Figure 5.2-A. Internal structure of the PBS Economic Scheduler.

5.2.2.1 Wrapper process

The entire PBS economic scheduler sits inside the "shell" of a C scheduler skeleton, provided with PBS. This skeleton is primarily responsible for setting up the scheduler's network listener (to receive notification of events from the server), creating the security token used to communicate with the server, and then creating the Java Virtual Machine (JVM) that is used to house the bulk of the scheduling code. When an inbound event notification is received from the server, this "shell" calls into the JVM (which is persistent from event to event, allowing global statistics and persistent database connections) to notify the Java program of the event.

5.2.2.2 Economic Scheduler Classes

The economic scheduler itself is composed of a number of Java classes, which are loaded automatically into the JVM upon demand and started by the C "shell". The bottom portion of Figure 5.2-A provides an overview of this set of classes. Essentially, Job and Node objects are used to model each job and node in the system, caching information about themselves and updating their state to always be consistent with that of the server itself, so they can provide the scheduling code with accurate information. These objects also provide manipulation methods by which the economic algorithm code itself can run, suspend, kill, and otherwise manipulate jobs and nodes in the system.

The scheduler code also contains Java classes that support the banking and logging functions of the scheduler. The banking and logging classes encapsulate access to the underlying databases, allowing the economic scheduler to simply make appropriate calls to them when necessary; the classes take care of establishing the required JDBC connections to the underlying databases and executing the proper SQL commands. JDBC itself is responsible for actually creating and sending the proper network streams to carry out the SQL commands themselves.

5.2.2.3 JNI

The Java Native Interface, JNI — shown in the upper portion of Figure 5.2-A — provides the economic scheduler with its connection to the PBS interface library, `pbs_ifl`. While the Java Native Interface provides the APIs necessary to *enable* a Java program to call a C library (and vice versa), a set of JNI "glue" routines had to be written to actually enable the particular functions required. Again, because of the implicit object-oriented structure of

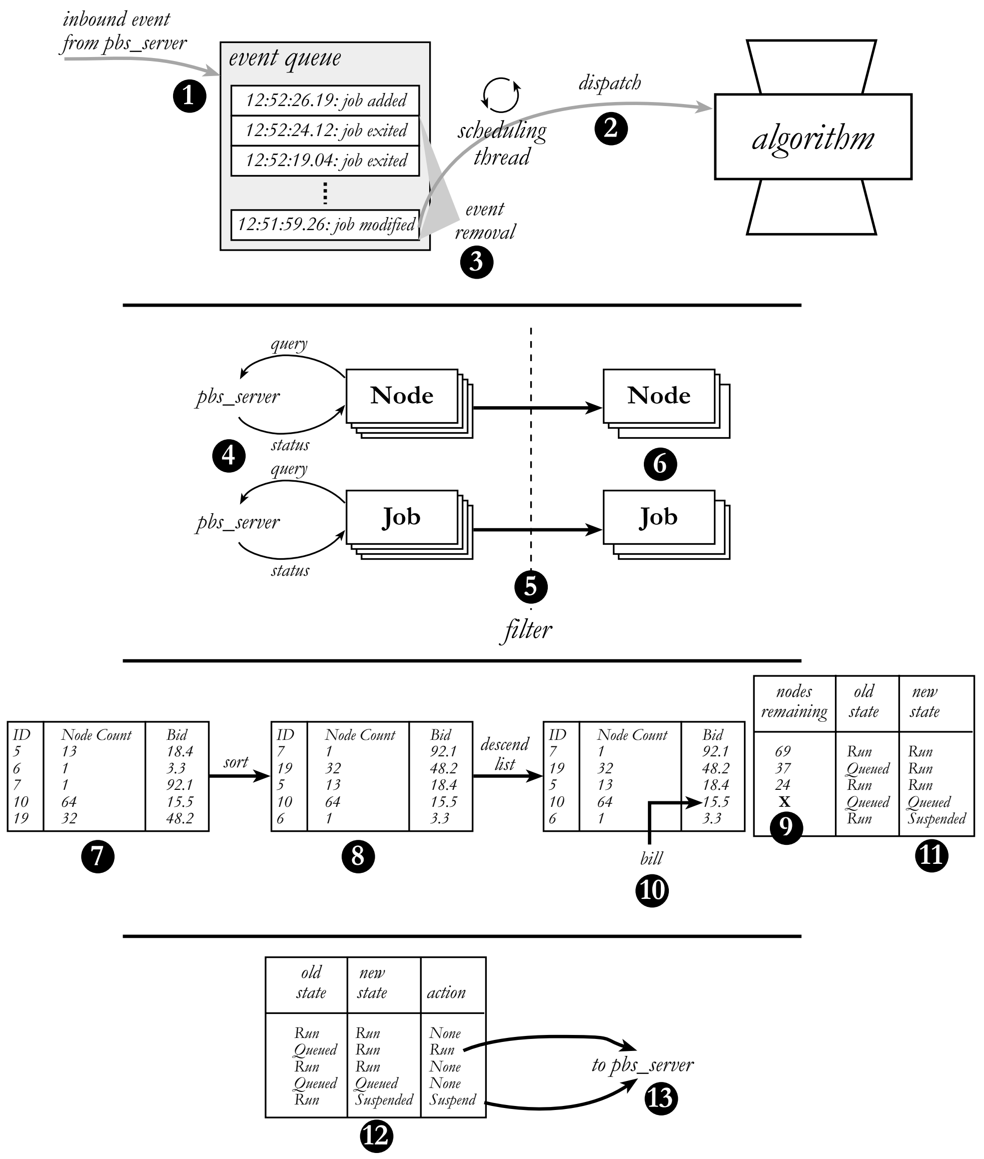
PBS, this proved to be straightforward: classes such as Server, Connection, Job, and Node (these last two distinct from the internal *scheduler* classes of the same name) provided direct manipulation of the underlying PBS structures and network communication transparently to the calling application via `pbs_ifl`.

5.2.2.4 JDBC

Similarly, Java Database Connectivity, JDBC — also shown in the upper portion of Figure 5.2-A — provides the economic scheduler with its connection to the MySQL databases used to track user activity and provide a robust banking service to the application. Because of the driver-oriented design of JDBC, the economic scheduler itself didn't need to concern itself with the particular type of RDBMS in use nor (other than at configuration time) the host or port on which it was running, nor any other details of the actual database; the MySQL driver for JDBC provided service directly to the scheduler. Bank and Log classes were created within the scheduler to encapsulate these database functions further, making database access completely transparent to the scheduler itself.

5.2.3 Scheduling Algorithm

Now that the structure of the economic scheduler has been detailed, the scheduling algorithm itself becomes of primary interest. Figure 5.2-B describes the implementation of the scheduling algorithm itself.



ID	Node Count	Bid
5	13	18.4
6	1	3.3
7	1	92.1
10	64	15.5
19	32	48.2

ID	Node Count	Bid
7	1	92.1
19	32	48.2
5	13	18.4
10	64	15.5
6	1	3.3

ID	Node Count	Bid
7	1	92.1
19	32	48.2
5	13	18.4
10	64	15.5
6	1	3.3

nodes remaining	old state	new state
69	Run	Run
37	Queued	Run
24	Run	Run
X	Queued	Queued
	Run	Suspended

old state	new state	action
Run	Run	None
Queued	Run	Run
Run	Run	None
Queued	Queued	None
Run	Suspended	Suspend

Figure 5.2-B. Economic scheduling algorithm.

5.2.3.1 Event Queuing

The entire scheduling algorithm is kicked off when an inbound event arrives from the PBS server. The C "shell" process delivers this event to the scheduling Java code, which timestamps the event and delivers the resulting bundle into a queue (1). If the scheduling algorithm is not currently running, a scheduling thread dispatches this event to the scheduling algorithm itself; if not, the thread blocks until the scheduling algorithm has completed, and then delivers the event (2). Just before delivery of the event, the scheduling thread notes the current time and removes all events in the queue with a timestamp less than the current time (3).

This method of event delivery, rather than direct invocation of the scheduling algorithm at each event, was chosen for a specific reason: during periods of heavy demand on the system, events may arrive as rapidly as several a second. Because the scheduling algorithm can itself take several seconds to execute (due to its need to communicate with remote daemons, which must in turn start processes on various execution nodes), a direct-execution strategy would either result in multiple copies of the scheduler running (inviting all manner of race conditions) or result in events being delivered long after they were originally generated.

Because the actual *content* of events is irrelevant to our scheduler (it is concerned only *that* something changed, not *what* changed), this strategy greatly increases the scalability of the scheduler. As long as it has been *started* at least once since the generation of an event, the particular action that generated that event will have been seen by the scheduler as already having occurred, and therefore the scheduler need not be run again.

5.2.3.2 Obtaining Status Information

Once the scheduling algorithm itself has been invoked, it immediately queries the server to update its internal view of the PBS "world". The economic scheduler directs all Node and Job objects to synchronize themselves with the actual state maintained in the server (4); also, new Node and Job objects are created to represent any new nodes or jobs that have been created in the server, and any Node or Job objects that represent nodes or jobs that no longer exist are destroyed.

Various conditions may cause a node or job to be *ineligible* for use in the PBS system: a node may be crashed or reserved, a job may be submitted by a user who is not authorized to use the system or a job may request more nodes than are available in the entire system, and so forth. The scheduler thus next sifts through all Node and Job objects in the system (5), retaining only those Nodes and Jobs that are actually eligible to participate in the scheduling algorithm (6). If a serious error occurs (such as the server crashing or all nodes failing), the scheduling process can be aborted here.

5.2.3.3 State Strategy

At this point, two design paths presented themselves: the scheduler could either attempt to respond directly to whatever event caused this particular scheduling cycle, or it could examine the entire system, note any discrepancies between the current system state and its scheduling policy (such as a job, likely newly created, that had a high-enough bid to run but was not yet running), and correct those discrepancies. Both alternatives presented unique advantages and disadvantages.

By simply responding to the event which triggered the scheduler, the scheduler could be made to run more rapidly than otherwise and could probably be simplified into a smaller number of possible code paths, a distinct advantage given the potential complexity of a scheduler. However, the disadvantage of this method — and, indeed, the key reason that it was not selected — is that it requires that the scheduler never miss an event: if it misses an event and therefore fails to take action on some critical action, the effects of this error may continue for quite some time. Given the experimental nature of the system and the potential for failure in an untested system, this was deemed to be a poor choice.

Rather, the scheduler proceeds using the alternative: it reads the current state of the system (via the newly-updated Job and Node objects), then computes the new, desired state of the system (by implementing the Vickrey auction algorithm). It then compares old to new, executing the fewest possible commands to alter the current state of the system to correspond with the new, "desired" state.

5.2.3.4 Vickrey Auction

Thus, once the scheduler has collected data on the current state of the system, it proceeds through a straightforward implementation of the Vickrey auction. Initially, it simply collects data of all jobs in to an array (7). The algorithm then proceeds to sort this array by descending bid, ranking the highest-bidding jobs at the top (8). (Because bids are specified as "per node, per minute", the ranking is correct: nodes that request more nodes or that run for more time are neither penalized nor given any advantage.)

The algorithm next takes note of the total number of jobs that are available in the system, whether currently running jobs or not. It descends the now-sorted list of jobs, subtracting each job's node count from this total number of jobs as long as sufficient nodes remain for each job. When a job is found that requests more nodes than remain, the processing stops (9). The bid of this job will be the price billed to all jobs that are running when this cycle completes (10).

The algorithm now knows which jobs should be running and which should not: those jobs that were encountered while there were sufficient nodes remaining must run, since they all have higher bids than any of the jobs that were not yet encountered — and those jobs must not run. The algorithm therefore constructs a list of "new states" of each job: jobs that will run will be in the *Running* state, while jobs that will not run will either remain in the *Queued* state (if they were previously not running) or will be placed into the *Suspended* state (if they were previously running). Free nodes are allocated for jobs that will run for the first time.

5.2.3.5 All Done

At this point, the scheduler has finished its cycle. It writes log information about its actions to the logging database (via the Log object and JDBC), and terminates its cycle. Upon receipt of the next event via the network, the C "shell" will re-activate the scheduler, and the process will begin all over again.

5.2.4 Challenges

5.2.4.1 Communication with GLUnix (signaling issues)

Several challenges arose during the construction of the economic scheduler for PBS; the significant challenges had to do with integration of a job suspend/resume facility, both into PBS itself and into the existing GLUnix parallel runtime layer — neither of which were designed with use of the generic UNIX suspend/resume facility (`SIGSTOP`/`SIGCONT`) in mind. We now consider these challenges and the solutions that were found to them.

5.2.4.2 Suspend/resume

As previously discussed, a provision for suspending and resuming parallel (and sequential) jobs is essential to building any truly non-FIFO scheduler for a parallel batch system: without such a facility, long-running jobs rule the cluster — once they begin, they must terminate of their own will before new jobs may replace them, no matter how high the priority of the new jobs.

PBS does include a "suspended" state for jobs, and `pbs_ifl` contains calls to suspend and resume jobs that are currently active. However, this facility was originally designed only for use on platforms that natively support (*i.e.*, support in the underlying operating system) suspension and resumption of executing processes. (As of this writing, this is restricted to Cray's Unicos operating system.)

Because such a facility was required for correct implementation of a Vickrey scheduler, the PBS system was extended to provide a simple implementation of this facility on all UNIX platforms. When a job must be suspended, it is sent the UNIX signal `SIGSTOP`. This signal, which cannot be caught (and thus modified or ignored), immediately suspends the targeted process until it receives the corresponding signal `SIGCONT`. While this mechanism is not ideal — for example, the process only ceases executing; it still uses virtual memory resources, open file descriptors, and so forth — it does provide the level of control requisite for implementation of our particular scheduler.

5.2.4.2.1 Runtime Systems

That said, such a mechanism still poses a few problems to the underlying mechanisms of execution. First, and foremost, most parallel software environments do *not* expect suspension and resumption to be an ordinary part of execution. While, in a sense, suspension and resumption is essentially just a very long context switch, it can still cause significant problems due to these various software environments.

When suspended (and later resumed), software built using these environments can react in a number of ways, from behaving properly to failing outright. For example, software communicating using standard TCP/IP connections typically behaves well: the operating system, being aware of the suspension and resumption, ensures that proper packets are sent and thus prevents the communication channel from failing. In sharp contrast to this, however, are any implementations of parallel protocols that expect the underlying communications network to be reliable or have implicit dependencies on real-time timers: these protocol implementations will fail when a source or destination process is suspended (and thus fails to respond to or initiate messages).

This sort of failure demonstrates a more fundamental design issue in these systems: unless all components of the system — applications, libraries, runtime environments, and the operating system itself — are designed to accommodate preemption (whether by suspension or true checkpoint) of parallel programs, no truly preemptive scheduling policies are possible. While the system presented here contained enough software that accepted suspension and resumption of programs (whether fortuitously or by design) to remain an attractive parallel-programming environment, this factor can be of serious concern when developing future systems with novel scheduling policies.

5.2.4.2.2 Signalling via GLUnix

The second challenge presented by this *ad hoc* method of suspending and resuming user processes is due to the structure of the entire runtime system. Because PBS does not itself spawn the parallel tasks, it *cannot* take responsibility for sending any signals, including `SIGSTOP` and `SIGCONT`, to the parallel tasks. Rather, it can only deliver these signals to the initial, single instance of the job, and hope this job will forward them to the spawned parallel tasks.

The execution mechanism used in the Berkeley PBS environment, GLUnix, forwards all signals received by the spawning task (called `glurun`) to all spawned parallel tasks by default, simply by catching the signal and passing it across the network. However, `SIGSTOP`, used by the PBS scheduler to suspend a job, cannot be caught — by definition — and thus cannot be passed to the parallel tasks! Instead, the delivered `SIGSTOP` simply suspends the `glurun` task itself (which is not doing any of the actual parallel processing; it simply manages the parallel tasks). The parallel tasks thus continue running while the `glurun` task ceases all processing. This has the relatively disastrous effect of preventing the `glurun` task from reliably capturing the output of the parallel tasks, while not suspending those tasks at all.

The solution to this problem was effective, but very far, indeed, from elegant: rather than just delivering a `SIGSTOP` to the process, parallel jobs (those requesting more than one node) were delivered a secondary signal (under the Solaris operating system, `SIGFREEZE`, chosen because its delivery to processes that do not catch/recognize it is simply ignored). The `glurun` program was modified to recognize this signal, translate it to the required `SIGSTOP`, and forward *that* signal on to the spawned parallel tasks before suspending itself. This procedure, though messy at best, would result in the desired suspension of the actual parallel tasks rather than just the `glurun` process. The resumption signal, `SIGCONT`, *can* be caught and thus did not pose the same problem.

5.2.4.2.3 Benchmarking

The final challenge in providing a suspend/resume facility to this system arises from the frequent use of the system for timing runs (benchmarks) of various sequential and parallel codes: a timer is read, code is run, the timer is read again, and the total time elapsed is assumed to be the difference between the ending and starting values. If processes are not suspended and run alone (*i.e.*, are not time-shared with other processes), this method is very effective. However, if processes are time-shared — or, as is the case here, suspended and resumed — this strategy can fail. If the timers used do not measure CPU time, but instead measure elapsed wall-clock time (as many of the high-resolution timers popular for this sort of work do), all time spent suspended will be included in the timings of the code, greatly skewing the results.

This problem is not unique to suspend/resume of jobs; if a job is time-shared with other jobs, the use of wall-clock timers will similarly fail. However, users often use a queuing environment for benchmarking work expressly *because* it does not time-share jobs; they then expect, with some degree of justification, that their timings will be accurate.

Again, the central issue is the failure of the underlying design to plan for a non-dedicated, preemptive environment; and again, the solution is messy but effective. Users can designate a special "pre-suspend" signal to be sent to a process a fixed amount of time (usually fifteen seconds) before it is to be suspended. This signal can be caught by the process, which typically will mark the time on its timers and then suspend itself (by sending itself a `SIGSTOP`). When resumed, the matching `SIGCONT` will be caught; the process will again note the time, advance its recorded starting time by the difference — to compensate for the amount of time spent suspended — and then resume operation.

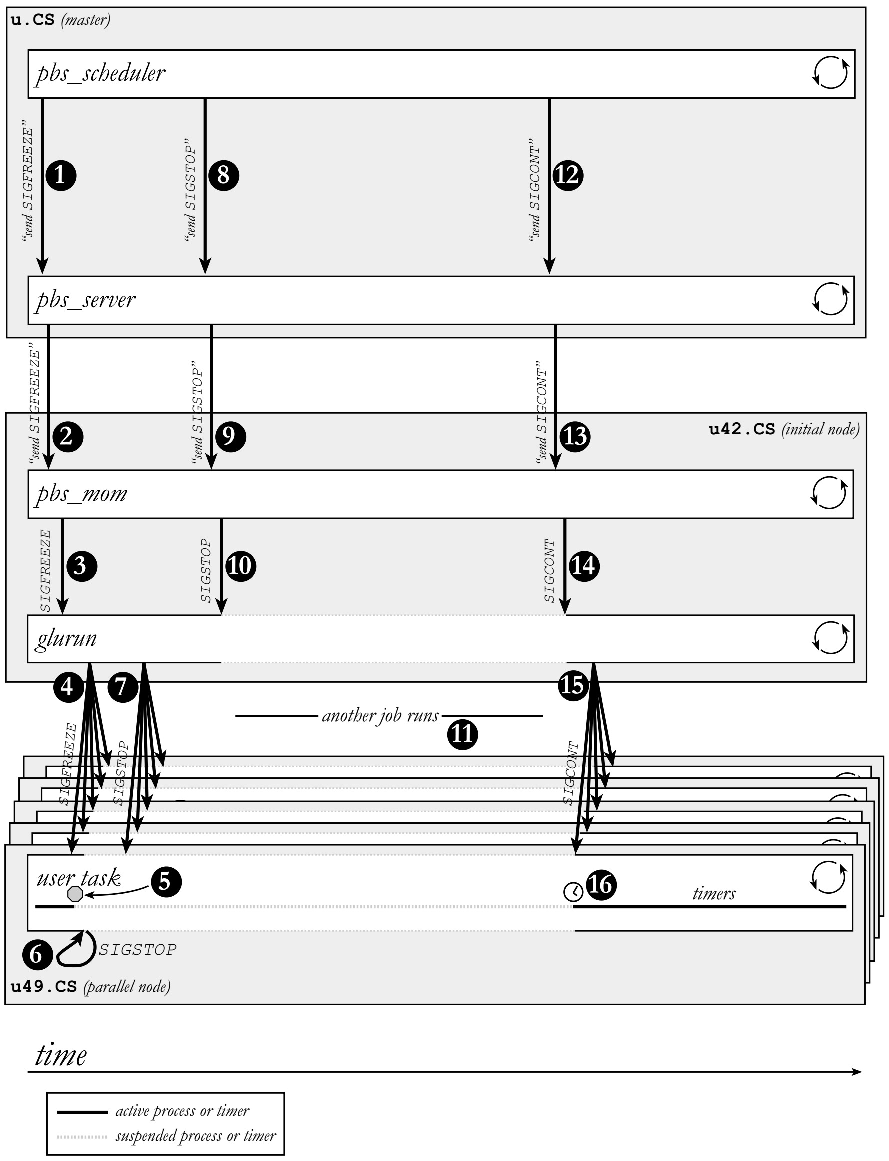


Figure 5.2-C. Sequence of events upon suspension or resumption of a distributed PBS job.

Figure 5.2-C indicates the entire series of events when a job is to be suspended (and, later, resumed).

1. The scheduler decides, using its Vickrey scheduling algorithm, to suspend a job. It sends a message, using the `pbs_ifl` library, to the PBS server, instructing it to deliver the `SIGFREEZE` signal to the job (warning the job that it is about to be suspended).

2. The server forwards this message on to the appropriate PBS MOM process — the one running on the "initial node" of the job, *i.e.*, the node on which the `glurun`

process is executing.

3. The PBS MOM sends an actual `SIGFREEZE` signal to all user processes. If this is a parallel job, this will include the `glurun` process.
4. (Parallel jobs only) The `glurun` process forwards this signal to its spawned parallel tasks.
5. (Parallel, benchmarking jobs only) Each parallel task realizes it is about to be suspended; it stops any running benchmark timers.
6. (Benchmarking jobs only) Each task (parallel or not) sends itself the `SIGSTOP` signal, ensuring that it stops immediately (and thus has the most accurate timing) instead of waiting for the formal PBS `SIGSTOP` signal.
7. (Parallel jobs only) The `glurun` process, after a fixed delay, sends the `SIGSTOP` signal on to its spawned parallel tasks. This is necessary because user jobs might otherwise simply decide to ignore the `SIGFREEZE` signal. Typically, however, the parallel tasks will already be suspended and thus will just ignore the signal.
8. The scheduler, after a fixed delay, sends a message, using the `pbs_ifl` library, to the PBS server, instructing it to deliver the `SIGSTOP` signal to the job.
9. The server forwards this message on to the appropriate PBS MOM process.
10. The PBS MOM process delivers the `SIGSTOP` signal to all executing user processes, ensuring that they are suspended.
11. Another job may run.
12. When the process is again runnable, the scheduler sends a message, using the `pbs_ifl` library, to the PBS server, instructing it to deliver the `SIGCONT` signal to the job.
13. The server forwards this message on to the appropriate PBS MOM process.
14. The PBS MOM process delivers the `SIGCONT` signal to all user processes, resuming their execution.
15. (Parallel jobs only) The `glurun` process forwards the `SIGCONT` signal on to its spawned parallel tasks. This resumes the execution of all spawned parallel tasks.

16. (Benchmarking jobs only) Each task (parallel or not) resumes its timers.

5.2.4.3 GLUnix Reliability

The significant design challenges of this system — primarily those having to do with benchmarking and signaling — have now been described. Design, however, is only half the experience, and so here we consider the other half of the challenges presented to running and using such a system: the experiential, day-to-day challenges of maintaining the entire PBS environment on the Berkeley cluster.

Before this experiment, it was well-known that the underlying GLUnix runtime layer could, and did, fail fairly unpredictably. These failures were due to a number of causes, from small, otherwise-unnoticed DNS configuration problems to the use of blocking I/O only in the GLUnix master server.

While these failures were relatively easy to notice under the traditional interactive usage mode of GLUnix, the underlying failure of GLUnix on a PBS server was often harder to notice and more devastating. As an example, if the GLUnix master hung (blocked for I/O from a client program that no longer existed), a user running a job interactively would simply see that his or her job exited immediately with the message "Cannot contact GLUnix master". PBS, cannot, in general, detect these failures, and would simply continue executing jobs in the queue — each of which would run for only a few moments. This causes the consumption of an entire queue full of jobs in only a few minutes, resulting in no useful work and frustrated users.

More generally, the underlying system fails in ways that are not easily programmatically recognizable; these failures propagate upwards and cause generalized failures of the entire PBS system. This pattern reinforces the notion that systems, *especially* distributed systems, must make failure at least as apparent as success.

6 Simulating an Economy with *econsim*

The full-scale implementation of an economic queuing system, as described in the previous chapter, was a clear step forward in our efforts to analyze the effect such a system would have on users. We could now put such a system into place, maintain it, and have it collect usage data for us over time; later, this data could be studied and analyzed at length.

The difficulty with this process is twofold. First, it requires, quite simply, a long time. Usage of the Berkeley NOW on which the economic queue is installed can, as previously demonstrated, fluctuate wildly over time; it is necessary to collect data over a period of several months (if not, indeed, years) to obtain an accurate picture of how users react to the economic queue. Second, this economic queue has limited flexibility: we cannot manipulate the algorithms and parameters used without risking invalidation of all the data we have gathered. Given the wide variation in cluster usage over time, it is difficult or impossible to conduct controlled, rigorous experiments.

Because of these limitations, we chose to continue our investigation into economic queuing systems by building a simulator: a software system that modeled the actual, deployed economic queuing software as closely as possible, but which could simulate the response of the system to various workloads and user behavior patterns in a matter of hours, not months or years. Below, we describe the motivation behind this simulator, the design of the simulator, and the results we obtained using the simulator.

6.1 Motivation

There were several motivations behind the development of the simulator. First, its ability to simulate many hours of time during each minute of real time provides a flexibility the real system does not: it is possible to repeatedly simulate substantial periods of real time, each time changing only one part of the simulator's behavior — thus obtaining reproducible, comparable results that can be compared and analyzed with certainty. Second, the fact that the simulator is composed entirely of software, with no human interaction, means that any parts of the simulator that need to be changed, *can* be changed. Instead of guessing at what users are doing and being restrained in making changes to a production system, the simulator can be changed rapidly and re-run over various periods of time to examine the effects such changes have caused. Third, results from the simulator can have a direct impact on the real-world, production system: as new results are obtained and new ideas are developed, they can be validated using the simulator, giving a much higher level of confidence and control when changing the production queuing system.

Most importantly, however, the simulator's development arose out of a direct need: over a year of fairly detailed cluster usage data has been recorded and made available to those conducting cluster research. Ignoring this data, in light of the analyses conducted by this project, would be a significant waste of existing resources about how a batch queuing system might affect a group of real-world users. Indeed, this reason was the immediate motivation for construction of a simulator — harnessing a year's worth of usage data to shed insight into some of the issues and decisions necessary in implementing a batch-mode computational economy.

6.2 System Design

Because the full-scale, real-world economic queuing system had already been completed and was fully working, developing a simulator to mimic the actions of this system became a much more straightforward problem: by closely imitating the design of the actual, full-scale economic queue, the simulator was assured of remaining faithful to the behavior of the "real thing" and of being straightforward to implement, manipulate, and extend. Thus, the simulator is designed as a generic framework that models the same PBS system components used in a the full-scale system: various components simulate the PBS server, PBS execution nodes, and the scheduler itself.

Many components of the simulator are considerably simpler than their real-world counterparts; this is primarily due to a lack of error checking and reporting. (Simulators never have hardware failures, nor do they need to worry about network errors or dead daemon processes.)

6.2.1 Block Structure

Figure 6.2-A shows the basic modules (Java classes) used in construction of the simulator. We consider each of these modules in turn:

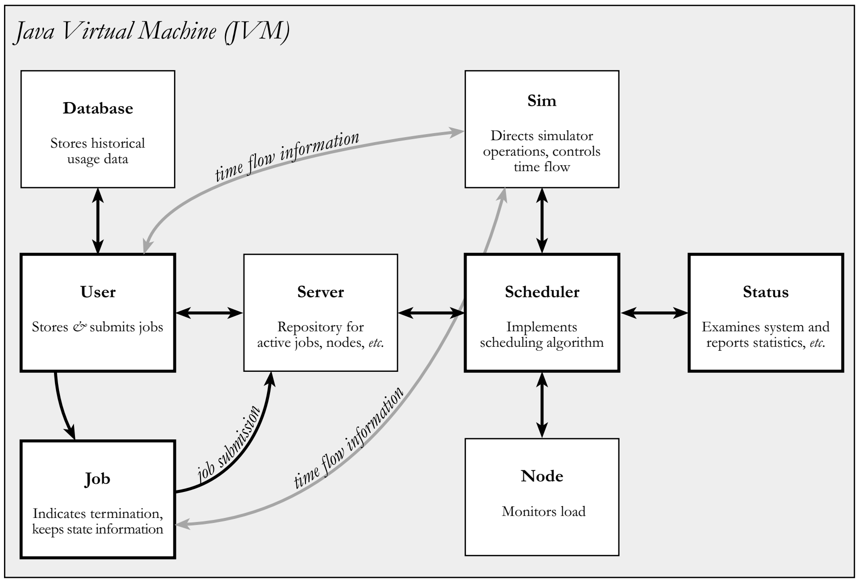

Java Virtual Machine (JVM)
Database
Stores historical usage data
Sim
Directs simulator operations, controls time flow
time flow information
User
Stores & submits jobs
Server
Repository for active jobs, nodes, etc.
Scheduler
Implements scheduling algorithm
Status
Examines system and reports statistics, etc.
job submission
time flow information
Job
Indicates termination, keeps state information
Node
Monitors load

Figure 6.2-A. Internal class structure of the simulator.

1. **Sim**. This module directs overall simulator operations; it also controls the time flow in the system (see below). It is responsible for initiating scheduling decisions and providing all of the other modules in the simulator an opportunity to "trigger" (do something) at each simulated time step. The Sim module also has another, crucial function: it controls the flow of time in the entire simulator. Because the simulator leaps through time, from one event to the next, various modules must be given the opportunity to participate in the mechanism by which the next simulated time is decided. The Sim module thus communicates with any objects that register their interest in this process, allowing them to set the next time to be simulated; the next time simulated is the soonest time requested by any element of the simulator.

2. **Scheduler**. This module is virtually identical to the real-world PBS scheduler: it retrieves job and node information from the server, computes the desired state of the system from this information, and issues commands to the server to bring the system into this new state.

3. **Server**. This module is also virtually identical to the real-world PBS server, though much simpler; it stores a database of all submitted jobs and all available nodes, allowing the scheduler to access this information.

4. **User**. The User module simulates the actions of an individual user. It draws the

user's historical record from a database (using Java Database Connectivity — JDBC) and, based on this record, submits jobs to the server's queue for immediate or later run (depending on the scheduling algorithm). The User module is also responsible for setting a bid on the job; how it does this is discussed below, in the section describing the plug-in architecture of the simulator.

5. **Database**. The Database module is simply a thin wrapper around a Java Database Connectivity (JDBC) connection to a remote SQL database that stores all the historical records available. Users connect to this database to obtain their lists of available jobs; information such as job name, original submission time, runtime, degree of parallelism, and so forth are available.

6. **Job**. The Job module is responsible for simulating a job; it keeps track of the job's original submission time, and informs the server when it is considered terminated. (Typically, the Job will simply monitor the total amount of time it has been running and indicate termination when this reaches the original runtime of the historical job, but this is not necessarily the case.) Jobs may exist but be unknown to the Server; these jobs are not considered to be submitted yet and thus are not eligible to be scheduled.

7. **Node**. The Node module is responsible for monitoring a node; it keeps track of the total number of jobs on the node at any given point and thus maintains a node load average.

8. **Status**. The Status modules make the simulator useful: they collect information from the Server, Jobs, and Nodes, and write useful information (such as node load average, number of jobs in the queue, total time a job was queued before being run, *etc.*) to output files. A number of Status modules have been developed; any and/or all can be used at runtime, depending on the data that should be collected.

6.2.2 Cycle

Figure 6.2-B shows the order of operations in a single cycle of the simulator, from top to bottom. Because the actual mechanisms of scheduling (and, indeed, the scheduling algorithm itself) are essentially identical to the real-world economic scheduler, they are not shown further here.

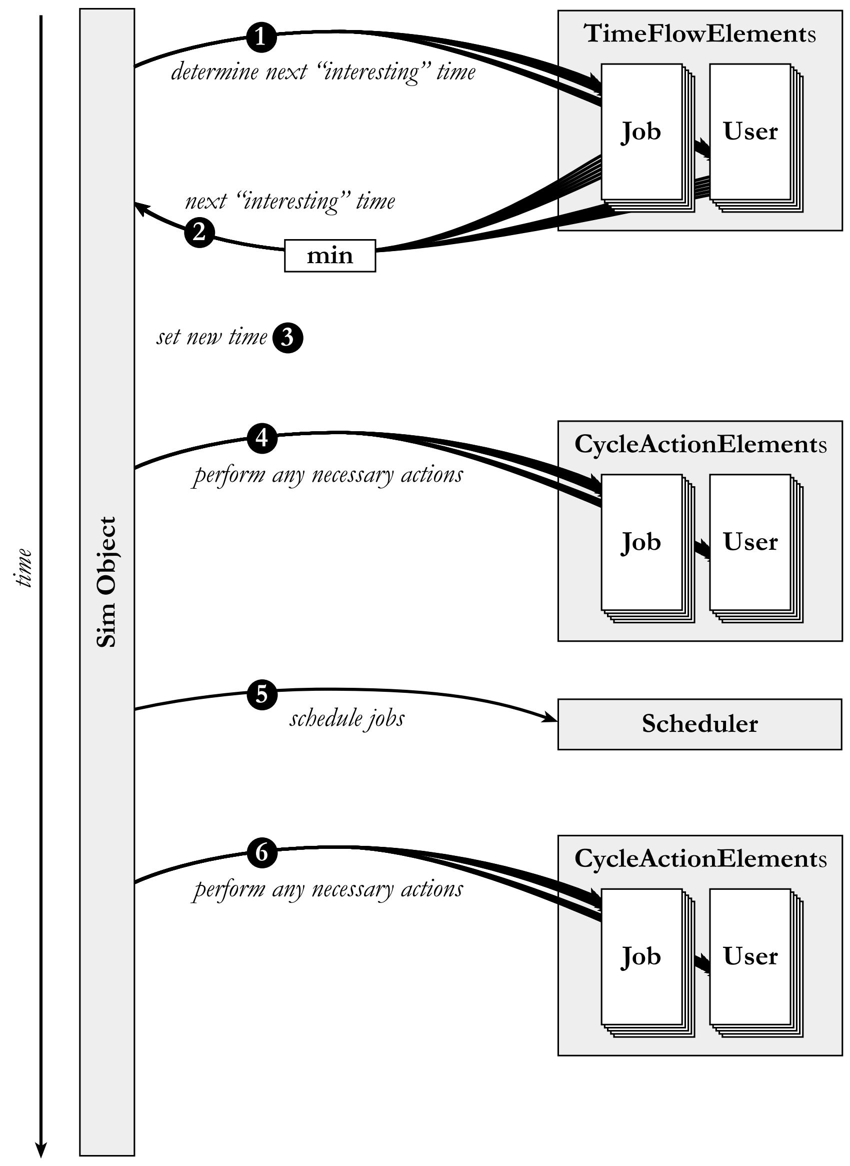


Figure 6.2-B. Sequence of events on each scheduling run.

1. **Query for next time step**. The simulator's first job on each cycle is to determine the proper simulated time. Rather than use a discrete, constant-time-flow model — where each cycle would simulate, *e.g.*, one second — the simulator is a discrete event simulator: it determines time steps dynamically, by leaping ahead to the next moment where something *changes* in the system.

   Because several different kinds of events can occur — jobs can start, jobs can terminate, users can submit jobs to the system, users can change the billing on a job, users can delete a job, *etc.* — the next time step is determined in a distributed fashion. Objects (such as users and jobs) can register themselves as "time flow elements" with the simulator. On each step, the simulator asks each time flow element for the next "interesting" time it has — that is, the next time when it, by itself, generates an event.

2. **Compute next time step**. The simulator receives responses from all time flow elements, indicating the next point in time when something interesting happens. It finds the minimal time (the soonest time) from all of these, which will be the next time step.

3. **Advance time**. The scheduler now sets the global simulated clock to this next time step.

4. **Pre-scheduling actions**. Similar to the concept of a "time flow element", the simulator maintains a list of "cycle action elements" — those objects that may wish to *do something* at each time step (like submit a job, remove a job, terminate, *etc.*). Prior to the scheduling run, each cycle action element is given the opportunity to do whatever it likes. This step is where, typically, new jobs will be introduced (a User object will notice that, as of the current simulated time, a historical user submitted a job, and will submit a job to the simulator), jobs will terminate (a Job object will notice that, as of the current simulated time, it has spent as much time running as the historical job did, and inform the server that it is terminating), and so on.

5. **Schedule**. The scheduler is now invoked; nearly identically to its real-world counterpart, it carries out the Vickrey scheduling algorithm and instructs jobs to run, suspend, or terminate, as necessary.

6. **Post-scheduling actions**. This step, identical to step 4, gives all cycle action elements another opportunity to perform any action they wish. This is distinguished from step 4 due to the possible change in status of various jobs from the scheduler's run in step 5.

At this point, the cycle is complete, and repeats.

6.2.3 Performance

The scheduler's performance — simulated seconds passed per second of real time — can vary wildly, depending on activity within the system. Measured in simulator cycles, we find that the simulator typically passes between fifty and one hundred cycles per wall-clock second; in our experience, this allows us to simulate a year's worth of data in somewhere between eight and twelve hours of time on a single, 500MHz Intel Pentium III-based computer.

6.2.4 Plug-in models

To this point, little has been said regarding the internal operation of certain components of the system — those components with a heavy outline in Figure 6.2-A. This is because these modules of the system can be extended to provide the simulator with differing behavior, to simulate differing real-world systems. The various mechanisms for extending the system for each of these four modules are discussed next.

6.2.4.1 Simulator

Perhaps the most straightforward extension of the simulator is the scheduler. Like a real-world queuing system, the scheduler that is present in the simulator can have a drastic effect upon the way jobs are chosen and run; also like a real-world queuing system, examining the various outcomes using these differing schedulers is critical to understanding the entire system.

The simulator allows for wholesale replacement of the scheduler. Two schedulers have been developed to parallel the schedulers that exist in the real-world system: the **Vickrey** (Economic) scheduler and the **GLUnix** scheduler, and one simulator has been developed to parallel a scheduler that is in very common use elsewhere: the **FIFO** (first-in-first-out) scheduler.

The Vickrey ("Economic") scheduler is, exactly as its name implies, a completely faithful model of the economic scheduler used in the PBS system. It orders jobs according to bid, begins running them, and sets system price at the bid of the highest-bidding job that cannot run.

The GLUnix scheduler is a faithful simulation of the *prior* real-world system (run on GLUnix alone, hence the name of this scheduler). It runs jobs immediately upon submission, placing them on the least-loaded nodes, but makes no attempt to ensure that jobs run alone on nodes. If more jobs are available at any given point than nodes, some nodes will have two (or more) jobs running on them and thus the underlying node operating system will time-share them. This scheduler is useful for contrast with the Vickrey scheduler, for validating proper simulator operation, and for collecting statistics that were not available in the historical record (such as node load averages over time).

The FIFO scheduler is a faithful, strict first-in-first-out queue-based scheduler: jobs are run in the strict order in which they entered the queue; at no point is any job run before any other job that entered the queue ahead of it in time. This scheduler is useful because, although the GLUnix scheduler was used in the Berkeley NOW environment for several years, it is not terribly common outside our particular environment; however, the FIFO model is extremely common and well-understood, so we chose to include it for comparison in our analysis.

6.2.4.2 User

The User module of the simulator is the second major module of the scheduler that can be replaced or extended. While there is no corresponding User *module* in the real-world queuing system that can be replaced, actual *users* in the real-world system can (and do!) behave very differently from each other. Some users may submit a very large number of jobs at once with a zero bid, allowing them to run "as time allows"; some users may apportion their jobs, ensuring that only a few are every queued at once and adjusting bids to make sure their jobs are always on top; some users may rarely use the system, only to come in at the last moment and be willing to bid whatever is necessary to ensure their jobs run immediately.

To capture this behavior, the simulator allows the wholesale replacement of the User module. Users, in the simulator, have complete control over their jobs: they read information about historical jobs from a database, but are in no way required to maintain consistency with these jobs when submitting jobs to the simulator's Server module. Users may choose to submit jobs at the same time they actually were submitted in the past; they may delay submission, submit early, or skip submission entirely. Users are also responsible for determining the bid to be placed on each job; they may bid zero always, a fixed high number, a fixed total amount per job, and so on. Because the User module is instantiated once for each user of the system, Users may collect data about how their jobs have performed in the past and allow it to influence future decisions.

In order to ensure that experiments were consistent across runs of the system, each User module records the bid it generates for each job in permanent database storage. When only the scheduler is to be varied from one run to the next, the Database User module (see below) is used, which recalls generated bids from this database; this ensures that identical bids are used for identical jobs and maintains integrity of the experiment.

6.2.4.2.1 Current User modules

Eight different User modules have been developed for the simulator. Currently, all eight modules have significant commonality: they all submit jobs at the exact time they were submitted in the historical data; they all leave the jobs running until they terminate; and they all modify only the bid associated with the job. The eight User modules are:

Zero-Bid User. This User submits jobs with a constant zero bid. Of note is that if this User is combined with the Vickrey scheduler, the result is a traditional, first-in-first-out queue.

Constant-Total-Bid User. This User submits jobs so that the product of their node count (degree of parallelism), run time, and bid is a fixed constant — *i.e.*, their "total" bid is always the same. Thus a job that runs twice as long as another receives exactly half the bid.

Proportional-Bid User. This user places a bid directly proportional to the runtime of the job; in this sense, it is the exact opposite of the Constant-Total-Bid user. Thus a job that runs twice as long as another receives exactly double the bid.

Random-Bid User. This user chooses a bid randomly from a uniform distribution on a fixed interval for every job.

Categorized-Bid User. This user attempts to place each job into exactly one category of job based on its duration (for example, "short", "medium", or "long"), and then assigns a randomized bid to each job based on its category. For example, "medium" jobs may be defined as those jobs lasting between fifteen minutes and one hour; these jobs may be given a bid chosen uniformly from the interval (25.0, 75.0).

Binary-Categorized-Bid User. This user categorizes each job into one of two categories: "short" or "long"; "short" jobs are given a single, constant "high" bid, while "long" jobs are given a single, constant "low" bid. This user was created in an attempt to model, in part, the behavior of actual users as they use the real-world economic-scheduling system.

Binary-Random-Bid User. This user randomly assigns a constant "high" or constant "low" bid to each job with a fixed probability; for example, 30% of all jobs (chosen randomly) may be assigned a "high" bid, with all remaining jobs assigned a "low" bid.

Mixed-Bid User. This user randomly decides to emulate one of the other seven modules; at startup, it selects one of the other modules, and proceeds to process all jobs as if it were that module. Thus the system can be simulated using a variety of different bidding strategies all at once.

Database User. When any given User module assigns bids to jobs, that User module can, optionally, record the bids it assigns into a database. The Database User then replays that record. This is used to ensure consistency in results: when varying a Simulator module but not the User modules across various runs, and when the User module assigns bids in a nondeterministic fashion (*e.g.*, the Random-Bid User), the Database User module is used for all but the first run to ensure that precisely the same bids are assigned each time.

6.2.4.3 Job

The Job module of the simulator can also be replaced or extended. In reality, the behavior of a user's job has an enormous impact on the performance of the system and resources required; memory-bound jobs, I/O-bound jobs, and CPU-bound jobs all behave quite differently. Because, however, we have chosen to monitor only CPU time, the Job module here is not as significant as the User or Scheduler modules; it monitors runtime only.

At the moment, then, the ability to plug in different types of Job modules is mostly useful for dealing with schedulers that sometimes time-share jobs on various nodes. When a job is time-shared, it typically slows down (thus increasing its total duration); this slowdown may vary due to the type of job. For example, given a node with n additional jobs on it, large parallel jobs may incur a slowdown of $3n$, small parallel jobs a slowdown of $2n$, and sequential jobs of n. However, I/O-bound or interactive jobs may not incur any significant slowdown at all. By using different Job modules for these different types of real-world jobs, the simulation may be made more accurate.

6.2.4.4 Status

The whole point of the simulator — to produce output that can be analyzed and viewed — is accomplished via a series of Status modules that hook in to the simulator. Each Status module contains a set of about a dozen callbacks ("job begun", "job ended", "scheduling run complete", and so on) that are invoked when the associated events have occurred. In turn, each callback is passed appropriate information (job name/size/bid, queuing delay/runtime, jobs in the queue, *etc.*) for that callback.

Typically, a Status module will then use this information to produce records in an output file that can later be read and analyzed. By creating a series of Status modules and adding them all to the system, a full picture of the simulator's behavior can be obtained — while still maintaining a modularity and comprehensability that would disintegrate if simple "output" statements were inserted throughout the scheduler.

Of particular interest is a status module collects all the statistics produced by individual modules on each cycle of the simulator, and aggregates that information into minimum, maximum, mean, median, and sum values over defined time periods (currently, each hour). This produces a "running total" picture of the scheduler as it makes its way over the simulated time period.

6.2.5 Data Sources

The data that is to be used as input to the scheduler can come from virtually any location. As previously noted, the major source is a set of log files produced by the GLUnix runtime system over more than a year. These log files produce one entry for each job submitted to the system; this entry contains the name of the user submitting the job, the executable file name, the time and date of the request, the duration of the job in wall-clock milliseconds, and the degree of parallelism of the job.

Generally, this is sufficient information to simulate such a job being input into a simulated system. However, due to the time-shared nature of the GLUnix system, the logged duration of a job can be quite different from the duration that the job would have had were it not time-sharing its nodes with other jobs: that is, if a job ran concurrently with three other jobs, its logged duration would be expected to be about four times as long as if it were run alone.

To compensate for this fact, as data is input into the simulated system, we measure the total load on the NOW cluster, expressed as a ratio of the total number of nodes occupied by jobs to the total number of nodes available. If this ratio is greater than one, then we divide the logged duration of a job by it before sending the job on to the simulated system. While this system is far from ideal, it compensates for this time-dilation effect to a moderate degree. It is impossible, from the data that were collected, to know whether two jobs did, in fact, share a set of nodes. Further, even if we had this information, it is impossible to know to what degree jobs' runtimes are increased by time-sharing with other jobs (*cf.* the related work on implicit coscheduling [25]). This compensation will, at least, adjust job runtimes to be somewhat more in line with those that would be expected were the cluster space-shared instead of time-shared.

7 Results/Analysis

Once the simulator had been built and the economic queuing system deployed, these two systems immediately began generating data. In this next section, we investigate the data gathered by the previous system (GLUnix), our economic queuing system, and the results of the simulator. We consider the quantitative effects of each system on job delay and run times, and the effect of bids within the system; we also consider the qualitative, observed effects of each system. Finally, we draw conclusions from the quantitative and qualitative results, summarizing our experiences with and data received from these systems.

7.1 Real Systems

7.1.1 GLUnix

Our first analysis simply concerns the existing GLUnix scheduler's performance over the year that we studied it. Fortunately, GLUnix was designed to record certain information about each and every job it processes; it writes these values into a log file. The values written are:

- Date and time that the job began;
- Name of the user who requested that the job be run;
- GLUnix "Npid" (global PID) for the job;
- Number of nodes on which the job ran in parallel;
- Duration (wall-clock time) of the job;
- The exact command issued.

By parsing this log file and placing its data into a database table, we were able to produce several analyses of the existing system's performance. Also, by tuning the simulator to exactly replicate the scheduling patterns of GLUnix, further analyses of the existing system's behavior could be produced.

7.1.1.1 Workload

We consider first the workload placed on the existing system; this not only characterizes the conditions under which the GLUnix scheduler operated, but it is also the input to our simulations of economic scheduler behavior.

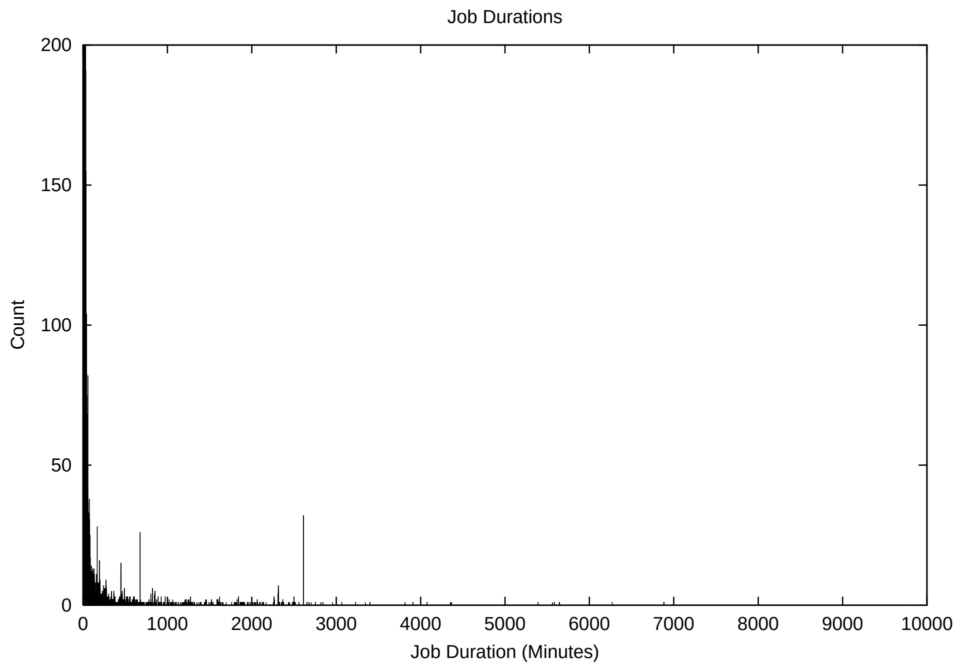


Figure 7.1-A. Histogram of job durations for the existing GLUnix system.

Figure 7.1-A is a histogram of the durations of all jobs that were submitted to the system over the year studied. Because of the tremendous variation in job length, the range of this histogram is necessarily limited; over 96% of all jobs lasted less than five minutes, while some jobs lasted for several days. Nevertheless, given that over a half-million jobs were processed during the time studied, this histogram demonstrates the trend expressed by the above statistic: very few jobs (less than 0.4%) lasted longer than one hour, while the number of jobs whose duration is best measured in seconds is very large.

This vast proliferation of short jobs in the system clearly has implications for any resource allocation strategy to be used; however, it is notable that the very design of the GLUnix system encourages exactly such submissions. Rather than submitting a job to a queue (and the associated user and system overhead), GLUnix responds in the same manner as the system `rexec` command, immediately running a job and printing its output to the user's terminal. This permits and encourages users to think of GLUnix as they would a typical system utility, and significantly alters the mix of job durations submitted to the system.

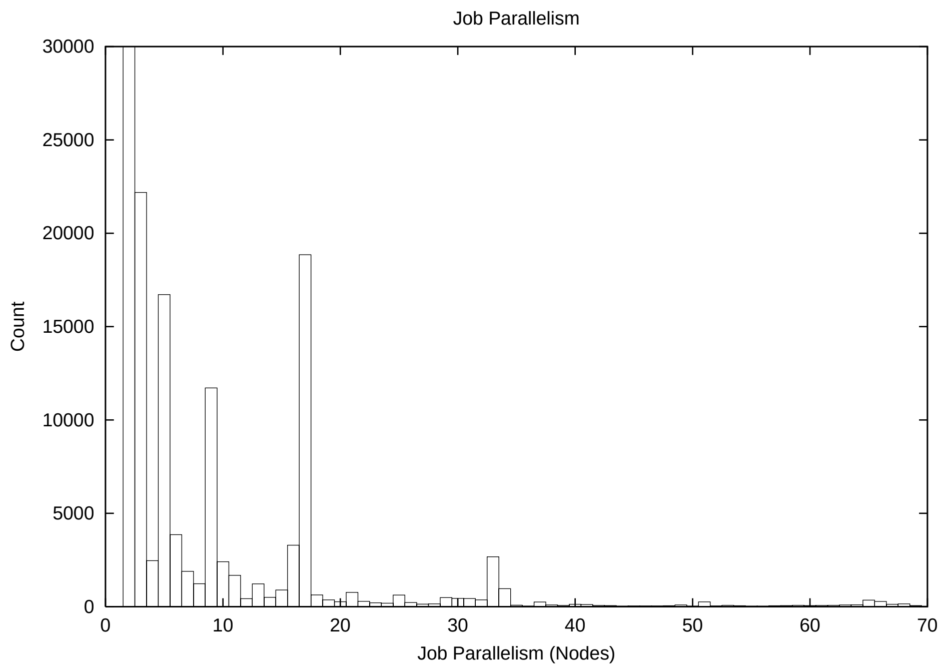


Figure 7.1-B. Histogram of job parallelism for the existing GLUnix system.

Figure 7.1-B is a histogram of the number of nodes requested (parallel degree) for each job processed through the system. Again, the range of the graph is truncated due to extreme outliers; 423,430 jobs requested only a single node of execution. If we consider only *parallel* jobs — those requesting at least two nodes — we find that of the total of 105,187 jobs, 21.1% (22,187 jobs) requested two nodes, 15.9% (16,716) requested four nodes, 11.1% (11,712) requested eight nodes, 17.9% (18,848) requested sixteen nodes, and 2.5% (2,669) requested thirty-two nodes. These "powers of two" degrees of parallelism thus accounted for 68.6% of all parallel jobs.

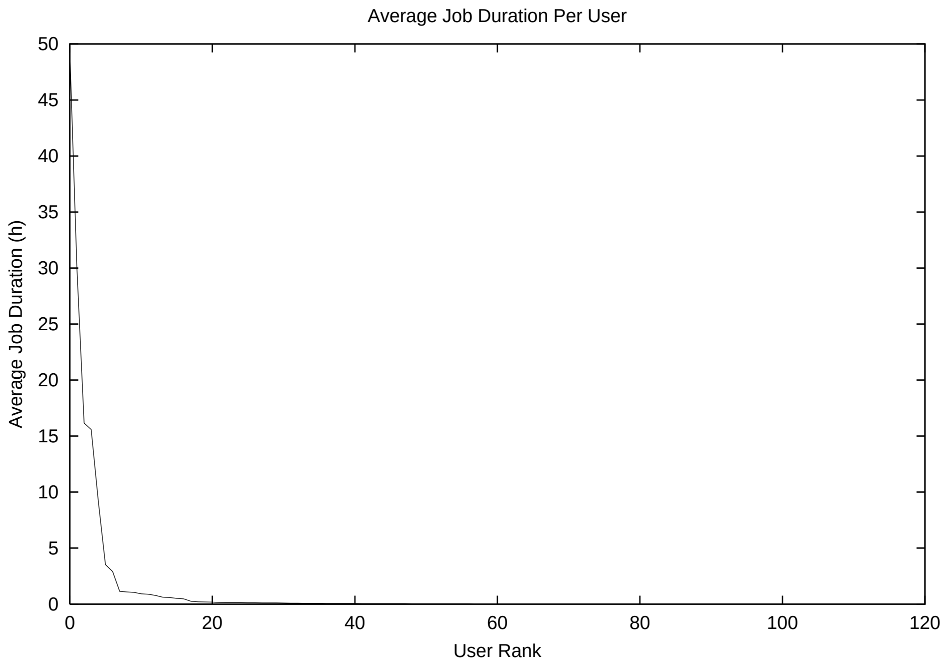


Figure 7.1-C. Average duration of each user's jobs.

Figure 7.1-C shows the average duration of each user's submitted jobs, as expressed in hours. While the vast majority of users (over eighty percent) submitted, on average, very short jobs, some users were responsible for submitting extremely long jobs: approximately five users submitted, on average, jobs that lasted at least fifteen hours, and the top ten users submitted, on average, jobs that lasted at least one hour. Given that the vast majority of jobs in the system (96%) were less than five minutes long, this wide discrepancy in users' submissions has significant implications for the design of any scheduler used: the schedule must be able to process short jobs efficiently, but fairly allocate resources for very long jobs, also.

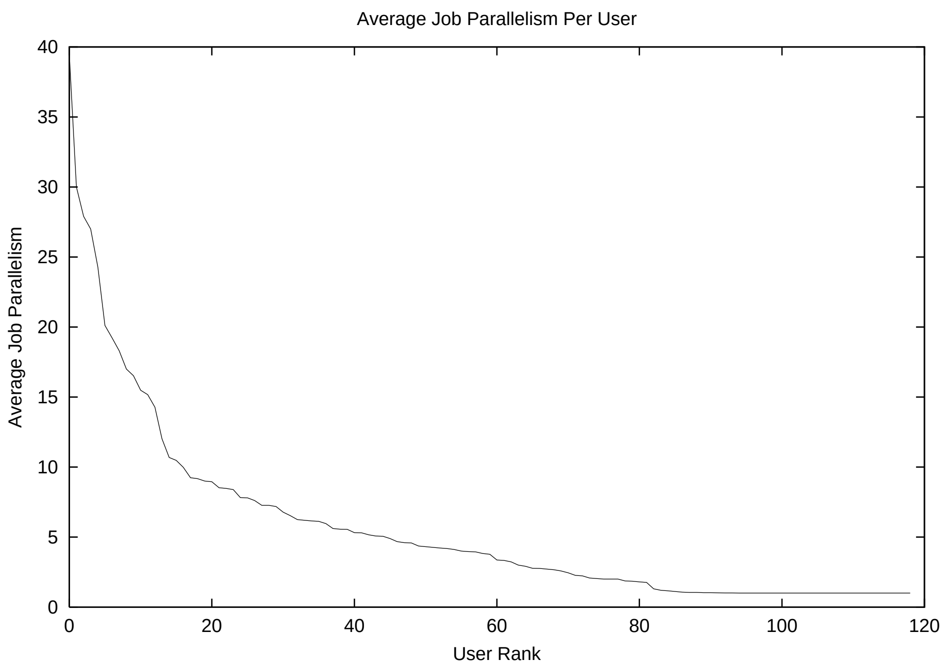


Figure 7.1-D. Average parallelism of each user's jobs.

Figure 7.1-D describes the average parallelism of each user's jobs: that is, for each user, it indicates the mean parallel degree requested across all of that user's jobs. This graph shows a more-gradual decline than the previous one, indicating that there is less drastic variation among users when it comes to parallelism of jobs. Still, it is clear that fully half of all users submitted jobs with four nodes or fewer on average, while ten users submitted jobs with an average parallel degree of at least sixteen.

7.1.1.2 Scheduling Behavior

We turn, now, to examination of the actual behavior of the GLUnix scheduler as it reacted to the workload placed on it over the period studied.

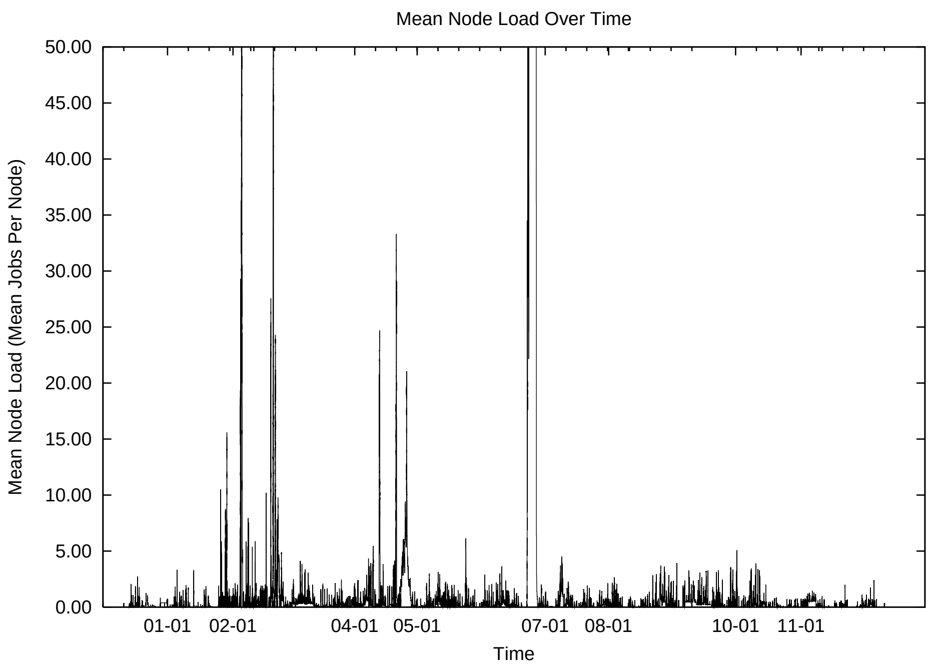


Figure 7.1-E. Mean running jobs on each node (approximate node load average) over time, existing GLUnix scheduler.

Figure 7.1-E shows the approximate average load average of the cluster over the year studied. Because true load-average data was not available, this data was instead obtained by "playing back" the existing GLUnix logs against the simulator, with the simulator precisely replicating the behavior of the GLUnix scheduler.

A cursory examination of this graph demonstrates the sheer variability of the load on the cluster: while the cluster spent large amounts of time idle or nearly so (days, even, at times), during short periods it was extremely overloaded, with a mean load average of ten or greater. Depending on the exact nature of the programs running, such a degree of time-sharing on the cluster may produce acceptable results (each program encounters a slowdown only roughly proportional to the number of competing jobs) or unacceptable slowdowns (due to the lack of gang scheduling at the operating-system level, messaging desynchronization occurs for those programs not using a technique like implicit coscheduling [25] and slowdowns are increased by an order of magnitude or more). Beyond the obvious enormous spikes in load, too, we see a multitude of small spikes showing an increase in time-sharing to a factor of 2–5 jobs. While this level of time-sharing usually does not exhaust physical memory, inter-node scheduling issues may still result in substantial slowdown above and beyond the typical time-sharing results.

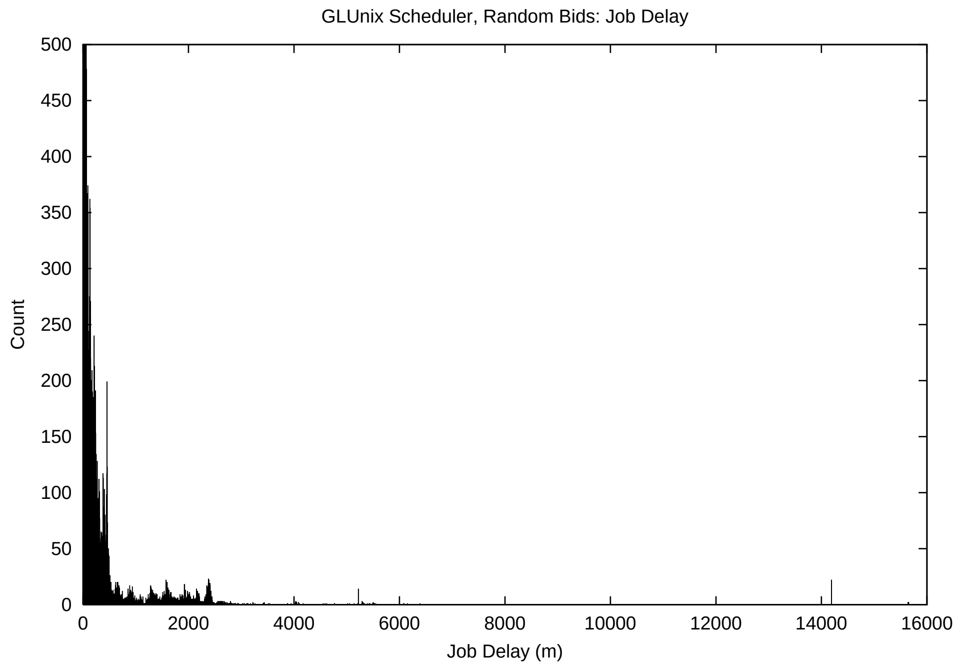


Figure 7.1-F. Histogram of job delays, GLUnix scheduler.

Figure 7.1-F is a histogram of the total amount of "delay" imparted to jobs due to GLUnix's scheduling policy of simply allowing the system to time-share the cluster. That is, for each job, we measure the total duration as actually executed on the cluster, the total duration if that job were to run by itself on the cluster, and compute the difference; this is the amount of time the job's completion was delayed by competing jobs, because GLUnix always begins running a job immediately.

It is clear that, while the vast majority of jobs had very small or no delay from GLUnix whatsoever, significant numbers of jobs have substantial — and even enormous — delays introduced. For example, dozens of jobs have delays introduced of over 2,000 minutes — which is more than 33 hours.

7.1.1.3 Analysis

From the previous graphs, several conclusions can be drawn regarding the way GLUnix schedules jobs on a cluster of workstations and its impact on users.

First, and most obviously, GLUnix's policy of running all jobs immediately and simply allowing the underlying operating system to time-slice among them produces enormous variation in the amount of time a job is delayed due to competition: while most jobs run with very little or no delay, some jobs are delayed immensely due to the competition from other jobs; in the worst case, this delay can be greater than an entire day. While this delay, from a user's standpoint, is somewhat predictable — users are at least somewhat aware of likely times for the cluster to be very busy — it nevertheless greatly decreases the confidence with which a user can be certain that his or her jobs will complete within a bounded amount of time.

Second, it is obvious that, even without a mechanism for the explicit queuing of jobs, delays very similar to the delays introduced by a queuing system nevertheless appear in the system. For short jobs, GLUnix is an effective scheduler, as, even in heavily-loaded situations, short jobs will complete relatively rapidly; for long jobs, however, GLUnix is a poor scheduler, as the increased delay for long jobs becomes increasingly unpredictable. Further, because this delay is distributed across the length of a job and can vary due to jobs submitted to the cluster after a job has begun running, a user can never truly be confident of a job's end time until that job has actually completed running.

7.1.1.4 Anecdotal Evidence

Because user behavior is a primary part of this study, we consider also the "soft" evidence of sheer experience with the system. In this section, we therefore attempt to express the general experience of administrators and users with the GLUnix system's scheduling behavior.

Of paramount importance is the evidence, both anecdotal and data-supported, regarding the *workload* placed on the cluster. As we will soon see, once the economic scheduling system was introduced, the cluster's workload dropped off dramatically; the observations made of users' behavior under GLUnix support this conclusion. In particular, under GLUnix, users tended to submit anything and everything that might conceivably be productive to the cluster: programs were not always fully debugged before being run, users were not certain that old jobs that they had created were actually killed when attempting to terminate them, and, in general, users tended to run jobs with little regard to the current demand on the cluster or other users' requirements.

We consider several possible explanations for this behavior:

- Submitting jobs to GLUnix is almost trivially easy: rather than having to prepare a script or job definition file, users simply need to learn the proper syntax of the `glurun` command.
- To a naïve user, GLUnix behaves almost identically to a standard single-user interactive system: jobs run immediately, and running several jobs at once simply slows down each job proportionally.
- GLUnix gives immediate feedback on all commands submitted; there is no queuing delay, so there is little advantage to be gleaned from waiting to submit a job until it truly is fully debugged and tested. Why bother debugging your program on a small set of nodes, when you can simply run it on a full-scale 32-node set — it might be bug-free already, after all!
- Because there is no queuing delay, the effect of other users' jobs on a users' own job (and, more importantly, vice versa) is not always immediately evident. Instead of finding that the presence of other users' jobs causes a substantial delay before a job begins running, a user soon discovers that contention for the cluster is only really important on final, production runs — and since one user's production run comes during another user's debugging session, there is little incentive to avoid causing contention with other users' jobs.

We consider, next, the reaction of users to the various scheduling decisions made by GLUnix. As is evidenced by the data presented, the system spent most of its time idle or nearly idle — certainly in a state where the aggregate demand for the system was less or much less than supply. In this state, users were free to submit jobs and have them run at will.

However, during periods of high demand — typically the few days or weeks preceding a major conference submission deadline or preceding the end of each semester, when final projects were due — cluster and user behavior moved into an entirely different mode. Because aggregate demand outstripped supply, users would submit their jobs, only to find them running far more slowly than expected. At this point, several types of behavior were observed:

- Some users simply waited for their jobs to complete, accepting the delays as inevitable. These users typically were the least disruptive to the rest of the system; however, users who were running jobs that attempted to measure performance over time typically received inaccurate (sometimes wildly) results, due to the multitude of context switches.

- Some users gave up, canceling their jobs entirely and missing deadlines (or, in the case of conferences, filed for extensions). While this behavior did not disrupt other users' jobs at all, we would like to ameliorate the need for this sort of behavior and provide users maximum flexibility in scheduling jobs. (Certainly, users will always procrastinate, and there will *never* be enough time left at the last minute; however, we would like to allow users this leeway as much as possible.)
- Some users had administrative privileges on the cluster (due to their affiliation with the NOW research group) and would use these privileges to pre-empt other users' jobs. This did not come in the form of abruptly terminating other users' jobs; rather, these users used their power to make *reservations* (see section 3.1.3 for details) that prohibited other users from using certain nodes for a certain period of time. While this allowed these users to complete their work easily, this mode of operation is extremely human-intensive: in essence, these users were scheduling the cluster entirely by hand, and using an administrative status to give themselves priority over other users.
- Some users used the cluster-status tools to determine who the other users competing for time on the cluster were, and contacted them directly (typically via electronic mail or in person) to resolve the conflicts. This was often somewhat successful, as the parties in question would at least re-order their jobs and avoid competing with each other. However, this was far more successful among close research associates, who had a previous interpersonal relationship, than among strangers; further, it was sometimes a source of interpersonal conflict, as users attempted to determine whose job was "more important". Finally, this, too, is not a scalable result, as it requires humans to make resource-allocation decisions based upon their own statements about the importance and immediacy of their work.

The anecdotal evidence, then, seemed to indicate two major faults of the current system.

First, because GLUnix ran users' jobs so eagerly, users rarely refrained from submitting anything to the cluster; instant feedback encouraged this behavior, and so substantially more demand was placed on the cluster than might otherwise be the case. Second, when the cluster was in a state where demand exceeded supply, *no* good resolution came from users: they would simply tolerate the delay, give up, or manually intervene in the scheduling process. Because there was no way for users to express the importance of their jobs to the system, and no way for the system to give direct feedback to the users, the system would degenerate to a process of human intervention and manual scheduling under stress.

7.1.2 PBS Economic Queue

From November 10, 1999 until May 16, 1999 — a period of approximately six months — the PBS system was installed and active on the Berkeley NOW as its primary method of resource allocation. The Vickrey economic scheduler, whose theory and implementation is described in section 5.2, was used as the scheduling policy for the entire PBS system. We present here the results of that experiment.

Of the approximately one hundred nodes present in the Berkeley NOW, nodes were assigned as follows:

- Seventeen nodes were assigned to a "debug/interactive-use" partition at the front of the cluster; this partition was scheduled using the existing GLUnix toolset, meaning jobs were run immediately upon submission, and any excess demand was absorbed by time-sharing this portion of the cluster as before.
- Seventy nodes were assigned to the primary PBS-scheduled partition, as use for production runs of user programs.
- The remaining nodes (between eight and thirty, depending on the experiments being run on them) were made available for those users conducting research that entirely prohibited all schedulers — for example, those students installing experimental network drivers and/or operating system kernels on machines that therefore were frequently rebooted or crashed.

During this time, all activity on the PBS-scheduled partition and all activity directly involving the PBS scheduler or system itself was meticulously recorded in a database. We present that data here.

7.1.2.1 Data

A total of 23 distinct users used the PBS system during this period; a total of 2,386 jobs were processed by the system. This is approximately 0.5% the number of jobs processed by the GLUnix system during the year studied; if we compensate for the longer duration of the GLUnix study, about 1% as many jobs per unit time were run on the PBS system as on the GLUnix system. Similarly, only about half as many users used the PBS system as the GLUnix system, one the difference in time periods is compensated for.

However, those 2,386 jobs:

- Were more likely to actually be parallel jobs than the previous GLUnix system: 30.2% of PBS jobs used at least two nodes, versus 19.9% for GLUnix, and the mean PBS job used 4.7 nodes, versus 3.5 for GLUnix;

- Lasted much longer than jobs in the previous GLUnix system: GLUnix jobs lasted, on average, 7.00 seconds, while PBS jobs lasted, on average, 12,434 seconds (about three and a half hours).

We now consider this data in graphical form.

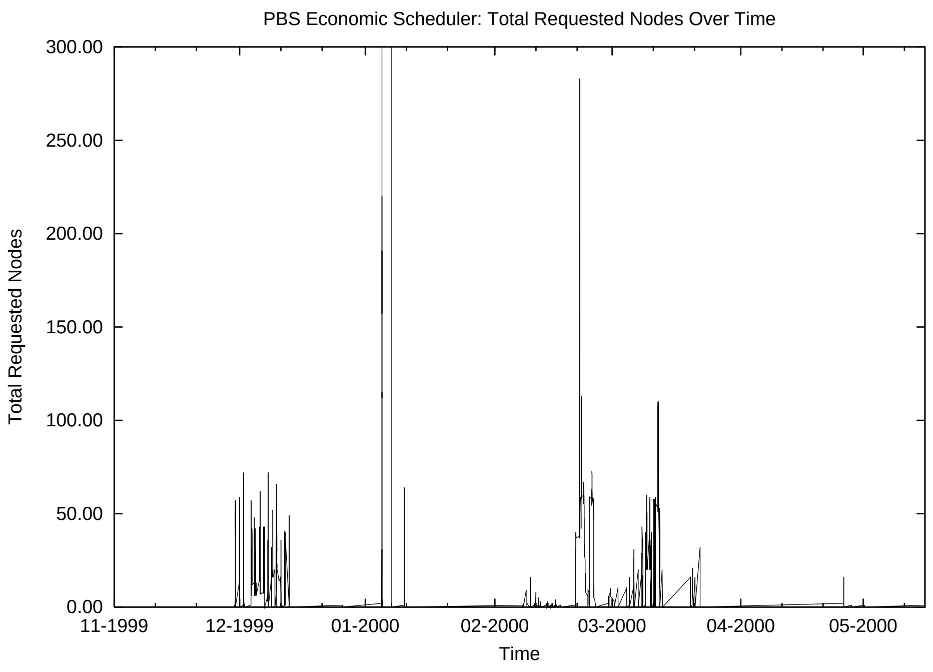


Figure 7.1-G. Sum total of nodes requested in the queue, over time,

PBS economic scheduler.

Figure 7.1-G displays the sum of the total number of nodes requested by all jobs in the PBS queue, over time. From this graph, two factors become apparent. First, the total demand placed on the PBS system was much less than that placed upon the previous GLUnix system. Second, aside from three short spikes, demand for the cluster very rarely exceeded supply — about seventy nodes were available to process PBS jobs at all times, and thus aggregate demand of fewer than seventy nodes indicates an oversupply situation.

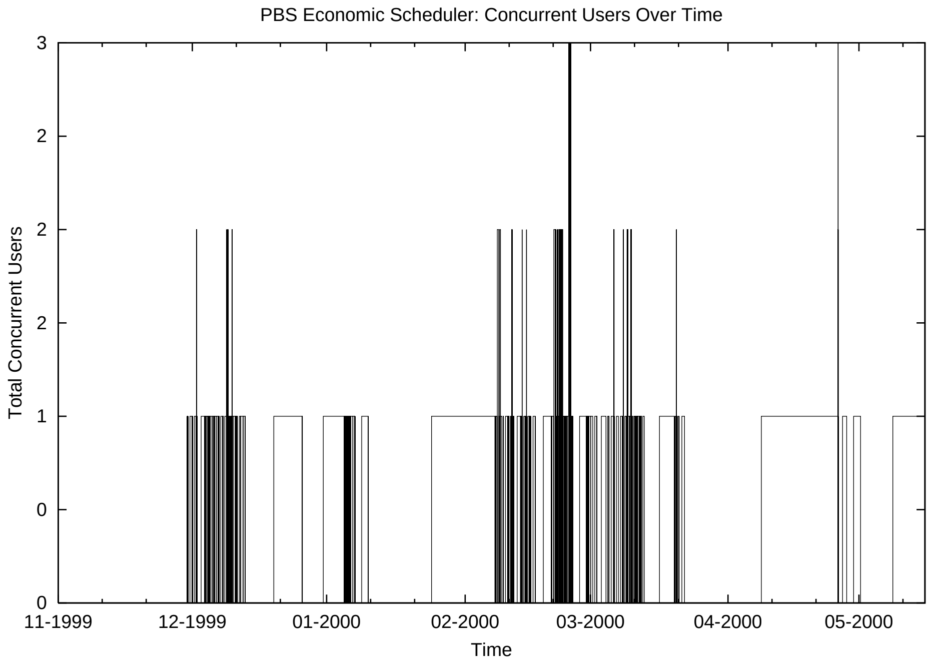


Figure 7.1-H. Total distinct users with jobs in the PBS system, over time.

Figure 7.1-H displays the number of distinct users with jobs in the PBS system (whether awaiting execution or actually executing) over time. This graph reveals a critical point regarding analysis of the PBS data: very rarely was there *any contention at all* for cluster resources, and, when there was, it was limited to two (or, exceedingly rarely, three) distinct users. Even during times when demand exceeded supply, the number of users competing for the cluster was limited to one (as in the mid-January spike in demand) or two to three (as in the late-February spike in demand).

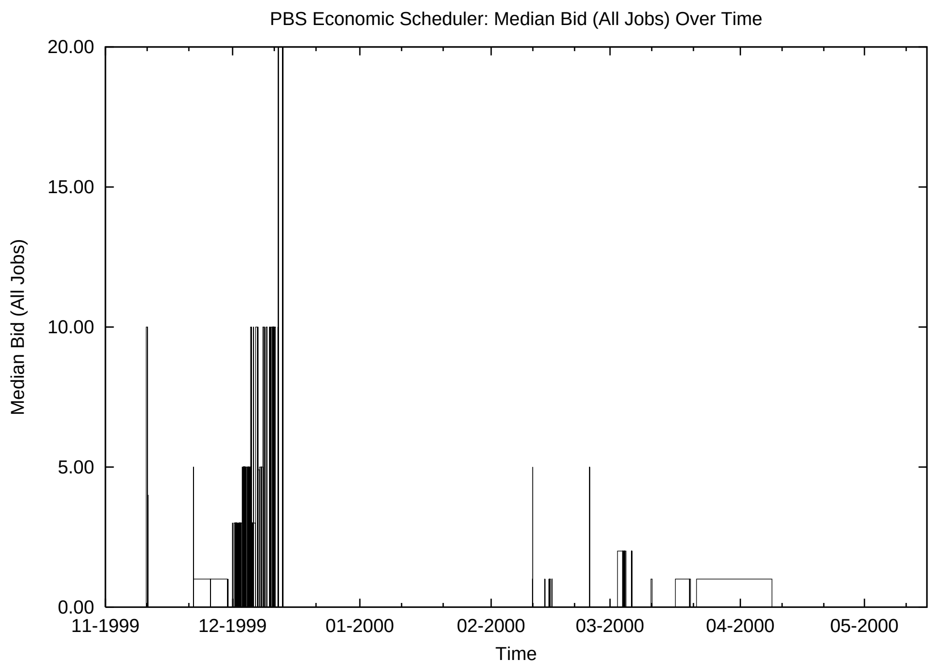


Figure 7.1-I. Median bid in the PBS system, over time.

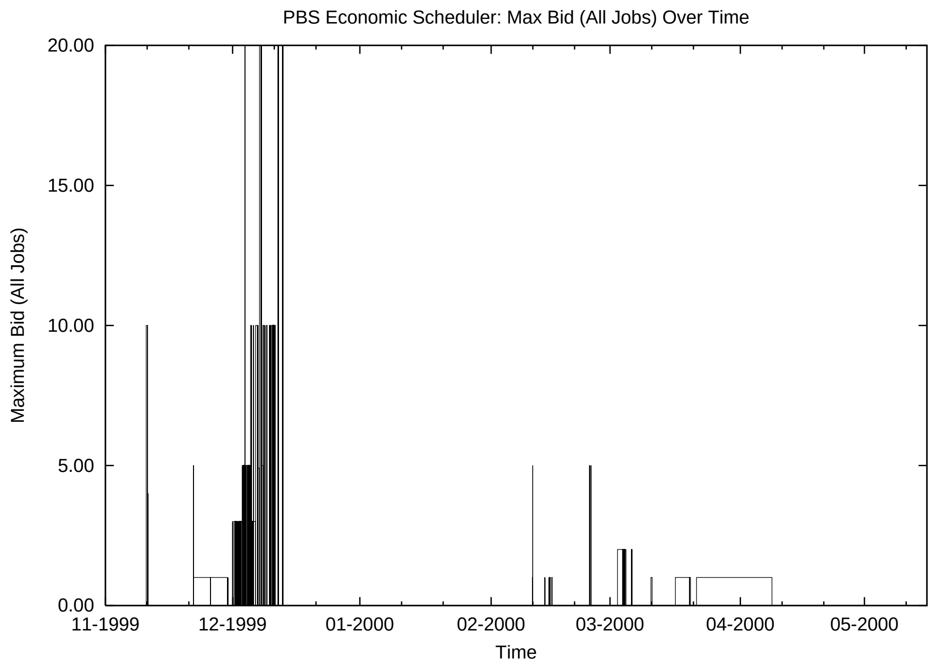


Figure 7.1-J. Maximum bid in the PBS system, over time.

Figure 7.1-I displays the median bid of all jobs in the PBS system over time; Figure 7.1-J displays the maximum bid of all jobs in the PBS system over time. By carefully comparing these graphs to the previous graphs presented, it becomes apparent that little or no connection exists between the bidding behavior of users and either the total demand placed upon the system or the total number of concurrent users contending for time on the system. In particular, the spikes in median/maximum bid available in the system come at times when only a single user had any jobs submitted to the system at all. Anecdotal evidence indicates that these spikes in bidding behavior were due to users' desire to experiment with the system and "play with the scheduler" rather than users' desire to actually promote the priority of their jobs.

7.1.2.2 Analysis

From the data presented regarding the PBS economically-scheduled system, several conclusions can be drawn.

First, and foremost, the simple reduction in traffic to the cluster was dramatic — astonishing, even, given the previous level of use of the GLUnix system. While it is impossible to determine the exact cause of this phenomenon, we attribute it to several causes:

- Users were aware that the system had an enforceable resource-allocation policy in place. Previously, the GLUnix runtime system had been viewed, correctly, as being primarily concerned with *mechanism* of execution, as opposed to *policy*. When the transition to PBS was made, users were immediately aware that resource usage was both a concern of the administrators and an issue that was being monitored; users therefore were more careful about their resource demands.
- The added time delay between submission of a job and its execution — even when significantly less than a minute — caused users to carefully consider whether their programs were ready to be run in a production environment.
- The availability of a small partition of the cluster for interactive programs and small-scale debugging runs meant that this traffic was offloaded from the production-level cluster.
- The direct expression of the cluster's level of usage as a fictitious "price" further caused users to reconsider whether they wished to demand resources from the cluster immediately, or whether they would be better-suited by running on their

own workstations or in the interactive/debugging partition.

- It is also possible that users simply had inherently less work to do during our experimental period than they had in the previous year.
- Use of the PBS cluster may have been more inherently difficult (due to the new toolset) than it had been before.

In general, to our surprise, we found that the simple presence of *any* scheduling system was sufficient to reduce demand to the cluster so much that there were no longer any substantial resource-allocation decisions left to be made! Indeed, merely *exposing* the fact that there was a nonzero cost of resources in the cluster seemed to dramatically reduce users' demands for those resources — in our case, so much that only in a few rare cases did the scheduler actually have any decisions to make about which jobs would run.

We also consider the observed reactions of users to the system, namely:

- Perhaps partially because our user base was drawn from an academic environment, users' response to the economic scheduler was one of general acceptance, but with the caveat that very few users understood exactly how they should assign bids to their jobs.
- Because no significant contention for the cluster ever occurred, users had little motivation to seriously consider what bids they were assigning to each job. Users tended to bid in a roughly random binary fashion — they assigned a bid of zero or "something" to each job, with very little correlation between the bid and any observed characteristics (runtime, parallel degree, *etc.*) of the job.
- This lack of motivation to seriously consider bids was exacerbated, probably greatly, by the fact that users were not bidding using "real" money: the credits that they used to bid for jobs could not be used to purchase any other resource, whether on the cluster or in the external world, and thus could be spent relatively freely.

Overall, our data suggests that while the system *did* solve a number of problems that had existed with the Berkeley NOW cluster previously, users did not exhibit much in the way of true economic behavior: users did not compete via price, nor did they judge their bids relative to their actual value of the job's execution or any external resource.

7.2 Simulator

Because the greatly diminished overall usage of the cluster led to a substantially smaller result set than expected, we focused our experiments on the *econsim* simulator and its application to historical data. Specifically, a number of user-simulation modules were developed to simulate both observed user behavior from users' initial experience with the system, and user's potential future behavior patterns as they gain experience with the system and their bidding behavior matures. By using these modules in conjunction with the gathered historical behavior, analyses of the system's behavior under different user and scheduler behavior can be thoroughly developed.

7.2.1 Data Set

All simulation was conducted using the same set of historical information: data on all users' jobs from December 13, 1998 through December 7, 1999 were gathered from log files produced by the GLUnix system and stored into a relational database for later access by the scheduler.

7.2.1.1 Jobs

Each job's record contained the:

- Name of the command executed;
- Duration of the job;
- Start time of the job;
- Degree of parallelism of the command (number of nodes used concurrently); and
- User who executed the command.

While a complete characterization of the job load is completely beyond the scope of this paper, several straightforward statistics are provided here:

- There were a total of 528,617 jobs.

- Running jobs used a total of 348,805,340 node-seconds; given that there cluster averaged about 100 nodes available over the 359 days studied and thus had 3,101,760,000 node-seconds available, this is a total usage rate of about 11.2%.
- The degree of parallelism of all jobs ranged from 1 (a sequential job) to 106 (a job using all nodes in the cluster). There were 423,430 sequential jobs — 80.1% of all jobs executed. 22,187 jobs (4.2%) used two nodes, 16,716 (3.2%) used four nodes, 11,712 (2.2%) used eight nodes, and 18,848 (3.6%) used sixteen nodes. Altogether, these "power-of-two" job sizes accounted for 495,911 jobs (93.8%). 3,747 jobs (0.7%) used more than sixty-five nodes (essentially "the whole cluster").
- Job durations ranged from 1 second to 583,105 seconds (about 6 days, 18 hours). Of these, 243,270 (46.0%) lasted less than five seconds, 353,259 (60.6%) lasted less than thirty seconds, 403,043 (76.2%) lasted less than one minute, 508,657 (96.2%) lasted less than five minutes, and 526,382 (99.6%) lasted less than one hour. Table 7.2-A gives a more detailed summary of these results.

Table 7.2-A. Percentile rank of job durations.

Job Rank by Duration	**# Ranking More Than**	**Job Duration**
10%	475,755	0.26 s
20%	422,894	0.56 s
30%	370,032	1.29 s
40%	317,170	3.22 s
50%	264,309	7.00 s
60%	211,447	18.71 s
70%	158,585	36.29 s
80%	105,723	77.66 s (1 m 17.6 s)
90%	52,862	150.73 s (2 m 30.7 s)
95%	26,431	222.58 s (3 m 42.6 s)
97.5%	13,215	564.22 s (9 m 24.2 s)
99%	5,286	1,563.59 s (26 m 3.6 s)
99.5%	2,644	3,013.60 s (50 m 13.6 s)
99.9%	529	27,301.40 s (7 h 35 m 1.4 s)

- Jobs lasting less than five seconds used a total of 1,354,443 node-seconds (0.4%); jobs lasting less than thirty seconds used a total of 8,080,453 node-seconds (2.3%); jobs lasting less than one minute used a total of 20,372,025 node-seconds

(5.8%); jobs lasting less than five minutes used a total of 55,120,498 node-seconds (15.8%); jobs lasting less than one hour used a total of 141,626,116 node-seconds (40.6%). Table 7.2-B gives a more detailed summary of these results.

Table 7.2-B. Percentile rank of jobs by node-seconds used.

Job Rank by Node-Seconds Used	**# Ranking More Than**	**Job Node-Seconds Used**
10%	475,755	0.26
20%	422,894	0.63
30%	370,032	1.83
40%	317,170	4.55
50%	264,309	12.46
60%	211,447	27.93
70%	158,585	60.34
80%	105,723	127.91
90%	52,862	235.80
95%	26,431	852.66
97.5%	13,215	2,126.72
99%	5,286	7,147.72
99.5%	2,644	11,597.32
99.9%	529	95,962.80

- This, then, satisfies the classic "80/20" rule: nearly all the jobs (96.2%) were short (less than five minutes), but nearly all the cluster's time (84.2%) was spent executing long jobs (those over five minutes — 3.8% of all jobs).

7.2.1.2 Users

- There were 120 distinct users of the cluster over time time period studied.
- Users ranged from those executing only 1 job over the time period studied to a user who executed 142,096 jobs (26.9%). Eleven users (9.2%) were responsible for 422,949 (80.0%) of all jobs executed. Table 7.2-C gives a more detailed summary of these results.

Table 7.2-C. Percentile rank of users by number of jobs executed.

User Rank by Jobs Executed	# Ranking More Than	Jobs Executed
10%	107	2
20%	95	10
30%	83	33
40%	71	92
50%	59	203
60%	47	571
70%	35	1,188
80%	23	2,143
90%	11	11,173
95%	5	28,436
97.5%	2	58,947
99%	0	142,096

- Users ranged from those consuming only 1 node-second to a user who consumed 52,805,002 node-seconds (14.9%). Eighteen users (15%) were responsible for 281,307,419 (80.6%) of all node-seconds consumed. This also satisfies the classic "80/20" rule: although many users used the cluster, only a few (15%) were responsible for most (80%) of the load placed on the cluster. Table 7.2-D gives a more detailed summary of these results.

Table 7.2-D. Percentile rank of users by node-seconds consumed.

User Rank by Jobs Executed	# Ranking More Than	Node-Seconds Consumed
10%	107	55.29
20%	95	1,643.17
30%	83	9,853.74
40%	71	47,343.05
50%	59	214,996.47
60%	47	678,868.26
70%	35	1,649,441.44
80%	23	3,482,336.09
90%	11	7,104,234.56
95%	5	13,013,128.82
97.5%	2	27,785,191.47
99%	0	52,805,002.34

7.2.1.3 Graphs

Also provided are several graphs for a visual characterization of the workload.

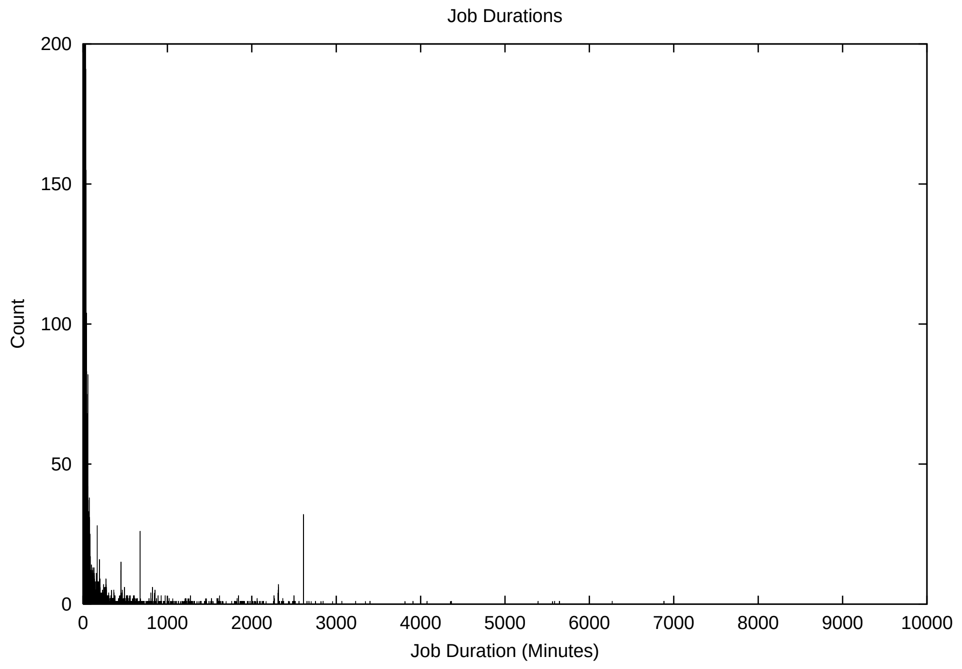


Figure 7.2-A. Histogram of all job durations.

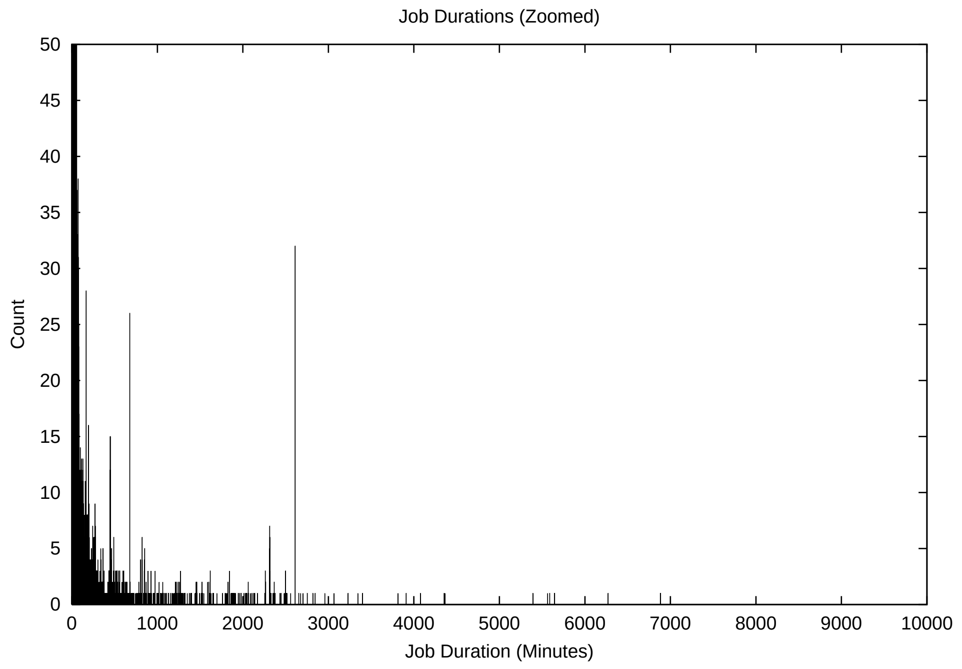


Figure 7.2-B. Histogram of all job durations (zoomed).

Figure 7.2-A and Figure 7.2-B are histograms of the durations of all jobs that have run through the system; different ranges are provided for detail. As outlined above in section 7.2.1.1, the vast majority of all jobs ran for only a few minutes; less than one job in 200 lasted for longer than an hour.

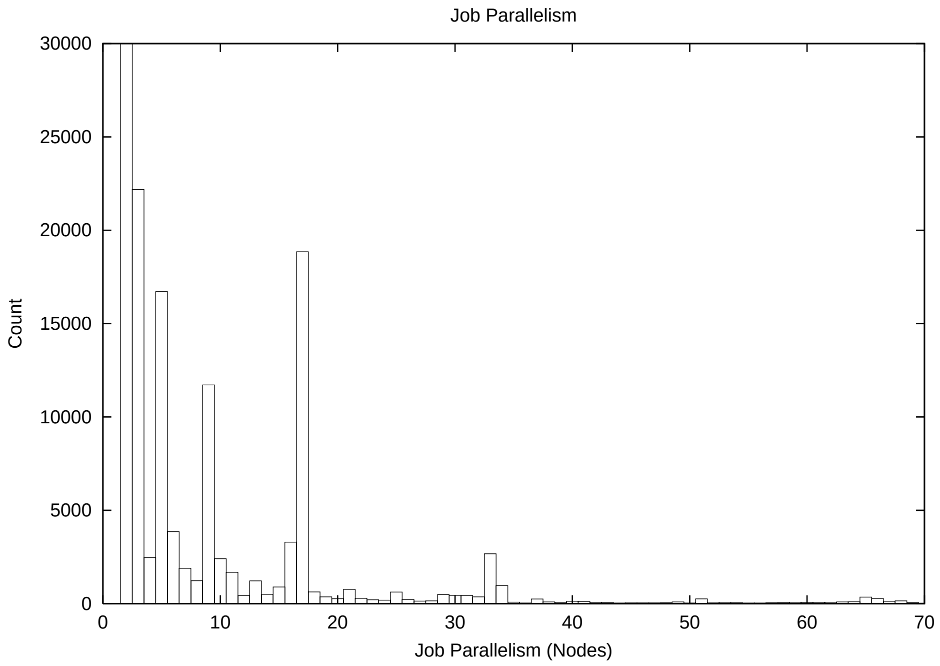


Figure 7.2-C. Degree of parallelism of all jobs.

Figure 7.2-C is a histogram of the sizes (degree of parallelism) of all jobs that have run through the system. As previously mentioned, the number of jobs using only one node is extremely high, accounting for fully 80% of all jobs run. Also evident in this histogram is the expected tendency of users to run their job on a number of nodes that is a power of two.

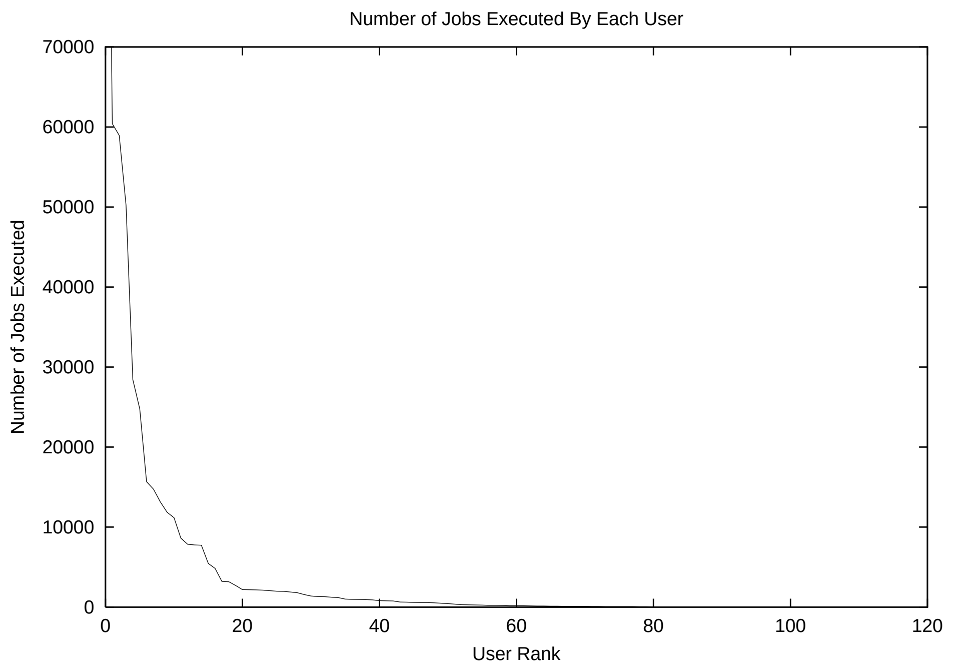


Figure 7.2-D. Number of jobs run by each user.

Figure 7.2-D is a graph of the number of jobs run by each user. We see here the expected pattern: a small number of users account for the vast majority of all jobs run; of the roughly half-million jobs run, fully 100 of the 120 users ran fewer than 2,500 jobs.

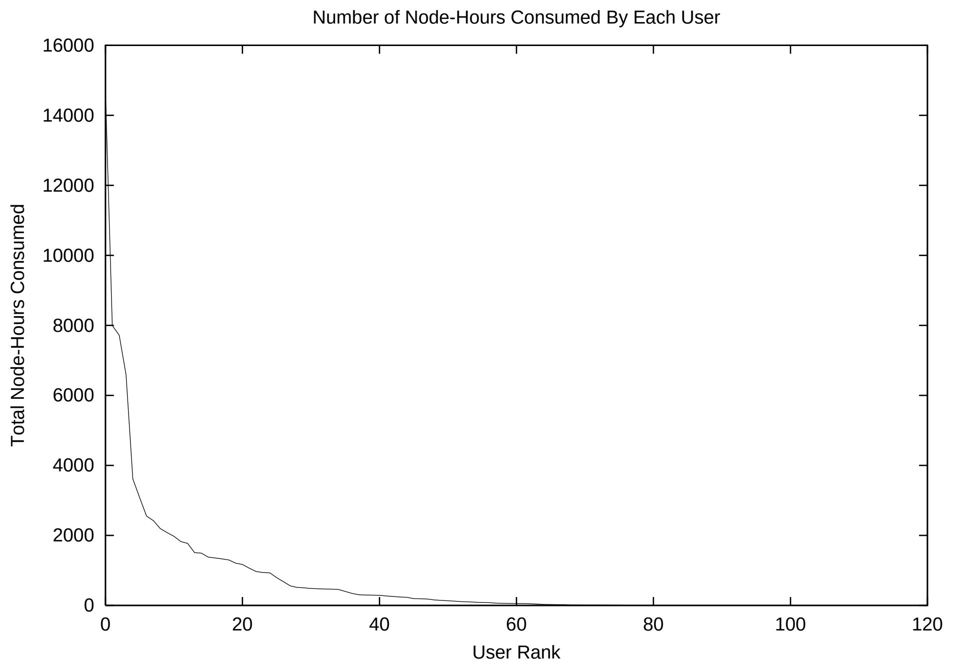


Figure 7.2-E. Node-hours consumed by each user.

Figure 7.2-E is a graph of the number of node-hours consumed by each user. This graph appears similar to the previous one, only exaggerated yet more; a very small number of users (18, of the 120 distinct users observed) accounted for eighty percent of all usage of the cluster.

7.2.1.4 User Modules

We turn, now, to the various User modules used in the simulator to produce results. Eight User modules were created and used for the data analysis; they are described here.

Zero-Bid. The Zero-Bid User module simply assigns a bid of zero to each job. It is used as a base-line result for each scheduler type.

Random-Bid. The Random-Bid User module chooses a bid uniformly from the range [0.0, 50.0) for each job, without regard to that job's characteristics. This module is used to simulate the case where a user's bidding behavior bears no relation to any immediately observable characteristic of his or her jobs.

Constant-Total-Bid. The Constant-Total-Bid User module chooses a bid for each job such that the product of its runtime, node count, and bid is equal to 1000.0. This module is used to simulate the case where a user places most importance on getting short and small jobs finished quickly, while long and/or large jobs are allowed to wait — *i.e.*, short and small jobs are used to generate results needed immediately, while long and/or large jobs are "production runs" where a significant delay is less important.

Proportional-Bid. The Proportional-Bid User module chooses a bid for each job exactly equal to the product of its duration and node count. This module is used to simulate the case where a user places most importance on ensuring a lack of delay of large, "production" runs, and considers smaller, shorter jobs less important.

Categorized-Bid. The Categorized-Bid User is, perhaps, a more-realistic version of the Constant-Total-Bid User. While shorter jobs are still given a higher overall bid, instead of continuously varying the job's bid with its duration, jobs are placed into one of several "categories", as shown in Table 7.2-E:

Table 7.2-E. Categories of jobs for the Categorized-Bid user.

Type	**Minimum Duration**	**Maximum Duration**	**Lowest Bid**	**Highest Bid**	Fraction of All Jobs
Very Short	—	5 minutes	125.0	275.0	96.2%
Short	5 minutes	15 minutes	60.0	140.0	2.2%
Medium	15 minutes	1 hour	25.0	75.0	1.1%
Long	1 hour	1 day	10.0	40.0	0.4%
Very Long	1 day	—	5.0	15.0	0.04%

These categories were chosen based upon the conceptual definition of a job as "very short", "long", *etc*. Obviously they do not come close to evenly distributing the jobs among them, because users' conception of a "very short" job may not be accurately reflected in the actual duration of that job when compared to others.

Binary-Random-Bid. The Binary-Random-Bid User attempts to simulate the observed behavior of users who actually used the system: users tend to bid zero or "something". Here, we assume the decision to assign "something" appears completely arbitrary with respect to each job's observable characteristics. Thus, this module assigns a zero bid to each job with probability 0.85 and a bid of 1000.0 to each job with probability 0.15.

Binary-Categorized-Bid. The Binary-Categorized-Bid User also attempts to simulate the observed behavior of users who actually used the system: users tend to bid zero or "something". Here, we assume the decision to assign "something" is based upon the duration of the job: all jobs lasting less than five minutes are assigned a bid of 1000.0, and all other jobs are assigned a bid of zero.

Again, the division of five minutes is based upon users' conceptual views of the system, not actual results: in fact, 96.2% of all jobs last less than five minutes.

Mixed-Module. The Mixed-Module User works by randomly choosing, from the other seven User modules available, a single module for each user, and then following that module's pattern throughout the simulation. In other words, approximately one-seventh of all Users in the system follow each of the other modules.

Because the other User modules can produce wildly different bids, the bids of all modules are adjusted by the Mixed Module so that the sum over all jobs of the product of node count, duration, and bid is roughly constant: that is, the "total amount bid" by a user should remain constant no matter which module he or she is assigned to. This prevents the constants used in each module (*e.g.*, the total product used in the Constant-Total-Bid module, or the high bid in the Random-Bid modules) from skewing the results of the Mixed Module.

7.2.1.5 User Modules Studied

Rather than trying to analyze results for all eight different user modules, we concentrate on the three user modules in particular which we feel are likely to be most representative of a long-term user base for a system with an economic scheduler:

- The **Constant-Total-Bid** module represents a typical user who expects short-running jobs to be completed as quickly as possible, but who is willing to allow long-running jobs to be preempted when necessary; this user assigns high bids to short-running jobs and low bids to long-running ones.
- The **Random-Bid** module represents those users who either do not fully understand the system or whose jobs have values independent of any observed characteristics of the jobs: they assign a bid to the job at random.
- The **Proportional-Bid** module represents a user who values "production", long-running jobs the most; job bids are assigned in direct proportion to the job's runtime. Short jobs are thus assigned small bids, indicating that they may be debugging runs, *etc.* whose results are not as urgent.

7.2.1.6 Scheduler Modules

Three scheduler modules were implemented for the scheduler, in order to analyze their various effects on the performance of jobs in the system. They are:

Economic Scheduler. The Economic scheduler is a faithful re-implementation of the PBS Vickrey scheduler. It schedules jobs according to precisely the same rules, taking into account the bid placed on the job, and is re-implemented only to take advantage of the simplicity provided by the artificial scheduler environment.

FIFO Scheduler. The FIFO scheduler is a very simple first-come, first-served scheduler: as each job is submitted to the system, it is placed at the tail of a queue. At any given point, the system will run as many jobs as possible from the head of the queue; a job ranking lower in the queue is never begun before one ranking higher in the queue.

GLUnix Scheduler. Like the GLUnix parallel-execution software, this scheduler runs all jobs, immediately, as presented; the underlying individual node operating systems are presumed to time-slice the tasks across the cluster. Because our data set provided no information on the precise type of each job — whether I/O-bound, memory-bound, or CPU-bound — we take the maximum number of tasks, *n*, running on any node assigned to a given parallel job and assume that job therefore runs with speed 1/*n*.

A total of twenty-four runs of the scheduler were made. Each scheduler available — Vickrey ("Economic"), FIFO, and GLUnix — was run with all users following, in turn, each of the eight available User bidding patterns. We present the results of these simulations next.

7.2.2 Results and Graphs

For each possible combination of User module and Scheduler module, a full simulation of the entire year of historical data was performed. During the simulation, copious statistics were gathered: 28 individual data points for each job, 21 individual data points for each user, 267 data points for each cycle of the simulator, 6 node-based data points for each period (here, one hour) that the simulator ran, and 1,341 aggregate data points (minimum, maximum, mean, median, and sum of each individual cycle data points) for each period.

7.2.2.1 Definitions

These statistics were then analyzed and used to create various graphs and tables for each combination of User module and Scheduler module. These graphs and tables are described below; first, however, we define terms required to precisely describe the graphs.

Variables. Each job is assigned the following variables:

- *t*, its innate duration (the length of time the job would take to complete if it were running all by itself on the cluster). This value is read directly from the historical data.
- *n*, the node count (degree of parallelism) of the job. This value is read directly from the historical data.
- *Dq*, the queuing delay of the job: how much time it spends waiting between being submitted to the batch system and the time at which it first begins running. This is determined by the scheduler in use; for the GLUnix scheduler, it will always be zero.

- *Ds*, the suspension delay of the job: the sum total duration of all periods during which the job is in the Suspended (not Running) state. This is determined by the scheduler in use; for the GLUnix scheduler, it will always be zero.
- *Dl*, the load delay of the job: the time by which the job is slowed because at least one of the nodes on which it is running is being shared between multiple jobs. This is determined by the scheduler in use; for the Economic and FIFO schedulers, it ideally will be zero, but sometimes increases due to the inability of the runtime system to migrate jobs.

Job Delay. The delay of a job is simply the sum *Dq* + *Ds* + *Dl*: the total amount of time the job had to wait because demand for the cluster outstripped supply.

Of particular importance when comparing the performance of the schedulers is an interaction between this measurement and the type of scheduling used by GLUnix. In general, GLUnix will produce longer delay values than the other schedulers, because GLUnix gives priority to *starting* all jobs immediately, not *completing* them. For example, given ten one-hundred-minute jobs, the FIFO or Vickrey schedulers will schedule them all in sequence, producing delays of (0 minutes, 100 minutes, 200 minutes, …, 900 minutes), for a total delay of 4,500 minutes. GLUnix, however, will schedule them all immediately; they will all complete 1,000 minutes later — each one 900 minutes late — for a total delay of 9,000 minutes. Whether this unfairly penalizes the GLUnix scheduler or not is an open question; for some users, having *some* results immediately may make up for a delayed completion of the job, while for others, partial results are useless.

Job Dilation. The dilation of a job is simply the result $\frac{Dq + Ds + Dl + t}{t}$: the ratio of the actual runtime to the ideal runtime of a job. This minimizes at 1.0 (100%; no delay from competition) and can increase indefinitely (in the case of, typically, short jobs that must wait many, many times their own duration, typically when scheduled under the FIFO scheduler or the Vickrey scheduler).

7.2.2.2 Graphs

Next, we consider the various graphs produced in order to analyze the data. Note that only graphs of relevance to the discussion are included in the paper itself; all 240 graphs can, however, be found at the paper's Web site [45].

- **Graph: Job Delay vs. Bid**. This scatter plot simply displays the relationship between job bid (in bid units) and job delay (in minutes) for every job in the simulation. For economic schedulers, it will ideally demonstrate that higher-bidding jobs have lower delays, and thus have points clustering towards the origin of the graph. Points in the lower-right area represent jobs with high bids and low delays; points in the upper-left area represent jobs with low bids and high delays; points in the upper-right area represent jobs with high bids *and* high delays. This graph will vary with each combination of scheduler and User module; although schedulers other than the Vickrey (Economic) scheduler do not factor bids into their scheduling decision, the resulting lack of correlation between bid and job delay will be apparent in this graph and will be different as different User modules assign different bids to jobs.
- **Graph: Job Delay vs. Bid (Zoomed)**. This graph is simply a segment taken from the previous graph, showing the area near the origin. Because many jobs will have delays much lower than the maximum delay, and because some User modules assign a bid to most jobs that is much, much smaller than the maximum bid assigned to any job, this graph exposes detail that the previous graph cannot. Like this graph's source, this plot will change with each combination of scheduler and User module.
- **Graph: Job Dilation vs. Bid**. This graph displays the relationship between job bid (in bid units) and job dilation (in percent) for every job in the simulation. Note that this graph is often far less tightly clustered than the graph of Job Delay vs. Bid; a job that is only a few seconds long that waits as little as one minute to run will have a net dilation of several thousand percent, and thus may produce points in the upper-right corner of the graph. This graph will also change with each combination of scheduler and User module, for the same reasons as the graph of job delay vs. bid.
- **Graph: Job Dilation vs. Bid (Zoomed)**. As with the graph of Job Dilation vs. Bid, a segment taken from near the origin of the previous graph is displayed

here. Also like the graph of Job Dilation vs. Bid, this graph will change with each combination of scheduler and User module.

- **Graph: Mean Delay (All Jobs) Over Time**. This graph is a time series indicating the mean total delay of all jobs in the system at each point in time. When demand remains less than supply, this decreases rapidly to zero; when supply outstrips demand substantially, this can surge to very large numbers very rapidly. In general, the closer points on this graph remain to the horizontal access, the better-performing the system. Unlike the previous graphs, however, this graph will remain constant across all runs using the FIFO scheduler and across all runs using the GLUnix scheduler: because a job's bid does not factor into scheduling decisions, varying only the bid will not alter the resulting pattern of delay over time.

- **Graph: Mean Dilation (All Jobs) Over Time**. Similarly, this graph displays the (arithmetic) mean total dilation of all jobs in the system at each point in time. This behaves similarly to the graph of Mean Delay (All Jobs) Over Time, except that, because the mean is arithmetic, not geometric, a short job with a very long delay can radically increase the value at any given point in time. For the same reasons that the graph of Mean Delay over Time does not vary within the FIFO scheduler and within the GLUnix scheduler, this graph will not vary within those schedulers either.

- **Graph: Total Delay (All Jobs) Over Time**. This graph simply shows the sum total delay of *all* jobs in the system at any given point in time. This provides a counterpoint to the graph of the mean delay of all jobs over time, which is influenced by periods when many jobs were highly delayed while many more jobs were introduced into the system and ran relatively quickly. Similar to the graphs of mean delay and dilation over time, however, this graph does not vary within the GLUnix and FIFO schedulers.

- **Histogram: Job Bids**. This graph simply shows the distribution of job bids across the full range of possible bids. Each rectangle represents a bid range of 5.0 units; the vertical axis is capped at a count of 1000.0 to prevent a great number of jobs with similar bids (such as zero) from making the rest of the graph unreadable. This histogram will, obviously, vary across different User modules, but will not be altered by the various Schedulers used.

- **Histogram: Job Delay**. This graph shows the distribution of job delays across the full range of possible delays. Each rectangle represents a range of delays of 5.0 minutes; the vertical axis is capped at a count of 500.0 to prevent a great number of jobs with similar delays (such as zero) from making the rest of the graph unreadable. Within the FIFO scheduler and within the GLUnix scheduler, this histogram will not change across User modules, because these schedulers are not affected by job bids; however, the Vickrey (Economic) scheduler will produce a different resulting histogram for each User module.
- **Histogram: Job Dilation**. This graph shows the distribution of job dilations across a range of dilations from 100% to 1000% (10x standalone runtime). Each rectangle represents a range of dilation of 5%; the vertical axis is capped at a count of 600.0 to prevent a great number of jobs with similar dilations (such as 100%) from making the rest of the graph unreadable. Similar to the histogram of job delay, this will vary across User modules only when combined with the Vickrey scheduler, and will remain constant across all User modules for the FIFO scheduler and for the GLUnix scheduler.

When presenting results, we will, in general, concentrate on the graphs of job delay and dilation vs. job bid; these present the effect that bids have upon the scheduling of jobs in the system. Graphs of delay and dilation over time are, however, useful for visualizing the way in which a particular scheduler and User module combine to handle the dramatic spikes in cluster load that occur over time.

7.2.2.3 Values

Finally, we consider the statistics used to analyze the effectiveness of the various schedulers and User modules.

- **Center of Mass: Job Delay vs. Job Bid**. This is simply the weighted average of all points on the graph of Job Delay vs. Job Bid — that is, if the delay of each point is *d* and the bid of each point is *b*, the computed value is $\frac{\sum d \cdot b}{\sum b}$. This value is thus a measure of the correlation between bid and delay; the lower this value, the higher the correlation. Jobs with high bids and high delays — those jobs an economic scheduler should avoid causing — tend to increase the resulting value a great deal, while those with high bids and low delays or low bids and high delays — both quite acceptable to an economic scheduler — do not.

- **Center of Mass: Job Dilation vs. Job Bid**. This value is formed exactly as above, except that job dilation is substituted for job bid. Typically, this substitution tends to produce considerably higher values than the above measurement, because none of the schedulers written have any bias towards shorter jobs — which are the jobs most likely to contribute toward high job dilations. Indeed, because most jobs in the system are so short (96.2% under five minutes; 50% under seven *seconds*) and thus can easily achieve very high dilations due to relatively small delays in their runtime, the scheduler may, conceptually, actually be doing a very good job — a delay of twelve seconds to run even a job only a few seconds long is not typically a great problem for a user — and yet produce quite high job dilation values.

7.2.2.4 User-Invariant Graphs

We first consider the graphs that remain invariant across User modules. These are the graphs of mean delay, mean dilation, and total dilation over time for the GLUnix and FIFO schedulers, and the histograms of job delay and job dilation for the GLUnix and FIFO schedulers.

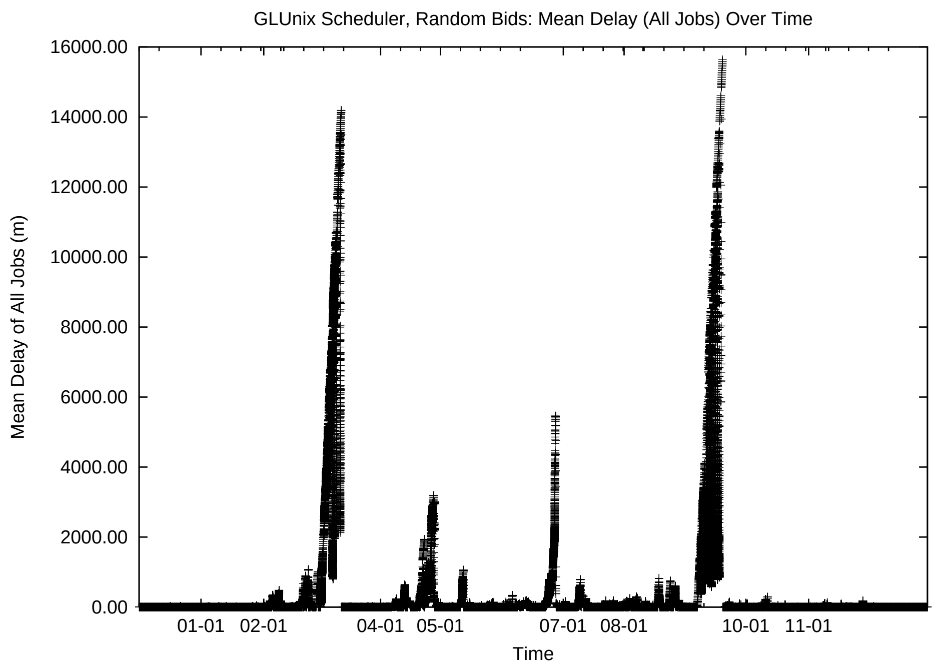


Figure 7.2-F. Mean delay over time, GLUnix scheduler.

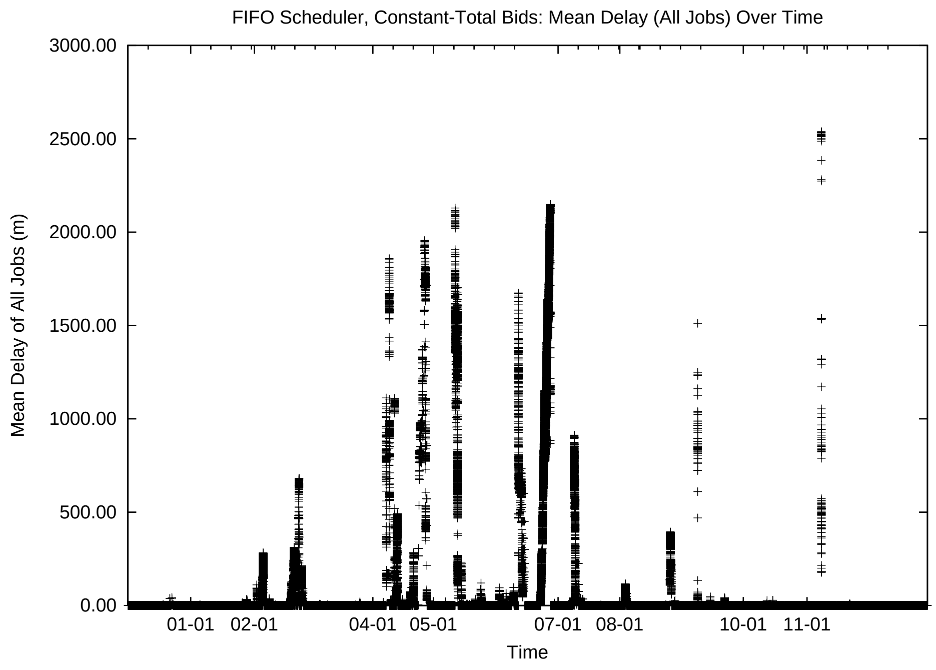


Figure 7.2-G. Mean delay over time, FIFO scheduler.

Figure 7.2-F shows the mean delay of each job in the system for the GLUnix scheduler over the time period studied, while Figure 7.2-G shows the same for the FIFO scheduler. These schedulers perform exactly as expected: during periods of high contention for the system, delays for each job increase without bound. At times, the average delay to run a job under a system using the FIFO scheduler approached two full days. Clearly, given the behavior typical of our target population (often students executing jobs for homework or paper submission deadlines less than a day or two away), these sorts of delays will cause major problems for the average user.

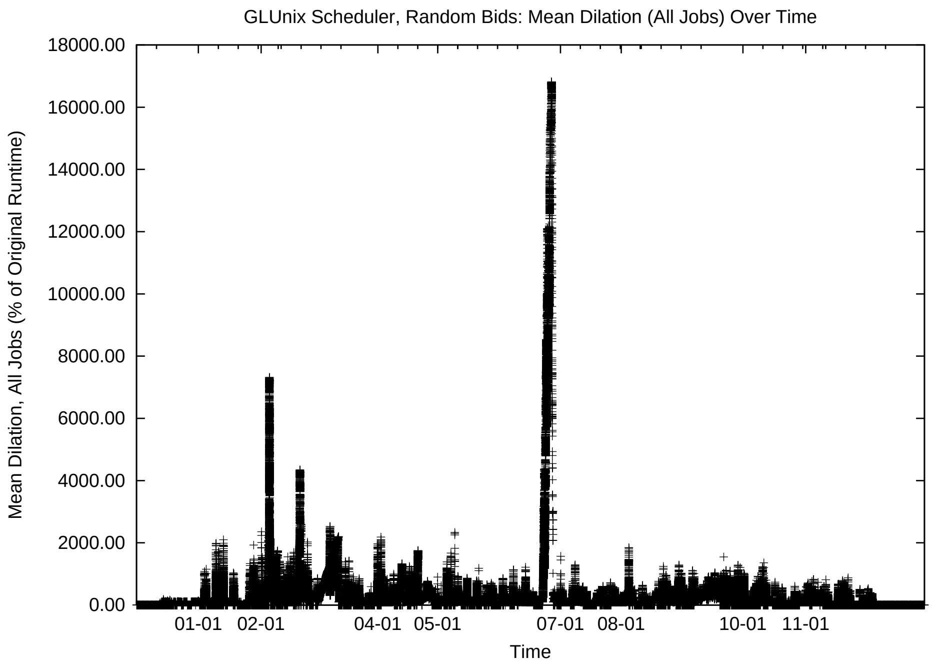


Figure 7.2-H. Mean dilation over time, GLUnix scheduler.

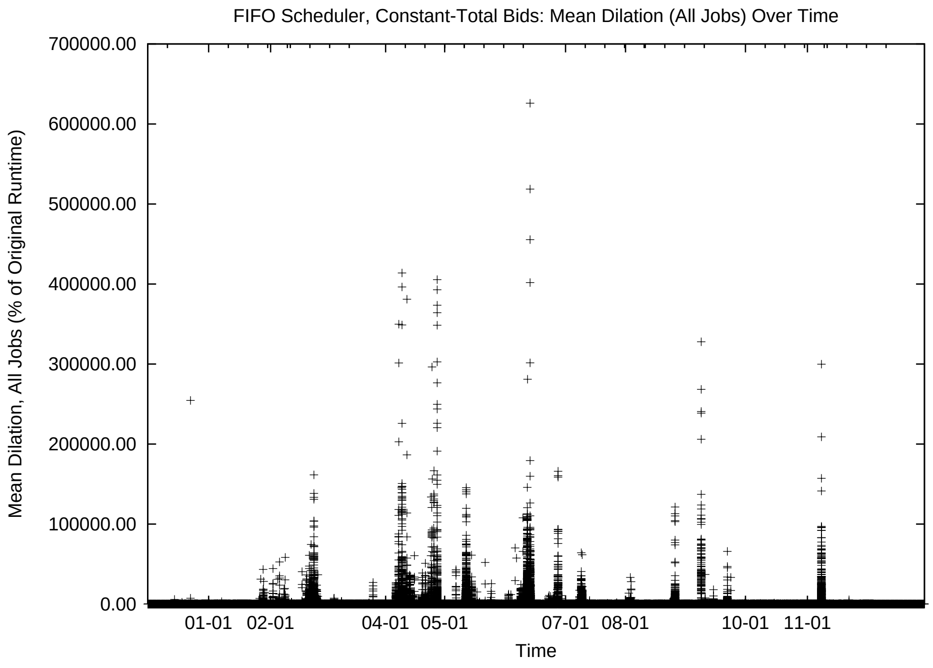


Figure 7.2-I. Mean dilation over time, FIFO scheduler.

Figure 7.2-H shows the mean dilation of all jobs in the system, over time, for the GLUnix scheduler; Figure 7.2-I shows the same data for the FIFO scheduler. These graphs further demonstrate the extreme dilation caused by both the GLUnix and FIFO schedulers; apparent, however, is a fundamental difference between them: because the GLUnix scheduler starts all jobs immediately, short jobs, even under heavy load, don't get a chance to experience truly huge delays, while the FIFO scheduler — treating long jobs and short jobs identically — sometimes causes short jobs to wait over six hundred times their original duration before they complete.

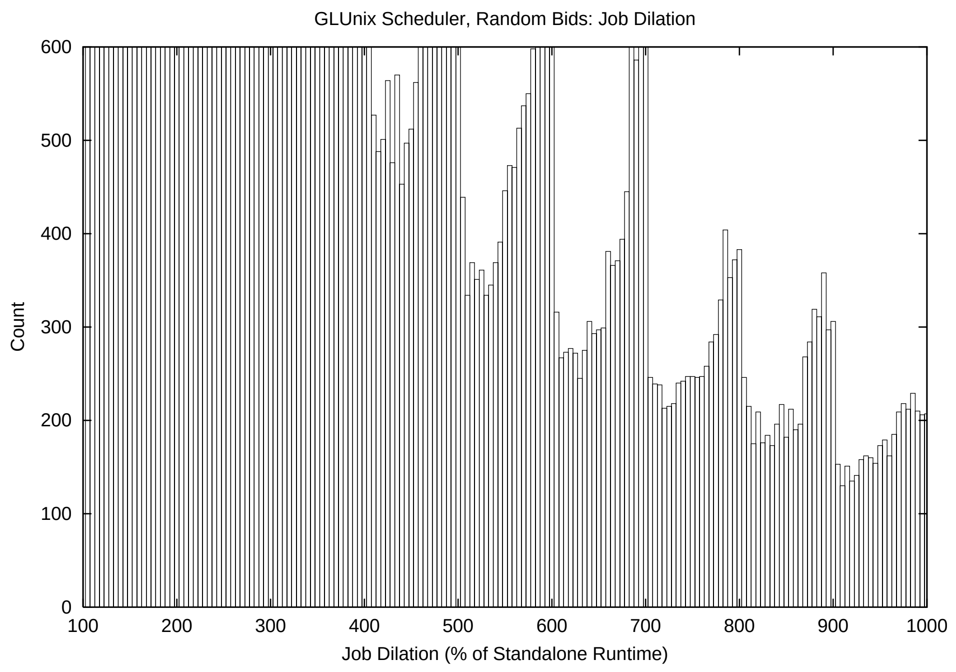


Figure 7.2-J. Histogram of job dilation, GLUnix scheduler.

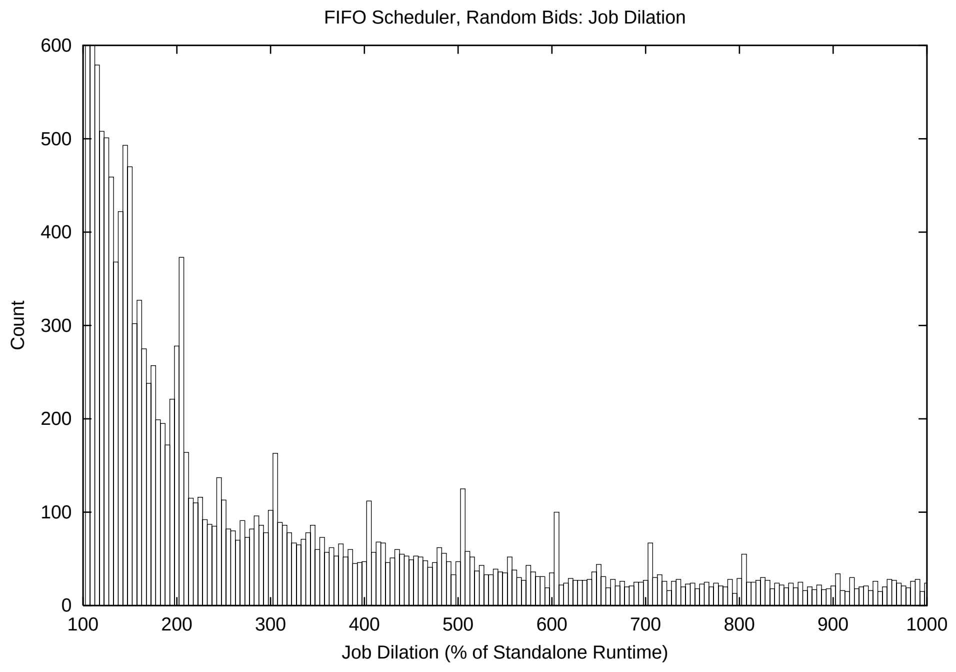


Figure 7.2-K. Histogram of job dilation, FIFO scheduler.

We turn now to the histograms of job dilation for the FIFO and GLUnix schedulers, as shown in Figure 7.2-J (GLUnix scheduler) and Figure 7.2-K (FIFO scheduler). Visual comparison of these histograms produces an immediately discernable result: while the FIFO scheduler can at times produce job dilation values much greater than the GLUnix scheduler, it also produces *fewer* job dilations greater than zero: because the GLUnix scheduler runs all jobs immediately, jobs incur dilation due to either jobs preceding them or following them (up to a job's own runtime), while the FIFO scheduler only incurs dilation if a job is itself preceded by jobs that prevent it from running when submitted.

7.2.2.5 Analyzing Job Delay and Job Dilation vs. Bid

A principal difficulty in analyzing the data collected of job delay and dilation vs. bid is the lack of a precise mathematical tool for demonstrating what is meant by "jobs with higher bids have lower delays". Because the correlation between bid and delay only becomes apparent when the cluster is oversubscribed — and, indeed, only becomes truly pronounced when demand far outstrips supply — straightforward correlation calculations between job bid or dilation and bid fail to adequately capture the true nature of the graphs.

Instead, we consider an intuitive and revealing composite graph and analyze the resulting statistics. We wished to create a set of line segments that captured the concept of the "upper bound" of the scatter plot of job delay or dilation vs. bid; however, this term is left loosely defined. A traditional convex hull, for example, would allow outlying points near the top of the graph to completely dominate the shape of this set of line segments.

Rather than calculate a convex hull, then, we constructed a set of line segments using the following procedure:

- The points at the extreme left and right edges of the graph are compared; whichever contains the lowest value becomes the starting point.
- The graph is then traversed, either from left-to-right or from right-to-left.
- At each x-coordinate, we choose the point with the highest y-coordinate.
- We construct a line segment from the last endpoint to this point with the highest y-coordinate.
- If the slope of the line (following the left-to-right or right-to-left ordering established in the first step) is positive, zero, or greater than a fixed lower bound (in our case, –3.0), we use this new point as the new endpoint of the line segments under construction.

- Otherwise, we proceed to the next *x*-coordinate using the ordering determined in the first step.

This procedure creates a set of line segments that represents the "upper bound" of a graph. The limitation on negative slope imposed avoids rapid vertical oscillations in the graph: for example, when bids are chosen randomly by users, very often a job with a given bid will incur a delay *d*, but the single job with the next-higher bid will have been submitted during a period of oversupply, and thus have delay zero. Without this limitation on negative slope in the graph, the graph rapidly fluctuates between zero and various positive numbers.

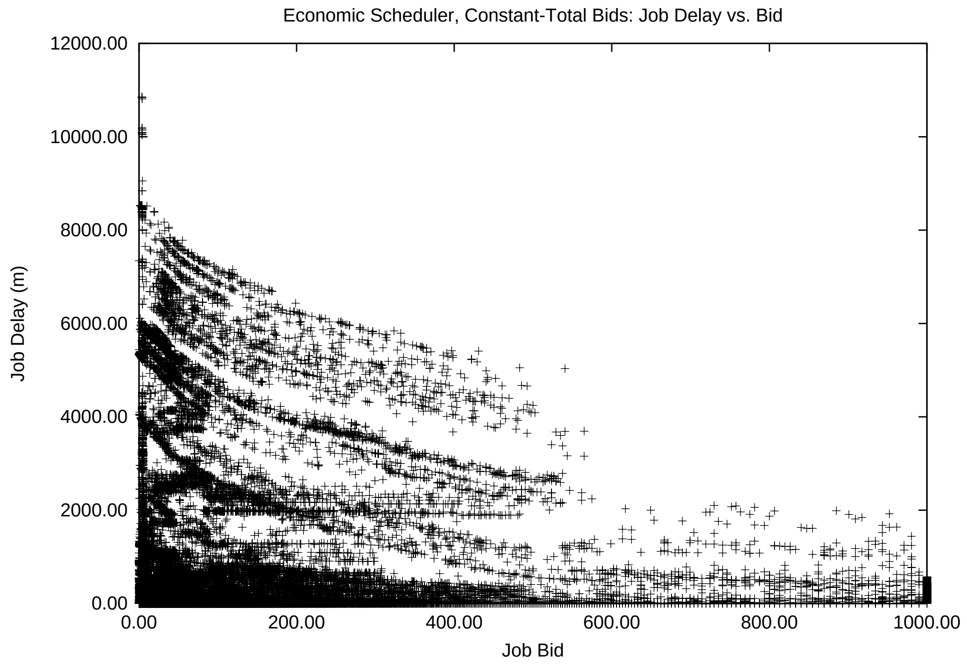


Figure 7.2-L. Job delay vs. job bid, Economic scheduler, Constant-Total-Bid users.

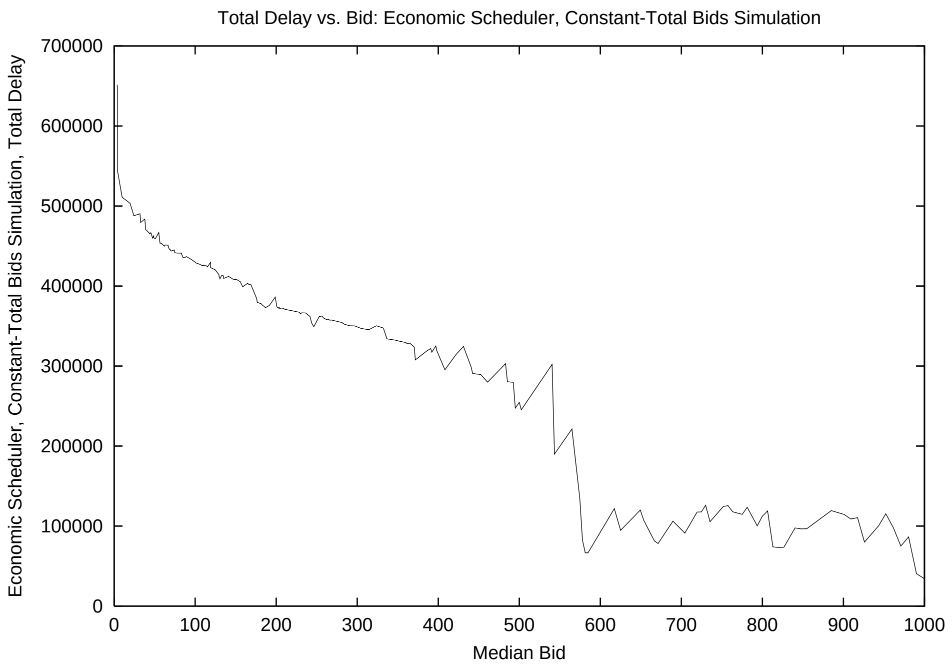


Figure 7.2-M. Job delay vs. job bid, Economic scheduler, Constant-Total-Bid users: constructed set of line segments.

Figure 7.2-L is a scatterplot of job delay vs. job bid for the Economic scheduler when using the Constant-Total-Bid user module; Figure 7.2-M is the set of constructed line segments for this graph. (The two graphs have different vertical scales due to the exact measurement statistic used, but have the same shape.) It is obvious from the graphs that the set of line segments constructed does accurately represent an intuitive "top edge" of the graph.

This procedure is now repeated for two graphs: the *numerator* graph — in this case, the graph that represents the Vickrey scheduler's behavior, and a *denominator* graph — a graph that represents either the FIFO scheduler's behavior or the GLUnix scheduler's behavior.

Once a numerator graph and a denominator graph have been constructed, these graphs are then divided in a piecewise-linear fashion: at any *x*-coordinate that is the start or end of a line segment in either the numerator or denominator graph, the ratio of the current value (along the line segments) of the numerator graph to the current value (along the line segments) of the denominator graph is calculated. This then becomes a point in the output graph.

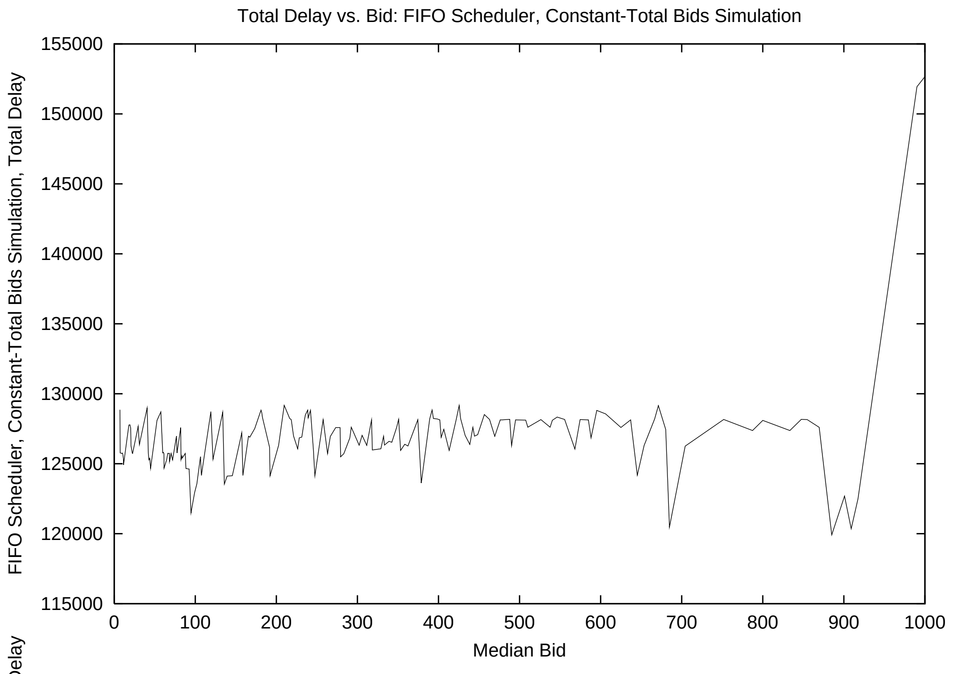


Figure 7.2-N. Job delay vs. job bid, FIFO scheduler, Constant-Total-Bid users: constructed set of line segments.

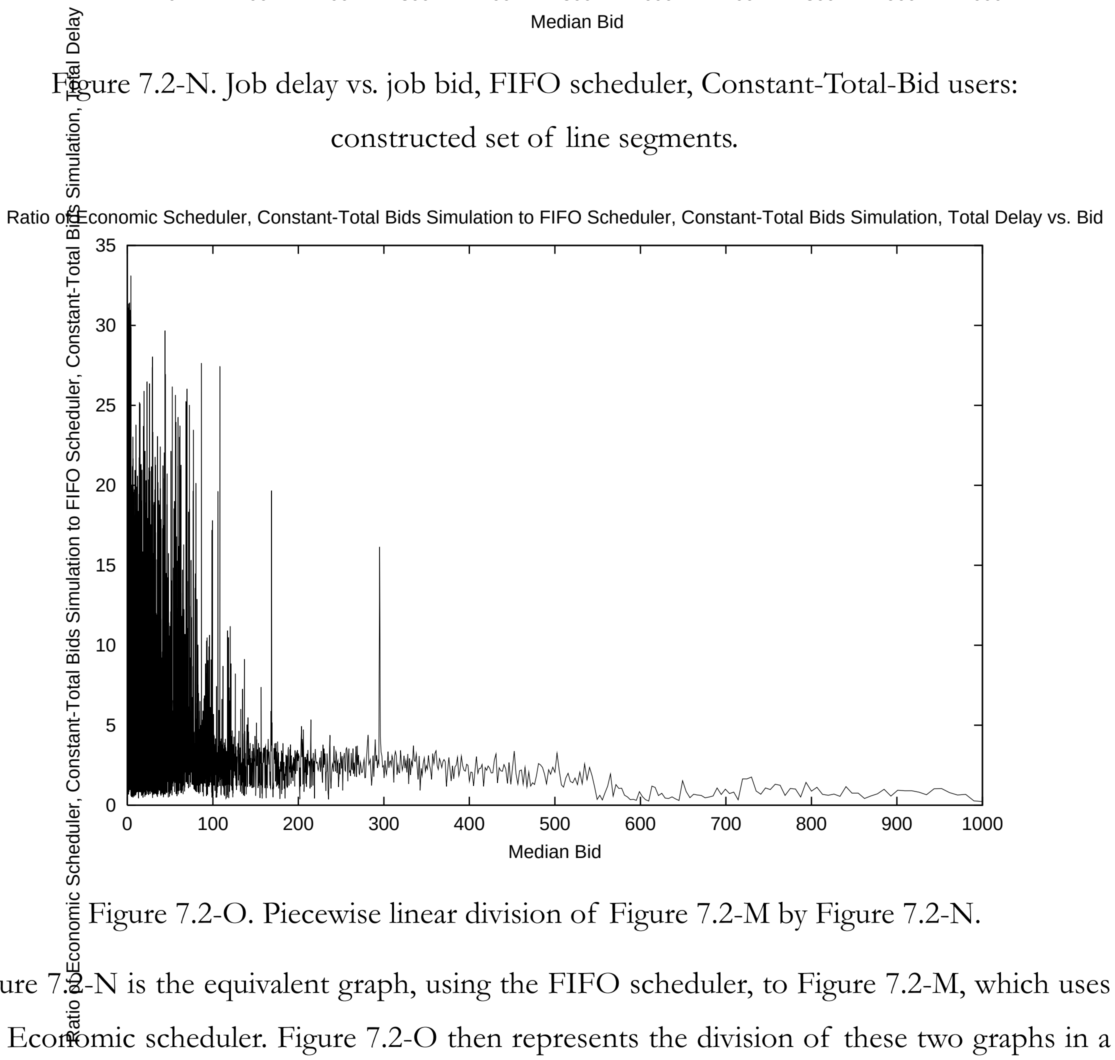


Figure 7.2-O. Piecewise linear division of Figure 7.2-M by Figure 7.2-N.

Figure 7.2-N is the equivalent graph, using the FIFO scheduler, to Figure 7.2-M, which uses the Economic scheduler. Figure 7.2-O then represents the division of these two graphs in a piecewise-linear fashion. While we will consider the meaning of the exact data involved later, this graph immediately and intuitively presents the advantage of using the Vickrey scheduler over the FIFO scheduler for jobs of various bids; for example, it is obvious that for jobs bidding above about 570, the total delay experienced was shorter under the Vickrey scheduler than under the FIFO scheduler (the desired effect), while jobs bidding close to zero experienced delays of up to four times as long under the Vickrey scheduler as under the FIFO scheduler.

This division of the graphs is now performed for each User module for all three graphs — the graphs using the Vickrey, FIFO, and GLUnix schedulers. By visually inspecting the graphs of job bid and delay vs. job bid, a clear picture of the immediate results of each Scheduler can be drawn; by examining the graph of the ratio of delay incurred by the GLUnix or FIFO schedulers to that of the Vickrey scheduler, a clear picture of the reduction in delay (or, for lower bids, increase in delay) by using the Vickrey scheduler can be drawn.

We turn now to an analysis of the scheduling behavior of the Vickrey scheduler (as contrasted to the FIFO and GLUnix schedulers) when given a set of jobs with bids assigned under three distinct user bidding schemes that we believe are most representative of users' actual behavior.

7.2.2.6 Constant-Total-Bid User

The Constant-Total-Bid User places bids on jobs such that the product of job duration, job parallelism, and bid is a constant (in our case, 1000). This strategy assumes that preventing shorter jobs from having substantial delays is most important to the user, and thus represents an environment in which users are willing to spend credits to allow their short jobs to pre-empt the "background", long-running jobs on the cluster.

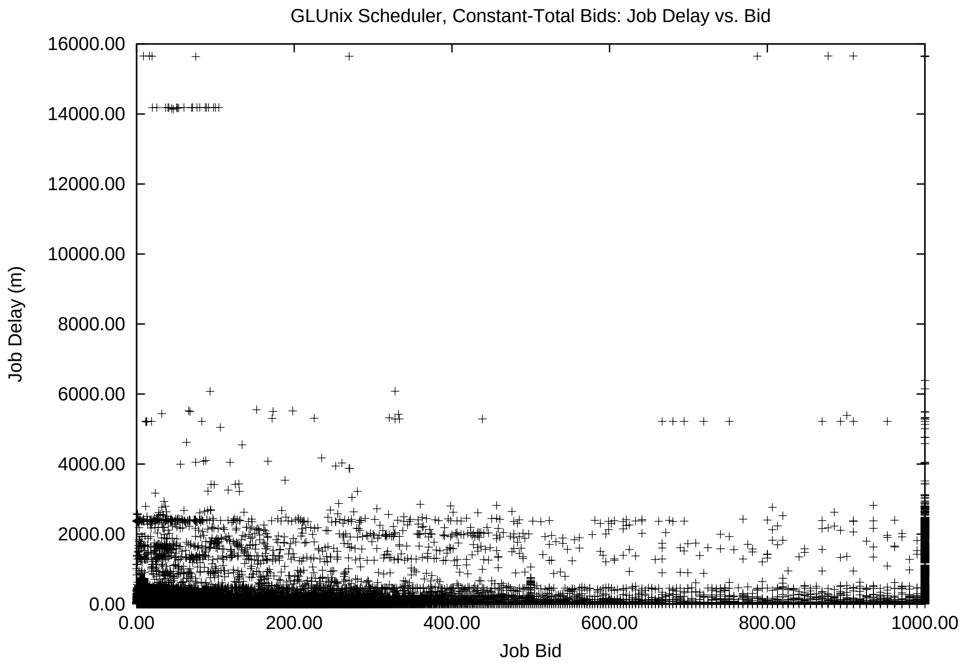


Figure 7.2-P. Job delay vs. bid, all jobs, GLUnix scheduler, Constant-Total-Bid User module.

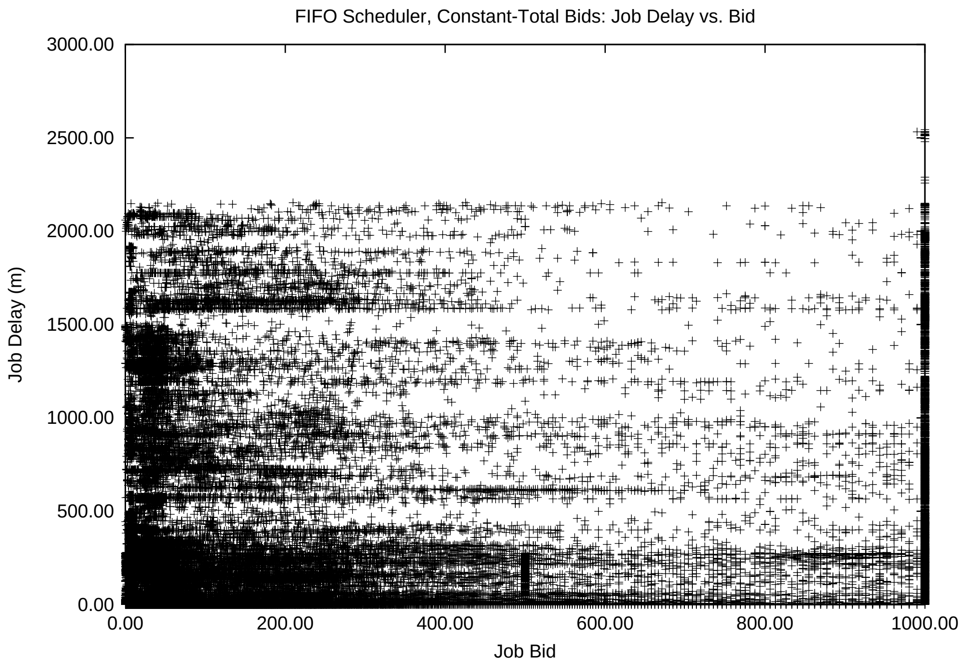


Figure 7.2-Q. Job delay vs. bid, all jobs, FIFO scheduler, Constant-Total-Bid User module.

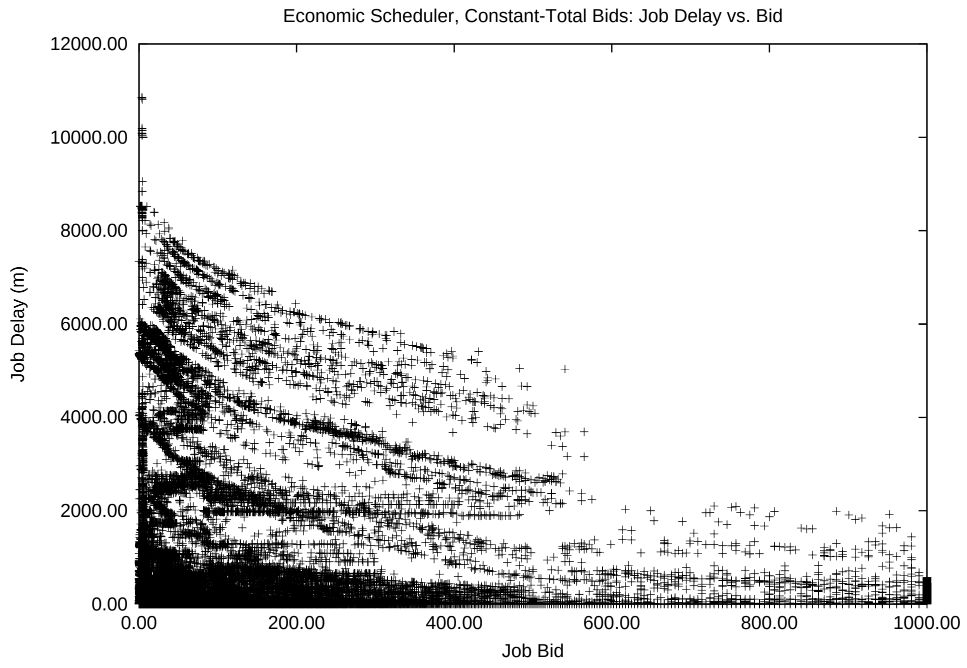


Figure 7.2-R. Job delay vs. bid, all jobs, Vickrey economic scheduler, Constant-Total-Bid User module.

Figure 7.2-P, Figure 7.2-Q, and Figure 7.2-R show the relationship between job delay and job bid for the three schedulers used in the simulated system. Immediately apparent is the lack of correlation between job bid and job delay for the FIFO and GLUnix schedulers, as expected; for example, under the FIFO scheduler, some jobs bidding 1000 credits per node per minute (the maximum bid) nevertheless were delayed just as much as jobs bidding zero. The GLUnix scheduler performed similarly to the FIFO scheduler, giving no precedence to jobs with high bids. Under the economic scheduler, however, jobs with high bids were delayed only a quarter as much as those jobs with high bids; by taking advantage of the bidding system, users operating under this pattern and scheduler would have been able to significantly decrease the expected delay of their jobs.

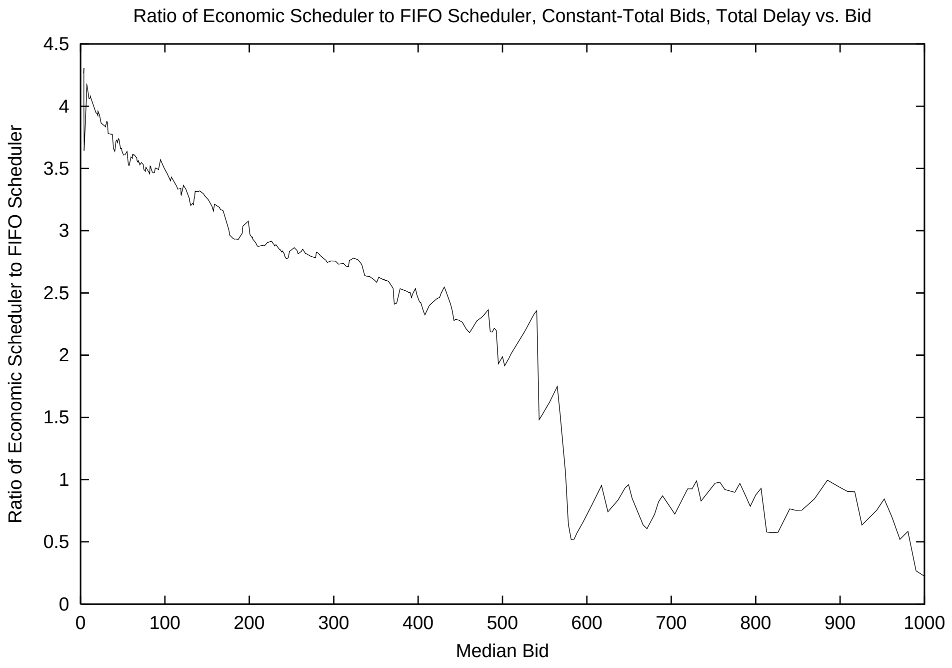


Figure 7.2-S. Ratio of delays, Vickrey economic scheduler and FIFO scheduler, for users' jobs bid using the Constant-Total-Bid bidding scheme.

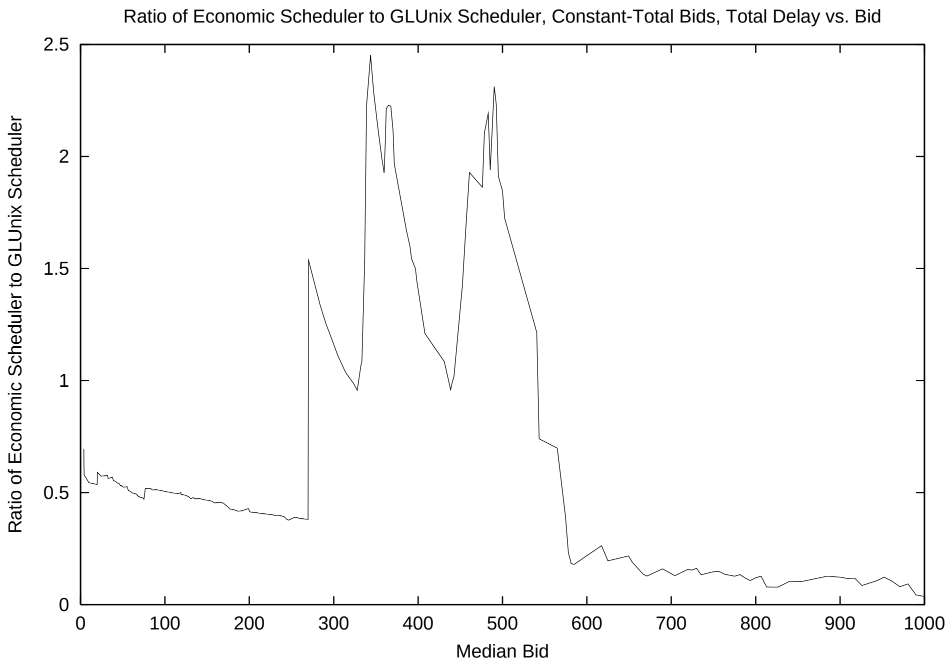


Figure 7.2-T. Ratio of delays, Vickrey economic scheduler and GLUnix scheduler, for users' jobs bid using the Constant-Total-Bid bidding scheme.

Figure 7.2-S demonstrates the ratio of delay given to jobs by the Vickrey economic scheduler to that given by the FIFO scheduler for the entire range of bids presented to the system; Figure 7.2-T similarly presents the ratio of the Vickrey economic scheduler to that of the GLUnix scheduler over the same range. The graph of the ratio of the Vickrey scheduler to that of the FIFO scheduler clearly demonstrates the potential benefits to the user of using an economic scheduling algorithm: by varying a job's bid, a user could predictably control the delay that job received by a factor of four, increasing it substantially at low bids and decreasing it substantially at high bids. The GLUnix scheduler's algorithm, meanwhile, produces a few outlying points that give its graph a peculiar spike (due to coincidentally small delays from the GLUnix scheduler) in the range of bids from about 300 to about 600, it nevertheless shows the same downward trend as the graph of the FIFO scheduler's ratio: by increasing a job's bid, the user could predictably control the delay that the job received by a similar factor of four to five.

Table 7.2-F. Centers of mass, Constant-Total-Bid User bidding pattern, Vickrey, FIFO, and GLUnix schedulers.

Scheduler	**Center of Mass, Job Delay vs. Bid**	**Center of Mass, Job Dilation vs. Bid**
Vickrey	132.9	119.5
FIFO	466.0	444.8
GLUnix	507.1	479.5

Table 7.2-F shows the calculated center of mass (see section 7.2.2.3) for the Vickrey, FIFO, and GLUnix schedulers for the graphs of job delay vs. bid and job dilation vs. bid. Simply from these measurements, it becomes evident that the Vickrey scheduler performs easily as expected: because it essentially avoids imparting high delays to those jobs with high bids, the center of mass of the graph moves quite significantly towards the left of the graph. The average "delay-bid product" for the Vickrey scheduler is only a quarter that of the other schedulers.

7.2.2.7 Random-Bid User

The Random-Bid User places bids on jobs entirely at random: no immediately observed characteristic of a job has any correlation with its bid. This is not to say that the random-bid user is intended solely as a model of those users who bid entirely at random (although experience shows that such users will indeed be present), but rather as a model of those users whose inherent valuation of jobs is based on characteristics that the system cannot observe.

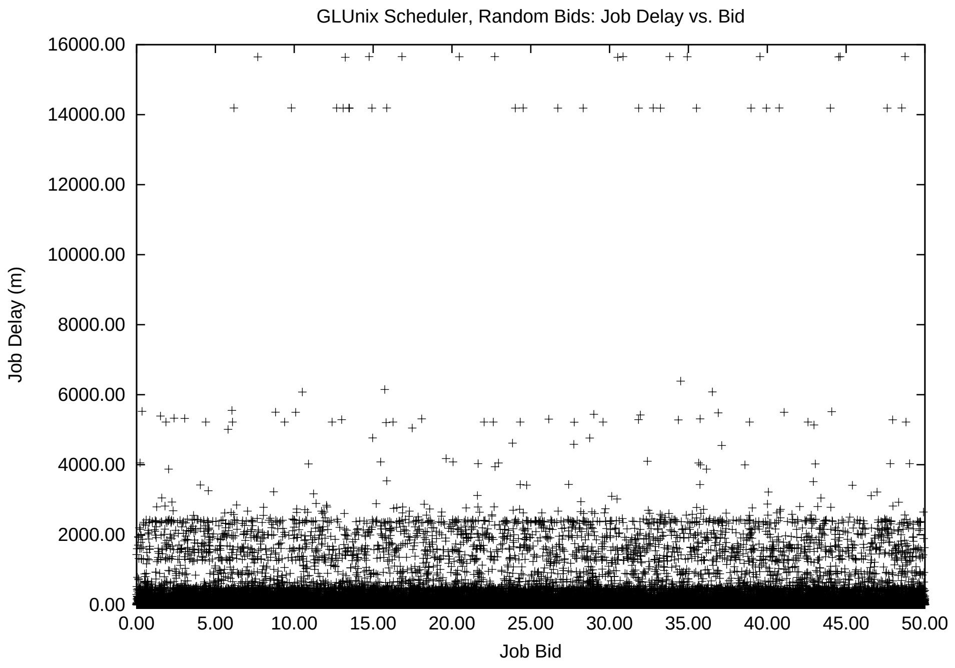


Figure 7.2-U. Job delay vs. bid, all jobs, GLUnix scheduler, Random-Bid User module.

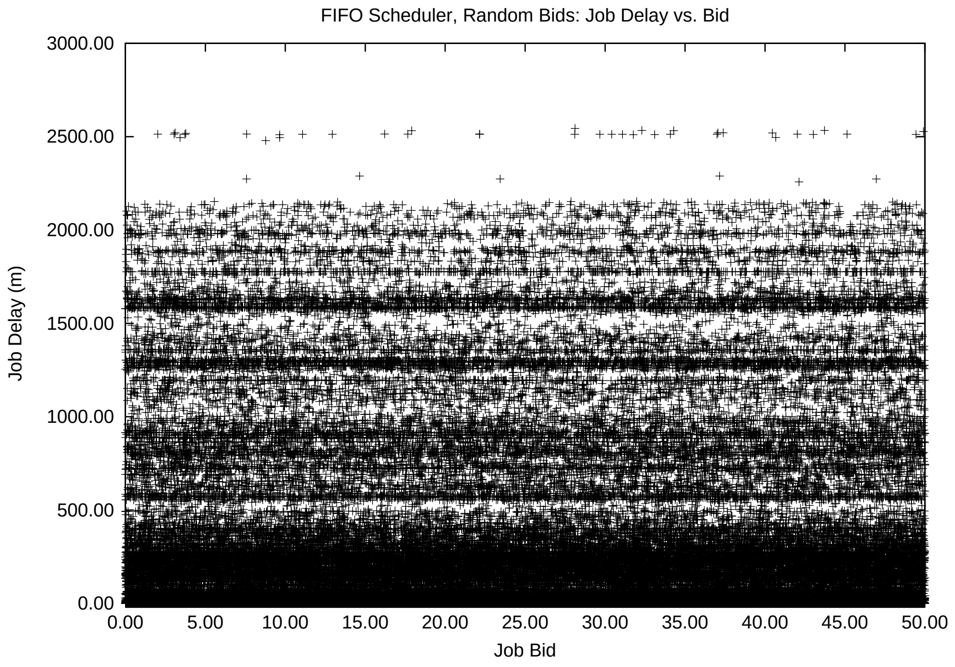


Figure 7.2-V. Job delay vs. bid, all jobs, FIFO scheduler,

Random-Bid User module.

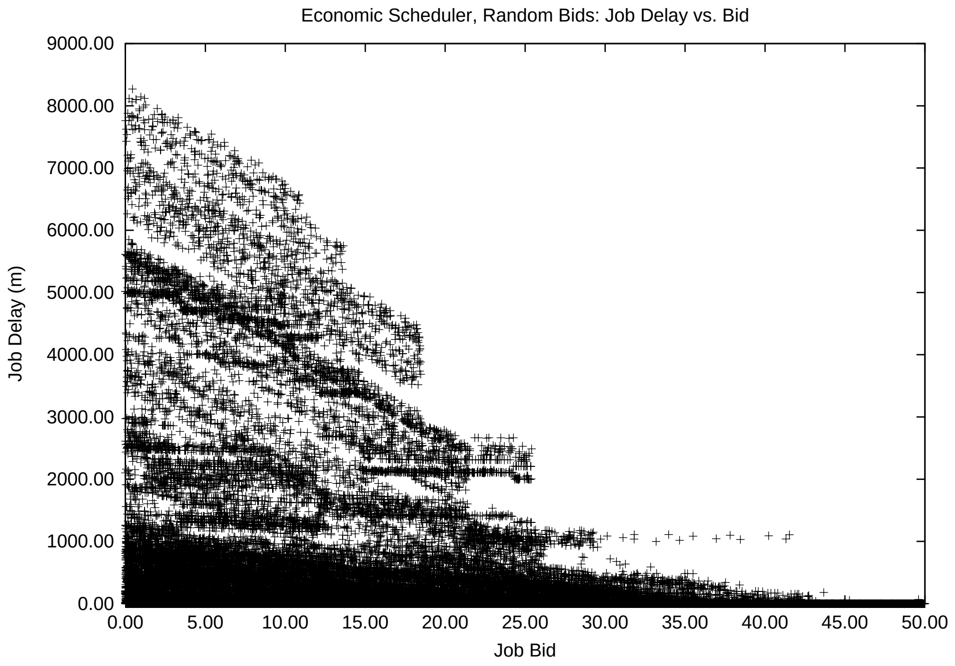


Figure 7.2-W. Job delay vs. bid, all jobs, Vickrey scheduler,

Random-Bid User module.

Figure 7.2-U, Figure 7.2-V, and Figure 7.2-W again show the relationship between job delay and job bid for the three schedulers used in the simulated system. To an even greater degree than is apparent under the Constant-Total-Bid User bidding scheme, the difference between the FIFO or GLUnix scheduler and the Vickrey scheduler is apparent: the FIFO and GLUnix schedulers have no correlation whatsoever between bid and job delay, while the Vickrey scheduler shows a strong trend towards a traditional reciprocal (1/*x*) graph. Indeed, a closer examination shows that jobs with bids above the mean (25.0) were delayed, at maximum, only one-eighth as long as jobs with bids near zero. This result is especially encouraging, because it indicates that users were able to greatly affect the amount of delay imparted to their jobs simply by ensuring their job's bid was above average.

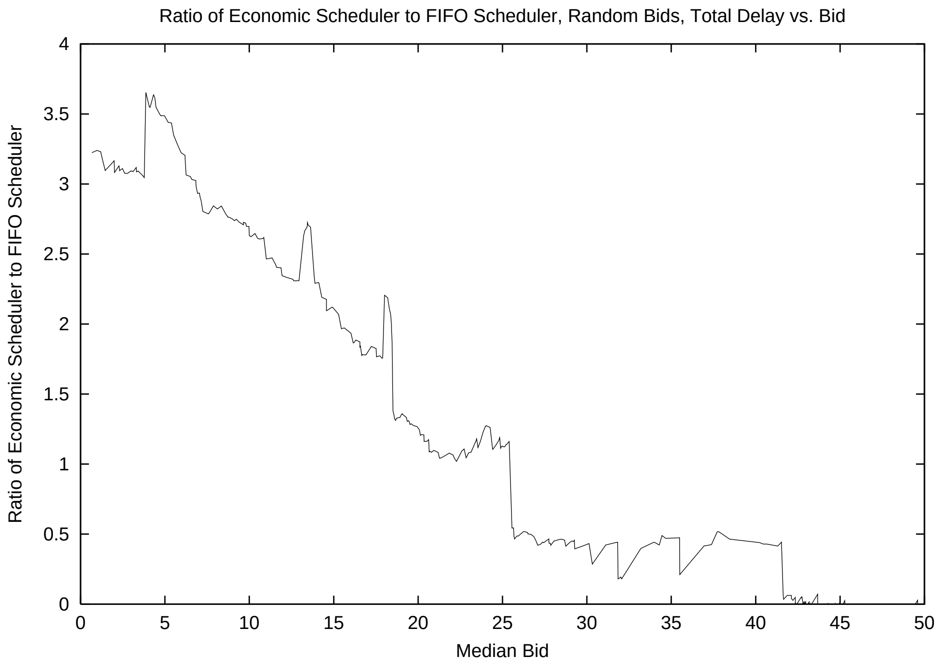


Figure 7.2-X. Ratio of delays, Vickrey economic scheduler and FIFO scheduler, for users' jobs bid using the Random bidding scheme.

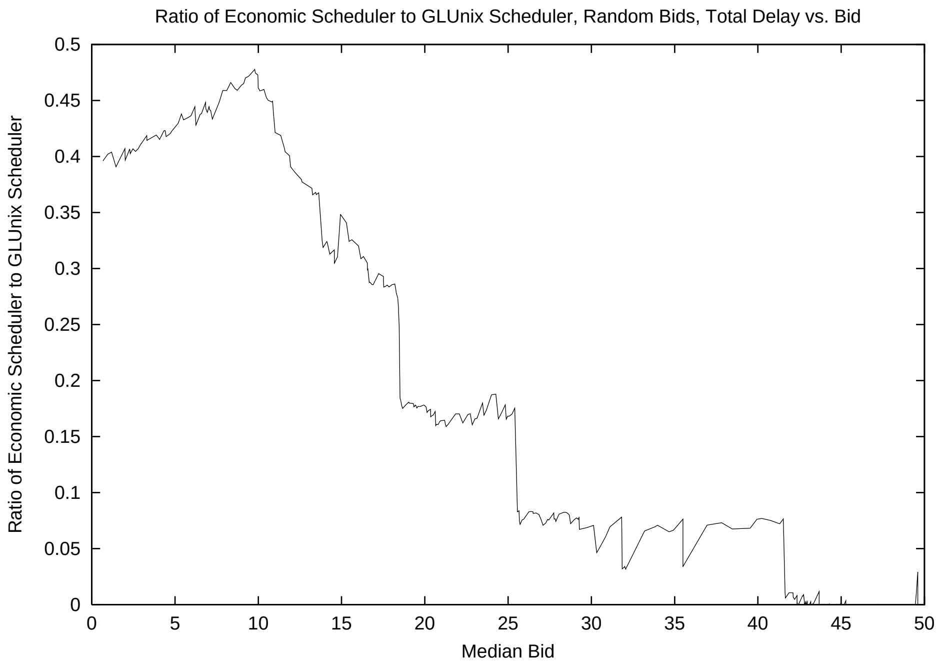


Figure 7.2-Y. Ratio of delays, Vickrey economic scheduler and GLUnix scheduler, for users' jobs bid using the Random bidding scheme.

Figure 7.2-X displays the ratio of job delay imparted by the Vickrey scheduler to that imparted by the FIFO scheduler across the entire range of bids presented; Figure 7.2-Y displays the same data, but compares the Vickrey scheduler to the GLUnix scheduler. These graphs demonstrate quite clearly the effectiveness of the Vickrey scheduler. When compared to the FIFO scheduler, the Vickrey scheduler imparted greater delays (by up to a factor of 3.5) to those jobs with low bids, while imparting much shorter delays (approaching nearly zero) to those jobs with high bids. Comparisons with the GLUnix scheduler show a very similar pattern, if a different scale (due to the GLUnix scheduler's inherent disadvantages when using this measure of "job delay" as previously noted in section 7.2.2.1).

Table 7.2-G. Centers of mass, Random-Bid User bidding pattern, Vickrey, FIFO, and GLUnix schedulers.

Scheduler	**Center of Mass, Job Delay vs. Bid**	**Center of Mass, Job Dilation vs. Bid**
Vickrey	9.08	8.98
FIFO	24.93	24.97
GLUnix	25.26	24.91

Table 7.2-G presents the corresponding measurements of the center of mass of the graphs of job delay and dilation vs. bid for the three schedulers under the Random-Bid User bidding pattern. Once again, the effect of the Vickrey scheduler is undeniable: while the GLUnix and FIFO schedulers have centers of mass near the actual center of the graph (25.0), the Vickrey scheduler produces a center of mass roughly one-third as much by eliminating high-bid, high-delay jobs.

7.2.2.8 Proportional-Bid User

The Proportional-Bid User is, in some sense, the exact opposite of the Constant-Total-Bid User: rather than ensuring a constant total product of job duration, job parallel degree, and bid, the Proportional-Bid User assigns a bid directly proportional to the inherent duration of the job (that is, the duration the job would have were it running entirely alone on the cluster). This module, then, attempts to simulate those users for whom small "debugging" runs of jobs are relatively unimportant, but for whom long-duration "production" runs are very important and should be interrupted as little as possible.

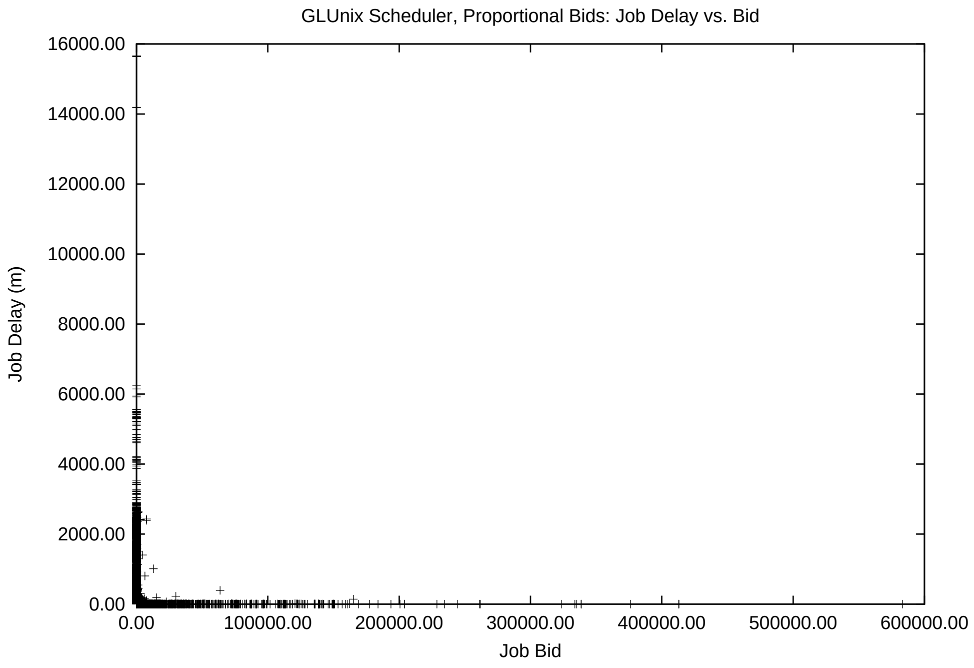


Figure 7.2-Z. Job delay vs. bid, all jobs, GLUnix scheduler, Proportional-Bid User module.

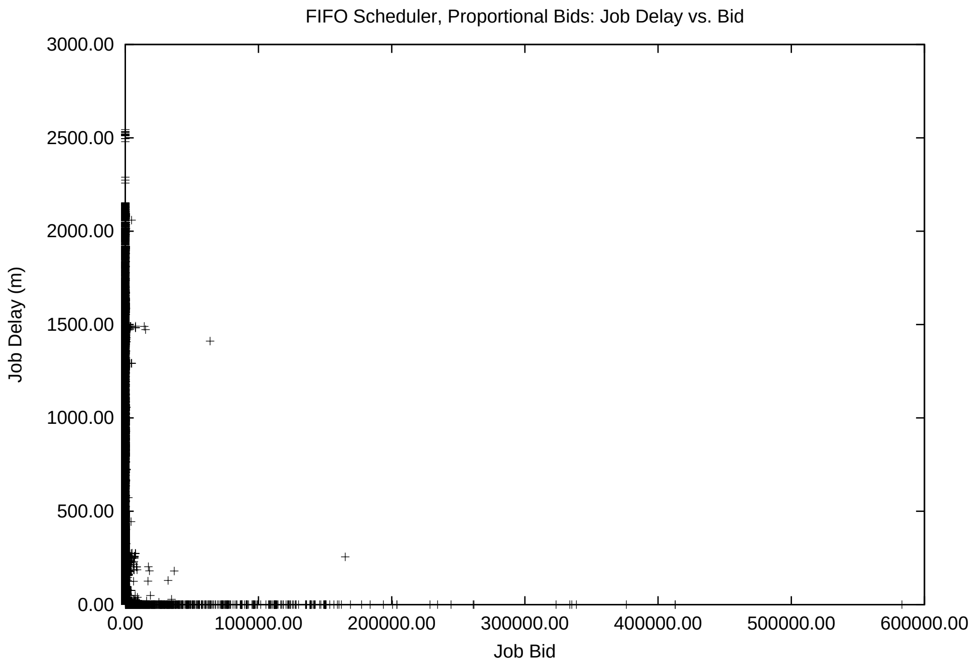


Figure 7.2-AA. Job delay vs. bid, all jobs, FIFO scheduler, Proportional-Bid User module.

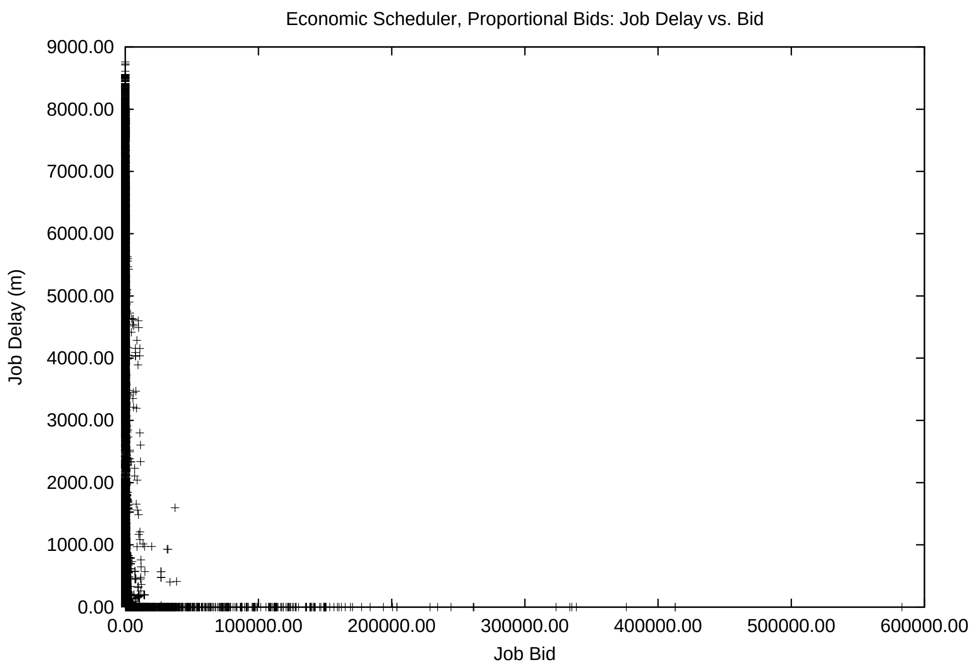


Figure 7.2-BB. Job delay vs. bid, all jobs, Vickrey scheduler, Proportional-Bid User module.

Figure 7.2-Z presents a scatterplot of job delay vs. job bid for the GLUnix scheduler, when presented with a set of jobs whose bids have been assigned by the Proportional-Bid User module; Figure 7.2-AA presents the same when scheduled by the FIFO scheduler, and Figure 7.2-BB presents the results when the same data is scheduled by the Vickrey scheduler. Immediately obvious is the fact that the range of bids is vastly wider than those in the past two examples; because a few jobs are extremely long (see 7.2.1.1), the range of bids is enormous. Further, because the vast majority of jobs are also very short, nearly all jobs cluster essentially on the Y-axis with bids near zero when compared to the few long-running jobs with enormous bids.

When contrasted to the graphs presented with the previous two user bidding schemes, the data here shows a much smaller difference between the various schedulers. In particular, the only obvious difference between the FIFO and Vickrey scheduler is the presence of a few outliers in the graph of the FIFO scheduler; similarly, the graphs of the GLUnix and Vickrey scheduler are distinguishable only by the presence of a very few outlying points.

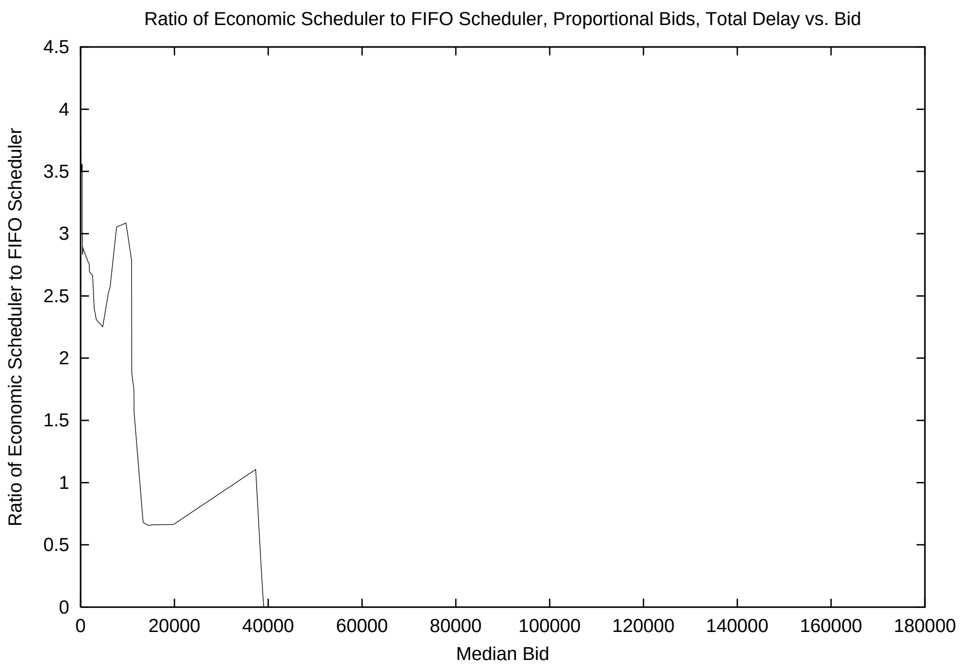


Figure 7.2-CC. Ratio of delays, Vickrey economic scheduler and FIFO scheduler, for users' jobs bid using the Proportional bidding scheme.

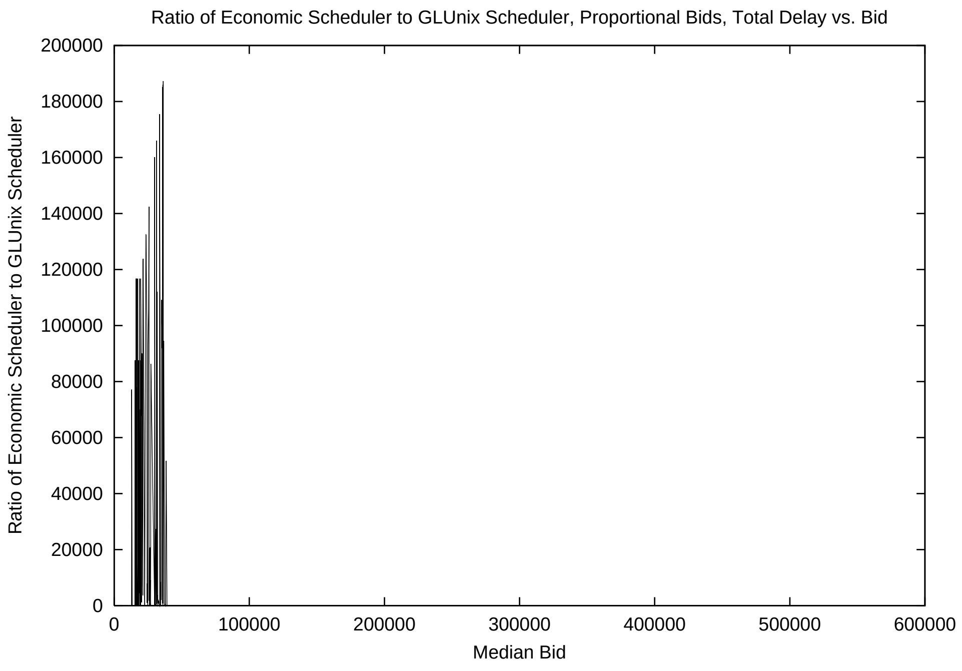


Figure 7.2-DD. Ratio of delays, Vickrey economic scheduler and GLUnix scheduler, for users' jobs bid using the Proportional bidding scheme

Similarly, Figure 7.2-CC, the graph displaying the ratio of the Vickrey scheduler's performance to the FIFO scheduler's performance on job delays, is significantly less compelling than that presented for the Constant-Total-Bid or Random-Bid User modules. Figure 7.2-DD presents the same ratio but for the Vickrey scheduler to the GLUnix scheduler; its data is so lopsided as to only indicate that for several jobs with bids between about 10,000 and about 40,000, the Vickrey scheduler actually *increased* their delays enormously.

Table 7.2-H. Centers of mass, Proportional-Bid User bidding pattern,
Vickrey, FIFO, and GLUnix schedulers.

Scheduler	**Center of Mass, Job Delay vs. Bid**	**Center of Mass, Job Dilation vs. Bid**
Vickrey	99.66	4.67
FIFO	53.92	42.80
GLUnix	53.66	79.79

Table 7.2-H presents the centers of mass of the various graphs associated with the Proportional-Bid User module. The data presented here offer two interesting points. First, the center of mass for the Vickrey scheduler on the graph of job delay vs. bid is actually *higher*, by a factor of nearly two, than it is for the FIFO or GLUnix schedulers. Second, the center of mass for the Vickrey scheduler on the graph of job *dilation* vs. bid is much lower — by a factor of ten to twenty — than it is for the FIFO or GLUnix schedulers.

Straightforward analysis of the nature of the various schedulers explains these seemingly odd results. A graph of center of mass is naturally influenced most by those jobs with the highest masses (bids) — under a Proportional-Bid User module, these are those jobs with long runtimes. Because these jobs have high bids, they tend to not be preempted much when compared to their overall runtimes, and thus accrue low dilation values; however, because they are very long-running, even a small dilation creates a large amount of absolute delay in the system.

7.2.2.9 Center of Mass

Finally, we present the results for the center of mass of the various graphs when using the various schedulers.

Table 7.2-I. Centers of mass, job dilation and delay vs. bid, for various Scheduler modules.

Graph	**Job Dilation vs. Bid**			**Job Delay vs. Bid**		
Scheduler	Vickrey	FIFO	GLUnix	Vickrey	FIFO	GLUnix
Binary Categorized	809.493	990.721	972.019	69.1116	981.446	984.880
Binary Random	466.480	851.958	848.034	496.043	848.899	850.768
Categorized	143.502	199.433	196.310	142.873	197.757	198.386
Constant-Total	119.477	444.793	479.538	132.939	466.005	507.132
Proportional	4.673	42.799	79.788	99.662	53.917	53.662
Random	8.981	24.969	24.912	9.0805	24.928	25.261
Zero	0.000	0.000	0.000	0.000	0.000	0.000

Except for the case presented in 7.2.2.8 — the center of mass of job delay vs. bid for the Proportional-Bid User — in every case the center of mass of the graph using the Vickrey economic scheduler is smaller than the center of mass for the corresponding FIFO or GLUnix scheduler's graph. As expected, the Vickrey scheduler does induce smaller delays in those jobs that have higher bids.

8 Conclusions

We now present the conclusions of our analysis of the pre-existing cluster scheduling system, GLUnix; the implementation and deployment of the PBS Vickrey economic scheduler; and the results obtained via the *econsim* simulator.

8.1 Make the user resource-aware

The most direct and most obvious result of our work is this: by making the user *aware* of the nonzero cost of resources, users may dramatically reduce their demand for resources. The pre-existing cluster scheduling system, GLUnix, presented the cluster to the user as, essentially, a very large SMP. While this model has many advantages in simplicity, familiarity, and immediacy, it also hides the true cost of resources from the user. Because clusters do not behave in the same way as traditional SMPs — they are essentially allocated on a per-node basis and cannot shift processes around with the same flexibility — users' requests did not take into account the true cost of their jobs, and thus the cluster rapidly became overloaded.

By introducing the PBS economic-queuing system, it was made apparent to users that resource use in the system was non-free: even though users actually could have demanded (and received!) far more resources than they actually did, their awareness that resources were bounded and fixed caused them to be far more conservative in their demands than they had previously been. Because they knew that demanding too many resources would result in a very visible and immediate effect (their job would be queued, instead of run), they demanded only as many resources as they actually required.

We therefore conclude that putting *any* sort of scheduling system in place that exposes the resource limitations of the underlying system can provide enormous benefits. By reducing demand on the cluster to that which is truly required, the entire cluster performs much better and users see a much more responsive and predictable system overall.

8.2 FIFO can cause very large latencies

A traditional approach to this issue would involve the installation of the simple, well-understood first-in, first-out (FIFO) queuing system onto the cluster. While this approach would indeed have the benefits explained above, it can also perform poorly in a highly dynamic and varied environment. Using the *econsim* simulator, we explored the effects a simple FIFO queue would have had, had it been installed on the Berkeley NOW over the yearlong period studied. At times, job latencies ranged into several days, meaning that even the shortest jobs would have taken days before even starting to run.

The FIFO scheduler does provide direct, immediate feedback to users on the state of the cluster and their own resource demands — users can see their job at the bottom of a queue of hundreds of other jobs. It also vastly improves the performance of parallel programs; because they run alone, the lack of true gang scheduling on most clusters does not cause the large slowdowns it does under time-shared systems. However, these improvements from the FIFO scheduler come at the expense of poor performance in certain situations; users' jobs may take days to run, even when they are short. The addition of even a small degree of time-sharing or suspension and resumption of long-running jobs would likely have a very positive impact on this scheduling.

Although various *ad hoc* algorithms have been developed to try to compensate for these deficiencies (*e.g.*, artificially boosting the queue position of particularly short jobs, or integrating fair-share algorithms), we believe that a true economic scheduler has the potential to resolve these issues more efficiently and effectively.

8.3 Economic queuing gives control and predictability

As compared to a GLUnix-style time-shared system or a traditional FIFO scheduler, an economic scheduling approach can provide several advantages.

Most importantly, economic schedulers can give users *control* over the delays experienced by their jobs. Using the traditional GLUnix scheduler, only an administrative reservation could affect the concurrent time-shared load (and thus delay) experienced by a job; using a traditional FIFO scheduler, only various *ad hoc* heuristics can allow the user any control over the queuing delay experienced by his or her jobs. Further, some of these *ad hoc* heuristics —

such as allowing users to tag only a certain number of jobs as "high priority" — actually trend towards approximating economic algorithms themselves.

By using an economic scheduler, however, users have the ability to, at any point, trade some of their limited supply of funds for an increase in priority of their job; alternatively, they may implicitly note that a particular job is of low importance by assigning few or no funds to it.

Further, an economic scheduler has substantial advantages in predictability. Under a GLUnix-style or FIFO-style scheduler, the total amount of delay a user's job experiences can vary wildly, from none at all when the cluster is in oversupply to several days' worth of delay when the cluster is very highly loaded. Unless the user is able to predict in advance how loaded the cluster may be — sometimes an easy task, sometimes not — the user has little idea of how long their job will actually take to run. Economic schedulers cannot eliminate this uncertainly entirely; however, especially for short jobs, they can reduce it: assuming users do not have vastly disparate funding levels, users with short jobs will almost invariably be able to outbid long-running jobs and thus preempt them.

This allows users with short or important jobs to exercise their rights to a fair share of the cluster, while allowing users with less-important jobs to willfully yield use of the cluster to other users' more immediate demands. While such control can be approximated using more traditional methods, the economic scheduler gives users a precise and predictable way to specify these desires.

8.4 Bidding patterns matter

Our extensive analysis of the behavior of various schedulers in light of several different user bidding patterns indicates that with each one of them, the Vickrey economic scheduler produces lower overall bid-delay products than a traditional FIFO or the existing GLUnix scheduler. However, the degree of success of such a scheduler is quite dependent on the exact nature of users' bids to the system. For example, when users' bids are assigned randomly distributed on an even interval, the Vickrey scheduler is able to perform exceedingly well, offering a nearly ten-to-one ratio of maximum job delay between the lowest and the highest bids. However, when user's bids are assigned using a proportional manner — that is, longer jobs get higher bids — the Vickrey scheduler performs significantly more poorly, allowing long delays to accumulate on some jobs.

We consider then that in a mature economically allocated system, the nature of users' bids to the system can profoundly affect its performance. In some cases, a Vickrey scheduler will perform very well; in others, a new type of economic scheduler may be required, or certain heuristics introduced.

8.5 Bids' currency matters

Based on our observations of users' behavior on the new economic-queuing system, especially their bidding patterns, we conclude that the lack of any *external* value of the currency ("credits") used for bidding causes users to bid much more arbitrarily and carelessly than they might otherwise. Because our system never truly found itself in a competitive mode — that is, demand nearly always remained *less* than supply — users were not interested in the bids they assigned to their jobs. Even when there was a small burst of competition, users usually simply assigned increasingly higher bids (1.0, 10.0, 200.0, 1000.0) to their jobs until their jobs started running; users seemed to be concerned very little about exhausting their supply of credits.

Because the bidding currency used had no external value, users had no incentive to preserve their account balances for any purpose beyond use internal to the cluster; because there was little need for currency within the cluster, users simply didn't pay much attention to their balances and bid whatever they felt like bidding. Using a currency *with* external value — for example, actual money (actual dollars, *etc.*) would likely produce very different results.

8.6 Traditional metrics don't capture user desires

Most important, however, is the ability of economic schedulers to capture users' desires for their own jobs more accurately and effectively than traditional schedulers. Existing schedulers are chosen to optimize a particular global system metric; they allow users to express their desires only in terms of derived metrics (priority, submission queue) or only via administrative override.

By allowing the user to express his or her desires for a job more directly — via a bid, which is measured against the user's level of funds remaining — economic schedulers give users a richer vocabulary in which they can express their requirements for a job. The notion of "funds" has real meaning to most users; they inherently understand the concept, and can balance their expenditures now against the need to keep some remaining for later.

By imparting direct information about user desires to the scheduler, economic systems allow that scheduler to make better decisions about which jobs to run immediately and which jobs will accept substantial delay; this results in all users being more satisfied with the level of service they receive from the system. By optimizing the system for what is, in the end, the sum total of user happiness with the system, an economic scheduler inherently works towards maximum user satisfaction.

9 Issues and Future Work

Considered now are some of the issues that arose in the design, implementation, and deployment of this system, and possible extensions of this work into future research.

9.1 Users understanding the system

The biggest challenge to the overall success of the system was in our ability to express to users the nature of an economic scheduler, so that they could make informed decisions as to what bids to assign to their jobs. While the fundamentals of the Vickrey scheduler are straightforward, the entire concept of an economic scheduler remained quite new to most users and thus took some adjustment. Observing users' interactions with the system, we noted that while the details of the Vickrey scheduler were actually quite easily understood, users remained unclear as to exactly what bids to assign to their jobs. After it became abundantly clear that users were essentially bidding in a random binary fashion — and, further, that exactly what users were bidding was relatively unimportant to the system — we added suggestions to our introduction to the system that users start with a bid of zero and increase it when necessary.

Nevertheless, we believe that with increased traffic to the system and thus serious use of the actual auction mechanism internal to the scheduler, substantial work — in user education and usability of the system — would need to be done before users would feel as comfortable using an economic scheduler as a traditional queuing system. We also consider the desire of users for *transparency* in the system: for users to feel confident in such a new scheduling model, it had to be immediately obvious exactly why the scheduler has made the decisions it did. Our user toolset had the ability to display all jobs in the system, but, because a Vickrey auction was used, only the bids of the user's own jobs could be displayed. This presents a significant challenge to transparency and to user acceptance of the system, as the other users' bids that are used to calculate the billing rate of a user's own jobs are not visible. We therefore believe that future work would need to be done into making the system more transparent if larger numbers of users and a greater workload were applied; as actual contention rose, users would become increasingly concerned that the system was equitable and predictable.

Along these lines, we propose also a simple system of *high-low* bids: because users tended to simply bid zero or "something" in the system as it exists today, providing an explicit one-bit mechanism to do exactly this might greatly ease users' approach into the system. Users already understand the concept of assigning priority to jobs, and providing a single switch to indicate "this job has high priority" (and thus assigning it a pre-set positive bid) would allow users to derive some of the benefit of an economic scheduler without trying to decide exactly what bids to assign to their jobs. As a user's experience with the system matured, they could, too, begin using more finely-grained bids to better adjust the response of the system to suit their needs.

9.2 Combination with Time-Shared System

We also consider the need for this space-shared, queuing model to coexist with a time-shared, immediate-execution model. While the queuing model provides a well-understood interface to the cluster, the existing GLUnix system uses a time-shared model that provides a much closer mapping to users when debugging programs or running relatively short-lived, interactive jobs. In our implementation, we simply created a space "outside the economy" for users to run these short-lived programs.

An economic model for time-shared systems has been developed and implemented in [22]. A natural extension, therefore, would be to integrate the work here with the economic model for time-shared systems, and allow the two to coexist and use the same underlying economic mechanisms. This integration is well beyond the scope of this paper; however, it seems apparent that integrating the two systems would give users the "best of both worlds" while maintaining the distinct advantages that an economic scheduling approach can have.

9.3 Production-system issues

A very large class of enhancements and improvements to our system would need to be made for use in a large-scale production environment, where reliability, security, and performance were critical. In general, very many parts of a batch system should be transactional: jobs should be executed once, even in the face of various system failures; failure of a job should be detected and that job's record "rolled back" for re-insertion into the queue (perhaps at the head), instead of the system simply proceeding onwards. A number of commercial systems have attacked very similar problems, and the solutions are relatively well-known [46].

The integration of economic systems with such a queuing system adds to the level of transactionality, fault-tolerance, and robustness that must be present in a production system, especially if currencies with external value (*e.g.*, actual dollars) are to be used. Such a system would require guarantees and transaction processing similar to that used by a real-world bank, and so similar systems would need to be added. In general, we believe that the implementation of an economic queue in a full-scale production environment, using actual money as the currency, would require a significant rearchitecture of the batch-queuing system to provide transactional guarantees to its end users on job runs, account billing, bidding, and so forth.

9.4 Intercluster issues

As we move towards the construction of a campus-wide Millennium intercluster, too, we consider the issues this might raise with our implementation of an economic scheduling system. An intercluster presents the interesting challenge of distributed administration of a cluster: each campus department will have its own small cluster of computers, administered locally, and yet the clusters will all need to interoperate. Because an economic scheduler treats the entire intercluster as one homogeneous unit, local administrators would have little influence over the jobs running on their own cluster.

We therefore consider possible future work into implementing an economy using different currencies for each administrative domain; by adjusting manually the exchange rate between a common central currency and a cluster's own local currency, administrators could easily influence the usage of their cluster by jobs drawn from a shared economy.

There are also certainly systems implementation issues that would need to be carefully considered in such an environment: because the entire cluster is so widely distributed, notions of fault-tolerance and distributed economic decision-making need to be addressed. Our current work is only suitable for situations in which all decisions are made centrally and then enforced; as an intercluster grows to include multiple smaller clusters across campus, this model has serious faults in fault-tolerance and scalability.

9.5 Technical issues

Finally, we consider the technical issues that arose during implementation of this system. The largest technical challenges to implementation came from the inability of the underlying runtime mechanism to handle the suspension and resumption of jobs well, and the inability of the runtime mechanism to checkpoint and migrate jobs whatsoever. These abilities are uncommon on clusters as of yet; however, their absence presents significant challenges to economic scheduling.

Without the ability to suspend and resume jobs, economic scheduling degenerates into a sort of FIFO queue as earlier jobs begin running and then cannot be pre-empted. While our system was able to integrate an inelegant form of suspension and resumption, it nevertheless caused problems with some users' jobs, and with benchmarks in particular. We suggest, therefore, that future economic scheduling may be dependent upon the ability of runtime mechanisms to cleanly handle suspension and resumption of jobs without disruption.

The further inability of the runtime mechanism to checkpoint and migrate jobs between nodes in the cluster caused further challenges to our implementation. Internal fragmentation of the available node space inevitably resulted, which caused a choice: either the economic model could be broken by allowing a lower-bidding job to run because a higher-bidding job's nodes were not all free, or both jobs could be run at once, degrading the delivered performance to each one. This is a fundamental result from the lack of process migration in this system and a significant challenge. Much existing work [47] has been done in this area, however, and so we merely note its importance to economic schedulers.

10 Bibliography

[1] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for Networks of Workstations: NOW. In *IEEE Micro*, February 1995.

[2] David Culler, Lok Tin Liu, Richard Martin, Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. In *IEEE Micro*, 1996.

[3] Myrinet Overview. http://www.myri.com/myrinet/overview/index.html.

[4] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, and Katherine Yelick. Cluster I/O With River: Making The Fast Case Common. *IOPADS '99*.

[5] A. Fox, S. Gribble, Y. Chawathe and E. A. Brewer. Cluster-Based Scalable Network Services. In *Proceedings of SOSP '97*, St. Malo, France, October 1997.

[6] Michael Stonebraker. The Case For Shared Nothing. In *Report on the International Workshop on High-Performance Transaction Systems*, Pacific Grove, California, September 1985.

[7] Andrew S. Tannenbaum and Albert S. Woodhull. Operating Systems: Design and Implementation.

[8] Immaneni Ashok and John Zahorjan. Scheduling a Mixed Interactive and Batch Workload on a Parallel, Shared-Memory Supercomputer. In *Proceedings of Supercomputing '92*, pages 616-625, November 1992.

[9] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First Syposium on Operating Systems Design and Implementation*, Usenix Association, November 1994.

[10] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Tech Report MIT/LCS/TM–528, Massachusetts Institute of Technology, 1995.

[11] Jeffrey S. Banks, John O. Ledyard, and David P. Porter. Allocating uncertain and unresponsive resources: In *RAND Journal of Economics*, Vol. 20, No. 1, Spring 1989.

[12] William J. Hausman and John L. Neufeld. Time-of-Day Pricing in the U.S. Electric Power Industry at the Turn of the Century. In *RAND Journal of Economics*, Vol. 15, No. 1, Spring 1984.

[13] Arye L. Hillman and John G. Riley. Politically Contestable Rents and Transfers. In *Economics and Politics*, Spring 1989, No. 1, pages 17–39.

[14] Ron Cocchi, Scott Shenker, Deboah Estrin, and Lixia Zhang. Pricing in Computer Networks: Motivation, Formulation, and Example. Tech report, University of Southern California, October 1992.

[15] Donald F. Ferguson, Chsitros Nikolaou, Jakka Sairamesh, and Yechiam Yemini. Economic Models for Allocating Resources in Computer Systems. In *Market-Based Control of Distributed Systems*, Ed. Scott Clearwater, World Scientific Press, 1996.

[16] William E. Walsh, Michael P. Wellman, Peter R. Wurman, and Jeffrey K. MacKie–Mason. Some Economics of Market-Based Distributed Scheduling. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, May 1998.

[17] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffery O. Kephart, and W. Scott Stornetta. Spawn: A Distributed Computational Economy. In *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, February 1992, p. 103–117.

[18] Michael Stonebraker, Robert Devine, Marcel Kornacker, Witold Litwin, Avi Pfeffer, Adam Sah, and Carl Staelin. An Economic Paradigm for Query Processing and Data Migration in Mariposa. Sequoia 2000 Technical Report 94/49, University of California, Berkeley, CA, April 1994.

[19] Ori Regev and Noam Nisan. The Popcorn Market — Online Markets for Computational Resources. *First International Conference On Information and Computation Economies*, Charleston, SC, 1998.

[20] Michael P. Wellman and Peter R. Wurman. Market-Aware Agents for a Multiagent World. *Robotics and Autonomous Systems* 24:115–125, 1998.

[21] Alexander Chislenko and Madan Ramakrishnan. Hyper-Economy: Combining Price and Utility Communication in Multi-Agent Systems. In *Proceedings of the 1998 IEEE ISIC/CIRA/ISAS Joint Conference*, Gaithersburg, MD, September 1998.

[22] Brent N. Chun and David E. Culler. REXEC: A Decentralized, Secure Remote Execution Environment for Clusters. In *4th Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*, Toulouse, France, January 2000.

[23] Brent N. Chun. Anemone: A Market-Based Cluster Batch Queue System. http://www.cs.berkeley.edu/~bnc/anemone/.

[24] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin H. Vahdat, and Thomas E. Anderson. GLUnix: A Global Layer UNIX for a Network of Workstations. In *Software — Practice and Experience*, April 1998.

[25] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan Mainwaring. Scheduling With Implicit Information in Distributed Systems. In *SIGMETRICS '98 Conference on the Measurement and Modeling of Computer Systems*.

[26] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. Parallel Computing on the Berkeley NOW. In *Proceedings of the 9th Joint Symposium on Parallel Processing*, Kobe, Japan, 1997.

[27] Steven S. Lumetta, Alan M. Mainwaring, and David E. Culler. Multi-Protocol Active Messages on a Cluster of SMPs. In *Proceedings of SC '97*, San Jose, California, November 1997.

[28] William Vickrey. Counterspeculation, Auctions, and Competitive Sealed Tenders. In *Journal of Finance*, Vol. 16, March 1961, p. 8–37.

[29] Paul Milgrom. Auctions and Bidding: A Primer. In *Journal of Economic Perspectives*, Vol. 3, Summer 1989, p. 3–22.

[30] V. V. Chari and Robert J. Weber. How the U.S. Treasury Should Auction Its Debt. Federal Reserve Bank of Minneapolis Quarterly Review, Fall 1992, p. 3–12.

[31] U.S. Department of the Treasury, U.S. Securities and Exchange Commission, and Board of Governors of the Federal Reserve System, *Joint Report on the Government Securities Market*. Washington, D.C.:Government Printing Office, 1992.

[32] Agorics, Inc. Government Securities — Auctioned Off. http://www.agorics.com/auctions/auction10.html.

[33] eBay. http://www.ebay.com/.

[34] Richard H. Thaler. The Winner's Curse: Paradoxes and Anomalies of Economic Life. New York: The Free Press, 1992.

[35] Generic NQS. http://www.gnqs.org/.

[36] GNU Queue. http://www.gnu.org/software/queue/queue.html.

[37] The Portable Batch System. http://www.openpbs.org/.

[38] International Business Machines Corporation. IBM RS/6000 hardware: SP Server. http://www.rs6000.ibm.com/hardware/largescale/SP/index.html.

[39] SGI, Inc. SGI Origin 2000 Datasheet. http://www.sgi.com/Products/PDF/2500.pdf.

[40] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. Beowulf: A Parallel Workstation for Scientific Computation. In *Proceedings, International Conference on Parallel Processing*, 1995.

[41] JavaSoft. http://www.javasoft.com/.

[42] Java Native Interface Specification. May 16, 1997. http://www.javasoft.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html.

[43] JDBC: A Java SQL API. January 10, 1998. http://www.javasoft.com/j2se/1.3/docs/guide/jdbc/spec/jdbc-spec.frame.html.

[44] Paul DuBois. MySQL. New Riders: December, 1999.

[45] Andrew Geweke. A System for Batch-Mode Economic Scheduling of a Cluster of Workstations. http://www.geweke.org/andrew/ucb/thesis.

[46] Philip A. Bernstein and Eric Newcomer. Principles of Transaction Processing (Morgan Kaufman Series in Data Management Systems). Morgan Kaufman Publishers: January, 1997.

[47] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. University of Wisconsin-Madison Computer Sciences Technical Report #1346, April 1997.