Market-based Cluster Resource Management

by

Brent Nee Chun

B.S. (University of Southern California) 1995
M.S. (University of California at Berkeley) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor David E. Culler, Chair
Professor Christos Papadimitriou
Professor Hal Varian

Fall 2001

The dissertation of Brent Nee Chun is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2001

Market-based Cluster Resource Management

Copyright Fall 2001
by
Brent Nee Chun

Abstract

Market-based Cluster Resource Management

by

Brent Nee Chun

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor David E. Culler, Chair

Resource management in high-performance, cluster computer systems is a challenging problem. Resources must be allocated amongst competing applications of varying levels of importance, and aggregate resource demand needs to be controlled to keep the system in a comfortable regime of operation. Effectively performing these tasks requires knowledge of user valuations of the resources being allocated and having a feedback signal that causes users to back off the system when it is overloaded. Unfortunately, current approaches to cluster resource management provide little, if any, means for users to express resource valuations and to influence their resource allocations. In addition, while feedback signals are provided, there are no associated incentives for users to pay attention to and respond to them. As a result, traditional systems are incapable of delivering the maximum possible value to users.

The thesis of this work is that by applying economic metaphors to cluster resource management, significantly more value can be delivered to users compared to traditional approaches. With market-based resource management, computational resources are allocated through markets where the cost of using a resource is directly related to supply and demand. Resource allocations are determined through use of economic mechanisms such as auctions where users place explicit valuations on the resources being contended for as expressed by mechanism-specific "bids". Explicit valuations provide systems with extra information which is used to optimize for user value as opposed to system-centric performance metrics, such as mean slowdown. High prices combined with charging creates feedback which causes users to back off the system when contention for resources is high. We hypothesize that ex-

plicit resource valuations and price feedback can lead to substantially higher value delivered to users.

To support our thesis, we demonstrate both feasibility through practical implementations of prototype systems and quantitative advantages through use of extensive simulations. For feasibility, we present a system architecture for market-based cluster resource management and describe two prototype implementations that realize this architecture, one for time-shared environments, the other for batch environments. Both were deployed on subsets of a 300+ CPU cluster of SMPs and subjected to use by a real user community. To quantify the benefits of market-based systems in a systematic fashion, we perform an extensive set of simulations of a 32-node cluster. We analyze performance sensitivity to distributions on resource valuations and distributions on workload parameters including job arrival processes, job resource requirements, workload burstiness, and utilization among others. Using these results, we pinpoint the types of workloads and the regimes of operation in which market-based systems have the most impact.

Results show that market-based systems deliver substantially higher value for both time-shared systems and batch systems compared to traditional approaches. For time-shared systems, we observe that explicit resource valuations are a huge benefit and see substantial improvements in value delivered (2x-5x for typical workloads) when resource valuations are known as opposed to when they are not known. On the other hand, we observe that price feedback is only moderately effective, yielding at most a 30% improvement in value delivered, since knowing valuations already provides systems with enough information to focus on high valued uses of resources during contention. For batch systems, we observe that knowing valuations leads to large performance improvements similar to the time-shared case when valuations are known. On the other hand, we observe that adequately parameterized supercomputing center policies with multiple FIFO queues with different priorities are almost as effective as a first price auction with time-varying bids for sequential jobs but substantially less effective for parallel workloads.

Professor David E. Culler
Dissertation Committee Chair

To my Mom

Contents

List of Figures vii

1 Introduction 1
- 1.1 Market-based Resource Management . . . 2
- 1.2 Contributions . . . 4
- 1.3 Organization . . . 6

2 Related Work 7
- 2.1 Traditional Cluster Resource Management Systems . . . 7
  - 2.1.1 GLUnix . . . 8
  - 2.1.2 SCore . . . 8
  - 2.1.3 MOSIX . . . 8
  - 2.1.4 Condor . . . 9
  - 2.1.5 NQE . . . 9
  - 2.1.6 PBS . . . 10
  - 2.1.7 LSF . . . 10
  - 2.1.8 IBM Load Leveler . . . 11
- 2.2 Supercomputing Centers . . . 11
  - 2.2.1 Pittsburg Supercomputing Center . . . 11
  - 2.2.2 Maui High Performance Computer Center . . . 12
  - 2.2.3 San Diego Supercomputing Center . . . 12
  - 2.2.4 National Center for Supercomputing Applications . . . 13
- 2.3 Resource-Aware Applications . . . 13
  - 2.3.1 Odyssey . . . 14
  - 2.3.2 Congestion Manager (CM) . . . 14
  - 2.3.3 ReMoS . . . 15
  - 2.3.4 Chang and Karamcheti's Work . . . 15
- 2.4 Multi-Agent Systems . . . 16
  - 2.4.1 WALRAS . . . 16
  - 2.4.2 Bidding Strategies in CDAs . . . 16
  - 2.4.3 Bidding Strategies in Fragmented Markets . . . 17
- 2.5 Pricing in Computer Networks . . . 17
  - 2.5.1 Cocchi et al.'s Internet Pricing Work . . . 18
  - 2.5.2 Edge Pricing . . . 18

2.5.3 Expected Capacity Allocation . . . 19
2.5.4 INDEX . . . 19
2.6 Parallel and Distributed Systems . . . 20
2.6.1 Miller and Drexler's Work . . . 20
2.6.2 Ferguson's Work . . . 21
2.6.3 Spawn . . . 22
2.6.4 Mariposa . . . 22
2.6.5 Stoica's Microeconomic Parallel Scheduler . . . 23
2.6.6 Sairamesh's Work . . . 24
2.6.7 MarketNet . . . 25
2.6.8 Popcorn . . . 25
2.6.9 Geweke's Economic Batch Queue . . . 25

3 System Architecture **27**
3.1 Three Fundamental Requirements . . . 27
3.1.1 Expressing Value . . . 28
3.1.2 Translating Value . . . 29
3.1.3 Enforcing Value . . . 30
3.2 Market Structure . . . 31
3.2.1 Goods . . . 31
3.2.2 Economic Mechanisms . . . 33
3.3 Architecture . . . 35
3.3.1 Resources . . . 36
3.3.2 Resource Managers . . . 37
3.3.3 Economic Allocators . . . 38
3.3.4 Access Modules . . . 39
3.3.5 End Users and Applications . . . 40
3.4 Implementation Issues . . . 41
3.4.1 Computational and Communication Complexity . . . 42
3.4.2 Scalability and Robustness . . . 42
3.4.3 Preemption . . . 43
3.4.4 Trust . . . 44

4 Prototype Implementations **46**
4.1 REXEC Remote Execution System . . . 46
4.1.1 Design Goals and Assumptions . . . 47
4.1.2 System Architecture . . . 50
4.1.3 Key Features . . . 50
4.1.4 REXEC Applications . . . 62
4.1.5 Related Work . . . 66
4.2 Time-shared Computational Economy . . . 67
4.2.1 End Users and Applications . . . 68
4.2.2 Access Module . . . 68
4.2.3 Economic Allocator . . . 69
4.2.4 Resource Managers . . . 72

4.2.5 Resources . . . 74
4.3 Anemone: a Market-based Batch Queue System . . . 75
4.3.1 End Users and Applications . . . 75
4.3.2 Access Module . . . 76
4.3.3 Economic Allocator . . . 78
4.3.4 Resource Managers . . . 82
4.3.5 Resources . . . 82
4.4 Experience . . . 83
4.4.1 Experimental Setup . . . 83
4.4.2 Observations . . . 84
4.4.3 Summary . . . 87

5 Stingray: a Market-based Simulator 88
5.1 System Model . . . 88
5.1.1 Module Interactions . . . 89
5.1.2 Examples . . . 91
5.2 Implementation . . . 91
5.2.1 Simulation Engine . . . 94
5.2.2 Modules . . . 94
5.2.3 Simulation Configuration Files . . . 100

6 Simulations 103
6.1 Evaluating Performance . . . 103
6.2 Time-shared Simulations . . . 105
6.2.1 Assessing the Benefits of Valuation Information . . . 107
6.2.2 Assessing the Benefit of Price Incentives . . . 116
6.2.3 Delayed, Imperfect Information . . . 124
6.2.4 Strategic Users . . . 128
6.3 Batch Simulations . . . 133
6.3.1 Assessing the Benefits of Valuation Information . . . 135
6.3.2 Evaluating Preemption . . . 146

7 Conclusions 158
7.1 Summary . . . 158
7.1.1 Implementations . . . 159
7.1.2 Performance . . . 160
7.2 Future Directions . . . 165

Bibliography 169

List of Figures

3.1 End-to-end System Architecture. 36

4.1 The REXEC Remote Execution Environment. 51
4.2 Internal Thread Structure and Data Flows for `rexec` and `rexecd`. 54
4.3 Measured Execution Time to Run a Null Parallel Program with REXEC as a Function of Number of Nodes. 64
4.4 Time-shared Computational Economy. 70
4.5 Measured CPU Allocation and Charging on a Single Node. 71
4.6 Stride Scheduling Algorithm on SMPs. 73
4.7 Anemone Batch File. 76
4.8 The Anemone Market-based Batch Queue System. 79
4.9 State Diagram for Growing and Shrinking the Batch Pool Based on Batch Supply and Demand, `BatchSup` and `BatchDem`. 81
4.10 Valuation Distributions for the Priced Cluster. 85
4.11 CPU Load Distribution for Priced and Unpriced Clusters. 86

5.1 Stingray System Model. 89
5.2 Stingray Module Configuration for the Time-shared Computational Economy. 92
5.3 Stingray Module Configuration for the Anemone Market-based Batch Queue System. 93
5.4 Event-driven Simulation in Stingray. 95
5.5 Sample Interconnection of Stingray Simulation Modules and Bulletin Boards. 101
5.6 Snippet from Example Stingray Configuration File. 102

6.1 Piecewise-linear Valuation. 104
6.2 System Configuration for Time-shared Simulations. 106
6.3 Time-shared Performance Sensitivity to Variation in Resource Valuations. . 108
6.4 Time-shared Performance Sensitivity to Variation in Resource Valuations (Normal and Exponential Length Jobs). 110
6.5 Time-shared Performance Sensitivity to User Tolerance to Delay. 111
6.6 Time-shared Performance Sensitivity to Burstiness in the Workload. 113
6.7 Time-shared Performance Sensitivity to Utilization. 114
6.8 Benefit of Knowing Resource Valuations in Time-shared Environments. . . 115
6.9 Time-shared Performance Sensitivity to Maximum Number of Retries. . . . 117

6.10 Time-shared Performance Sensitivity to Retry Rate Threshold. 119
6.11 Time-shared Performance Sensitivity to Retry Times. 120
6.12 Time-shared Performance Sensitivity to Utilization. 121
6.13 Time-shared Benefit of Price Feedback. 123
6.14 Time-shared Performance Sensitivity to Periodic Update Time. 125
6.15 Comparing State Update Algorithms. 127
6.16 Quantifying the Impact of Strategic Users 132
6.17 System Configuration for Batch Simulations. 134
6.18 Batch Performance Sensitivity to Variation in Resource Valuations. 136
6.19 Batch Performance Sensitivity to Variation in Resource Valuations (Normal and Exponential CPU Length Jobs). 138
6.20 Batch Performance Sensitivity to User Tolerance to Delay. 139
6.21 Batch Performance Sensitivity to Burstiness in the Workload. 141
6.22 Batch Performance Sensitivity to Utilization. 142
6.23 Batch Performance Sensitivity to Parallelism in the Workload. 144
6.24 Batch Performance Sensitivity to Parallel Degree Distribution. 145
6.25 Benefit of Knowing Resource Valuations. 147
6.26 Benefit of Fine-grain, Time-varying Resource Valuations. 148
6.27 Batch Sensitivity to User Tolerance to Delay with Preemption. 150
6.28 Batch Performance Sensitivity to Mix of Low/High Value Jobs with Preemption. 152
6.29 Batch Performance Sensitivity to Job Lengths with Preemption. 154
6.30 Batch Performance Sensitivity to Parallelism in the Workload with Preemption. 155
6.31 Benefit of Preemption in Batch Environments. 156

Acknowledgements

I would like to thank my advisor, Professor David Culler, for helping me become a better researcher and for being patient with me, despite my numerous side projects over the years. There are very few people whom I respect and admire as much as David. I am thankful of having had the privilege of working with him during my time here at Berkeley.

I would like to thank my fellow officemates, both past and present, in 467 and 533 Soda Hall. In alphabetical order, they are Cedric Krumbein, Lok Liu, Rich Martin, Matt Welsh, Frederick Wong, and Chad Yoshikawa. I consider myself fortunate to have been surrounded by such a talented and interesting group of individuals. Thanks again for all the great times.

I would like to thank the members of the Berkeley NOW and Millennium Projects. Besides the 467/533 crowd, I would also like to thank Eric Anderson, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Steve Lumetta, Alan Mainwaring, Steve Rodrigues, and Alec Woo from NOW and Phil Buonadonna and Jason Hill from Millennium. Alan, in particular, was especially influential in my development as a systems researcher.

Many thanks to Eric Fraser, Matt Massie, and Albert Goto for providing and maintaining a stable computing environment in which research could be done. It is not easy managing large-scale distributed computing systems like NOW and Millennium, especially when much of the software is experimental. Nevertheless, they have done an admirable job. Thanks again.

Professors Jim Demmel, Christos Papadimitriou, and Hal Varian all served as members on either my qualifying exam or dissertation committees. I would like to thank them for their early feedback on my ideas on market-based resource management and for taking large chunks out of their incredibly busy schedules in order to talk to me.

Ngeci Bowman, Charis Kaskiris, Jason Lee, and Tao Ye all helped to make life as a graduate student more enjoyable through various "extracurricular" activities. Tao (aka the stomper), in particular, has been a great friend through of all this, even if she does wear 20 lb shoes. Atlantis 2000, the numerous XW parties, and the full moon party on Koh Phangan were all immense amounts of fun.

Navigating the complex administrative maze at Berkeley would have been significantly more painful had it not been for Kathryn Crabtree, Peggy Lau, Bob Miller, and Terry Lessard-Smith. Over the years, they've done a terrific job of shielding computer science graduate students from the UC bureaucracy. Because of this, I could probably count

the number of times I've been to Sproul on two hands.

Of course, none of this would have been possible without the love and support of my Mom. As far back as I can remember, she has gone out of her way to make things easier for me in all aspects of life. Graduate school here at Berkeley was no exception. She has made huge sacrifices in order to help me achieve my goals and I cannot thank her enough for that.

Finally, last but certainly not least, I would like to thank my girlfriend Sandy for her love, patience, and support during my final stretch of graduate school. Through often stressful times, she has always been understanding and supportive of everything I do, even if it did mean spending less time together. She is truly an exceptional person and I consider myself extremely lucky to have met someone like her.

Chapter 1

Introduction

Resource management in high-performance, cluster computer systems is a challenging problem. Resources must be allocated amongst competing applications of varying levels of importance, and aggregate resource demand needs to be controlled to keep the system in a comfortable regime of operation. Effectively performing these tasks requires knowledge of user valuations of the resources being allocated and having a feedback signal that causes users to back off the system when it is overloaded. Unfortunately, current approaches [7, 21, 28, 37, 46, 48, 68, 71, 103] to cluster resource management provide little, if any, means for users to express resource valuations and to influence their resource allocations. In addition, while feedback signals are provided, there are no associated incentives for users to pay attention to and respond to them. As a result, traditional systems are incapable of delivering the maximum possible value to users.

The thesis of this work is that by applying economic metaphors to cluster resource management, significantly more value can be delivered to users compared with traditional approaches. With market-based resource management, resource allocation is based on explicit user valuations of resources as expressed by a willingness to pay (e.g., a bid). During contention, market-based systems use user resource valuations to distinguish between low and high valued use of resources and allocate resources accordingly. Resources are priced based on supply and demand so the cost of using a resource is directly related to the degree of contention. High prices combined with charging creates price incentives which encourage users to back off from the system when contention for resources is high. We hypothesize that explicit resource valuations combined with price incentives can lead to substantially higher value for users.

In this chapter, we provide a high-level overview of this thesis. In the following section, we summarize the key components of our study of market-based cluster resource management. Next, we present our primary research contributions. Finally, in the last section, we describe the organization of the rest of this dissertation.

1.1 Market-based Resource Management

With the maturing of clusters from low-cost supercomputers to general-purpose computer systems [3, 5, 13, 15, 22, 55, 76, 92], resource sharing has become the rule rather than the exception in all but very specialized cases (e.g., Google [39], Inktomi [49]). Resource management systems must grapple with how to allocate resources to competing applications of varying levels of importance and how to control aggregate demand to keep the system in a comfortable regime of operation. Traditional systems, unfortunately, do not take the importance of applications into account when allocating resources nor do they provide feedback signals which encourage users to back off the system when resources are overcommitted. Consequently, when contention occurs, traditional systems are unable to focus resources on the applications that are delivering the most value and thus important applications receive less than they should, unimportant applications receive more than they should, and suboptimal value is delivered to users.

To address the suboptimal value delivered to users with traditional approaches, we propose systems based on market-based resource management. With market-based resource management, resource allocation is based on economic mechanisms (e.g., auctions) that optimize for user value as opposed to scheduling policies that optimize for system-centric performance metrics. Resource allocation is driven by explicit user valuations of resources, expressed as mechanism-specific "bids". Larger bids lead to larger allocations which result in more timely execution and consequently more value delivered since large bids correspond to high value and valuations decay with execution time. Resources are priced based on supply and demand so the cost of using a resource is directly related to the degree of contention. High prices combined with charging creates price incentives which encourage users to back off from the system when contention for resources is high and thus helps avoid the "tragedy of the commons".

The idea of applying economics to resource management in distributed systems is not new. Previous research efforts [30–32, 36, 56–59, 75, 78, 82, 83, 94, 99] as early as two

decades ago have already made some headway into understanding the potential benefits of market-based systems. Unfortunately, most of this work has focused on markets more as a system organizational principle to manage complexity, rather than a means to improve value delivered to users. To more fully understand the capabilities of market-based systems for delivering increased user value, two challenges remain. First, practical implementation experience is needed to flesh out the issues and trade-offs in building market-based systems at a level that does not arise in the high-level designs to date. Second, quantitative evidence is needed to determine the types of workloads and the regimes of operation where market-based systems have the largest impact.

To address the first challenge, Part 1 of this thesis presents a system architecture for market-based resource management and describes two concrete implementations of it. Both systems were deployed on subsets of the UC Berkeley Millennium cluster, a 300+ CPU cluster of symmetric multiprocessors (SMPs), and subjected to use by real users. The first system provides market-based CPU allocation for time-shared environments. Resource allocation is based on competing, user-assigned rates which specify the amount the user is willing to pay per node CPU minute. The second system provides market-based CPU allocation for batch environments. There, resource allocation is done using a Vickrey auction which orders jobs in the queue by the amount they are willing to pay per node CPU hour. Through a detailed discussion of the design and implementation of these prototypes, we highlight a number of systems issues and trade-offs that arise when building practical implementations of market-based systems.

To address the second challenge, Part 2 of this thesis describes a market-based simulator and presents results from an extensive set of simulations. We compare the performance of market-based systems with traditional systems in terms of total value delivered to users. We analyze performance sensitivity to various distributions on resource valuations and distributions on workload parameters including job arrival processes, job resource requirements, workload burstiness, and utilization among others. Using these results, we test the thesis that market-based systems can deliver significantly more value to users. We quantify the value of being able to know resource valuations when allocating resources and show that the value of valuations increases as contention for resources increases. Additionally, we quantify the value of having price feedback and show that feedback results in higher value delivered when contention is high.

1.2 Contributions

To support the thesis that market-based systems can deliver significantly more value to users compared to traditional approaches, we have made several research contributions:

1. We articulate the three fundamental requirements that market-based systems must meet in order to deliver more value to users. These requirements all revolve around how to obtain and make use of extra information, specifically explicit user valuations of computational resources. To deliver more value, market-based systems must provide (i) means for users to express value, (ii) policies to translate value into resource allocations, and (iii) mechanisms to enforce differential resource allocations. Previous work in economics provides significant leverage in addressing the first two requirements while the rich quality-of-service and scheduling literature helps address the third.

2. We present a system architecture for market-based cluster resource management. The architecture decomposes market-based systems into five distinct layers which map well onto the structure of modern distributed systems. It can be used to describe a wide variety of market-based systems and thus provides a vocabulary which allows us to compare and contrast different market-based designs. Specifically, when using this architecture, we bring to the forefront the specific design decisions made to address the three fundamental requirements of market-based systems. Consequently, we are able to abstract away extraneous implementation details and focus in on how the systems are fundamentally trying to deliver more value.

3. We describe the implementation of two prototype systems that realize the system architecture. To achieve the ultimate goal of delivering more value to users, the three requirements of market-based systems must be met in the context of a real system. Real distributed systems impose additional requirements including low computational and networking overhead, scalability and robustness, trust and security, and so forth. Practical implementations of market-based resource management must take these factors into account in their design and implementations in order to avoid negating the benefits provided by the markets they create. Towards this end, we describe two prototype implementations that have been deployed on a large cluster of SMPs and have

been subjected to use by real users.

4. We present a model for user valuations of computational resources and a way to evaluate market-based systems against traditional systems in a way that is meaningful to users. We make a case for user-oriented performance metrics and present aggregate utility as one such metric. In our work, user valuations are modeled as piecewise-linear functions, where magnitude reflects job importance and rate of decay reflects user sensitivity to delay. Such valuations are typical of workloads in computational science and engineering, where jobs have varying levels of importance and the key thing users care about is execution time. Summing over all valuations for all jobs in the workload, we can then compute aggregate utility, a concrete metric which quantifies overall value delivered to users.

5. We present simulation results which quantify how much more value market-based resource management delivers to users compared with traditional resource management systems. For time-shared systems, we show that having valuation information leads to substantial improvements (2x-5x for typical workloads) in total value delivered to users, while price feedback is only moderately effective, yielding at most a 30% improvement. Benefits are shown to be most pronounced when workloads are bursty and there is a wide range of valuations on the underlying computational resources. Substantial benefits are also observed with less demanding workloads, specifically when users are fairly tolerant to delay and also at low levels of utilization with even a moderately bursty workload.

   For batch systems, we observe that having valuation information leads to large improvements in value delivered that are comparable to the time-shared case. Again, benefits were observed to be most significant when workloads are bursty, which in batch systems is the usual mode of operation. Substantial benefits are also observed with less demanding workloads, again when users are fairly tolerant to delay and also at low levels of utilization. Simulations also show substantial gains in value delivered (up to 14x) for highly parallel workloads. Finally, we also observed that adequately parameterized supercomputing center policies with multiple FIFO queues with different priorities are almost as effective as a first price auction with time-varying bids for sequential jobs but substantially less effective for parallel workloads.

1.3 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we discuss related work from a wide variety of areas related to both traditional and market-based resource management. In Chapter 3, we set the stage for the rest of the thesis by discussing the three fundamental requirements that market-based systems must meet in order to deliver more value to users. From there, we examine both economic and systems perspectives on satisfying these requirements and propose a system architecture for building market-based systems. In Chapter 4, we show that this architecture is both natural and feasible to realize by describing two prototypes systems. Both systems provide markets for CPU time, the first for time-shared environments, the second for batch environments. The design of these systems highlights a number of systems issues that arise when building practical implementations of market-based systems. In Chapter 5, we describe the simulator we used to simulate the behavior of market-based systems, including the two prototypes. Given this simulator, in Chapter 6, we present simulation results which quantify the benefits of market-based resource management compared with traditional resource management using a wide variety of workloads. Using these results, we pinpoint the types of workloads and the regimes of operation in which market-based systems have the most impact. Finally, in Chapter 7, we present our conclusions and discuss directions for future work.

Chapter 2

Related Work

In this chapter, we discuss a wide range of research related to market-based cluster resource management. We examine relevant work from a variety of different areas including parallel and distributed systems, computer networks, and artificial intelligence. In the following sections, we discuss previous work in traditional cluster resource management systems, resource management in supercomputing centers, resource-aware applications, multi-agent systems, pricing in computer networks, and market-based resource management for parallel and distributed systems.

2.1 Traditional Cluster Resource Management Systems

Traditional cluster resource management systems optimize for global, system-centric performance metrics. Compared to market-based systems, they provide limited, if any, means for expressing resource valuations and do not provide effective feedback signals. Instead, traditional systems make implicit assumptions on user resource valuations and encode these assumptions in policies which optimize for system-centric performance metrics such as mean response time. Feedback signals exist (e.g., CPU load on each node) but there are no incentives for users to observe and respond to them. The end result is that lack of explicit resource valuations leads to suboptimal resource management when optimizing for user value while lack of an effective feedback signal leads to overcomittment of resources during periods of high demand.

2.1.1 GLUnix

GLUnix is a user-level remote execution system for running parallel and sequential jobs on a cluster of workstations [37]. It supports time-shared interactive use, time-shared batch use, and dedicated access. For time-shared use, resource management is implemented using a centralized master process that uses a simple load balancing policy based on periodic samples of node CPU loads. For dedicated access, resource management is implemented using a reservation system that allows users to sign up for dedicated slots of time on sets of nodes. In both cases, resource usage is unrestricted. Users are free to submit unbounded amounts of work without consequence, even when the system is heavily loaded. In addition, users also have no way to express resource valuations other than by choosing dedicated access over time-shared access.

2.1.2 SCore

SCore is user-level, remote execution system designed specifically for running high performance parallel programs on a cluster of PCs [46]. It supports two modes of usage: time-shared, gang-scheduled interactive use and dedicated access. Resources are space partitioned based on type of usage supported. Resource management is implemented using queueing and admission control policies. For time-shared use, admission control policies are used to bound the degree of sharing on time-shared resources while in batch use, they are used to maintain the service contract of dedicated access. For time-shared access, queueing is implemented using multiple queues which have different scheduling priorities. Once admitted, time-shared jobs are routed to nodes using a simple load balancing policy based on CPU load. Like GLUnix, resource usage in SCore is also unrestricted. The key difference with SCore is its use of admission control and multiple queues for controlling time-shared use, the latter of which provides a rudimentary form of expressing resource valuations.

2.1.3 MOSIX

MOSIX is a load balancing system for clusters that uses preemption and process migration to balance load evenly across cluster nodes [7]. Resource management in MOSIX is implemented by nodes periodically exchanging state information with random subsets of nodes in the cluster and applying CPU load and memory thresholds to decide whether to

migrate jobs to less loaded nodes. By actively preempting and migrating processes, MOSIX mitigates the effects of load imbalances caused by bad placement decisions and unknown job completion times. Like GLUnix and SCore, resource management in MOSIX is based on the tacit assumption that balancing CPU load across all nodes results in allocations which leave users better off. Resource usage is also unconstrained and users have no way to express resource valuations.

2.1.4 Condor

Condor is a distributed batch queue system for running compute-intensive applications on loosely coupled pools of computers [28]. It uses centralized processes called Central Managers (CM) for sharing within pools and proxies called gateways (GW) to link pools together for sharing across pools. Resource management is implemented by the CMs which match job requests to resources using node-specific resource management policies and user resource usage history. Using node-specific policies gives users control over how their machines will be shared in a manner not unlike that of peer-to-peer systems such Napster and Gnutella. Using user resource usage history provides some degree of fairness. Exchange of resources within and across pools is assumed to be altruistic in nature. There are no incentives for users to share their machines nor is there any way for users to express resource valuations. Because of this, Condor is useful mainly as a system for exploiting machine idle time in environments where users are all cooperative and socially responsible.

2.1.5 NQE

The Network Queueing Environment (NQE) is a batch queue system for clusters of heterogeneous machines [21]. Resource management is implemented using three mechanisms. First, centralized load balancing routes incoming jobs to the "best" nodes. Second, node admission control policies control the degree of sharing. Third, node scheduling policies specify how to schedule jobs from multiple priority queues for execution. Node schedulers always select the job with the the highest intraqueue priority in the highest interpriority queue that is non-empty for execution. They schedule such jobs for execution based on admission control policies that specify a particular degree of sharing. Once running, jobs are assigned nice values which influence their resource allocations. In NQE, interqueue priorities, intraqueue priorities, and nice values are all specifiable by the user. This combination of priorities thus provides users with a means, albeit somewhat complicated, to

express resource valuations in terms of system-specific mechanisms. However, this scenario is clearly problematic when users are competing for shared resources since in those cases, there are no incentives to prevent users from always specifying the highest priority for all their jobs.

2.1.6 PBS

The Portable Batch System (PBS) is a widely used batch queue system for running parallel and sequential jobs on clusters [68]. Resource management is implemented using a centralized scheduler. The scheduler schedules jobs from a collection of batch queues using a site-specific scheduling policy. Scheduling policies are implemented by system administrators who use configuration files and a programmatic interface to PBS to implement policies which they feel are most appropriate for their cluster installations. Typical policies use simple rules based on CPU requirements, memory requirements, queueing time, user resource usage history, and other parameters to compute queue priorities. In most PBS installations, policies also allow for some rudimentary way for users to express resource valuations, usually by providing the choice of multiple queues with different priorities. However, as with NQE, there are usually no incentives to prevent users from always selecting the highest priority. Given the extensibility of PBS, though, it seems natural that PBS could be extended to implement such functionality. Geweke's work [36] is an example of such a system and is discussed in Section 2.6.

2.1.7 LSF

Load Sharing Facility (LSF) is a commercial product for time-shared and batch execution of parallel and sequential jobs on a cluster of heterogenous computers [71]. Time-shared resource management is implemented using centralized load balancing combined with per-host scheduling policies and resource limits. Per-host scheduling policies are used to implement suspension policies based on load thresholds which, when combined with progress migration, can be used to implement MOSIX style load balancing. Resource limits are used for admission control to avoid overloading the system and to support resource reservations. Batch resource management is implemented using the same mechanisms as the time-shared case along with configurable scheduling policies on multiple priority queues. Scheduling policies are based on the same inter and intra queue selection policy as NQE with different policies implementing different schemes to decide how to compute job priorities. Like most

traditional systems, LSF provides no way for users to express resource valuations and does not implement a useful feedback signal.

2.1.8 IBM Load Leveler

IBM Load Leveler is a popular batch queue system that is used on the IBM SP2, a parallel machine that is essentially just a cluster of RS6000 nodes [48]. Resource management is implemented using centralized load balancing based on configurable goodness metrics and a configurable scheduling policy over multiple priority queues. Like NQE, scheduling policies implement admission control to control the degree of sharing and schedule jobs based on interqueue and intraqueue priority in that order. Priorities are computed based on administrator-defined policies which depend on the usual parameters: CPU resource requirements, queueing time, and so forth. Configurable goodness metrics and configurable scheduling policies are provided to give system administrators the ability to implement policies which they feel are most appropriate for their systems. Like most of the systems in this section, Load Leveler also suffers from the same problems associated with the inability to express resource valuations and the lack of an effective feedback signal.

2.2 Supercomputing Centers

Supercomputing centers use conventional batch queue systems augmented with charging for resource management. They provide rudimentary means for expressing resource valuations by allowing users to choose amongst multiple, differentially-priced queues with different scheduling priorities when submitting their jobs. Prices combined with charging creates a meaningful feedback signal that influences user behavior since users have finite funds for purchasing resources. Because prices are usually fixed, though, price feedback is only effective as a rate-limiter over long intervals of time. However, for batch usage, this may be sufficient since batch systems implement their own admission control anyway and usually provide multiple queues with different scheduling priorities.

2.2.1 Pittsburg Supercomputing Center

The Pittsburgh Supercomputing Center (PSC) uses traditional batch queue systems (NQE) along with charging policies for resource management on their high performance

computing systems. Charging is based on resource usage which is measured in machine-specific units of resource consumption called service units (SUs). Machines have static policies which are used to compute SU usage and to debit user accounts. On the Cray J90, for example, SUs are computed using the following equation:

$$SU = \frac{3}{4}CPU + \frac{1}{32}MemInt + \frac{1}{100}Con$$

where CPU is CPU time used, $MemInt$ is a memory integral is megaword hours, and Con is the connection time to the machine. For the most part, instantaneous demand is not a factor in PSC's charging policies. The only place it comes into play is with multiprogrammed use, for which PSC provides discounts based on the number of sharers. Since PSC schedules jobs in FIFO order, lack of responsive pricing can be problematic since there are no incentives to prevent low valued jobs from being added to an already long queue, thereby blocking future high valued jobs. Furthermore, despite the existence of a charging system, users still have no way to express resource valuations to the system.

2.2.2 Maui High Performance Computer Center

The Maui High Performance Computer Center (MHPCC) also relies on a batch queue system with charging for resource management on its 602-node IBM SP2. Resources are spaced partitioned into two pools: a time-shared and a dedicated batch pool. Jobs are scheduled onto these pools using IBM Load Leveler in conjunction with the Maui Scheduler which performs backfilling. Like PSC, charging is based on application resource usage measured in machine-specific service units. At MHPCC, a service unit is essentially equivalent to a node CPU hour and is computed independent of the state of the system (e.g., current demand). Because charging is independent of demand, price feedback is ineffective. In an attempt to address this, rate limiting policies are implemented which constrain the number of simultaneously running jobs each user can have. Like PSC, MHPCC also has an underlying charging system but users have no way to use this system to express resource valuations.

2.2.3 San Diego Supercomputing Center

The San Diego Supercomputing Center (SDSC) implements charging policies on its high performance machines which include a 256-node T3E and a 1152-processor IBM SP2. Charging is based on application resource usage measured in machine-specific units

of resource consumption called service units (SUs). Charging policies are static and do not depend on the state of the system. Unlike PSC and MHPCC, users are provided with rudimentary means for expressing resource valuations. Valuations are expressed when submitting jobs through the selection of one of multiple priority queues, each of which has a different scheduling priority (0.5, 1.0, 1.8 on the T3E) and incurs charges proportional to the increase in priority relative to a base priority. Using these queues, users map large variations in resource valuations down to a fixed number of small choices. Valuations are thus quantized. How effective this quantization is depends on the interaction between queue pricing/priorities and actual distributions on resource valuations from the users. Regardless of the details of this interaction, though, the ability to obtain a higher scheduling priority by paying more is still an improvement over other systems which only offer a single quality of service or provide unconstrained access to multiple qualities of service.

2.2.4 National Center for Supercomputing Applications

The National Center for Supercomputing Applications (NCSA) uses a batch queue system and charging policies for resource management on its Origin 2000 parallel computer. Charging is based on resource usage measured in machine-specific service units and is based on a static charging policy. Like SDSC, NCSA also offers users the choice of multiple priority queues with different scheduling priorities (1.0, 1.25, 2.0) and charging rates associated with them. In addition, charging also takes into account heterogeneity in CPU speeds which, on their Origin 2000 system, concerns whether nodes have 195 MHz or 250 Mhz MIPS R10000 processors. To account for CPU speed differences, scaling factors are applied to SU usage: 1 for the base 195 MHz R10000's and 1.282 for 250 MHz R10000's. SU usage is computed by first measuring CPU hours, applying the queue priority factor, and finally by scaling it by the CPU speed factor. Using this pricing scheme, NCSA is effectively implementing differential pricing based on quality of service and is providing users with a simple way to express resource valuations through the selection of queue priorities and CPU speeds.

2.3 Resource-Aware Applications

Resource-aware applications make runtime trade-offs between quality and performance by altering their behavior based on feedback information on resource availability. Such adaptability is important in order to make application performance robust against

fluctuations in resource availability in systems with scarce, performance-critical resources. Previous work in this area has mainly focused on applying adaptability to systems where competing applications are all owned by the same user or all users are assumed to be cooperative. Extending this work to support systems with competitive users requires additional incentives beyond just the local desire to sustain a desired level of performance.

Market-based systems provide the necessary price incentives which make adaptivity a viable strategy in systems shared by multiple users. Without price incentives, adaptivity is not a stable strategy. A resource-aware application, knowing that all other applications will reduce their individual resource demands during periods of contention, has no incentive to reduce its individual resource demands. Rather, it would prefer to just keep its demand constant and wait until all other applications adjust until aggregate demand falls to a point where the system is back in a comfortable regime of operation. With appropriate price incentives, applications are rewarded for adapting to contention and thus adaptivity becomes a viable strategy.

2.3.1 Odyssey

The Odyssey system provides an infrastructure for constructing data-centric mobile applications that can quickly adapt along application-specific axes of fidelity in response to changes in resource availability [64]. In Odyssey, each application dynamically expresses its resource expectations on data types that specify windows of tolerance on performance-critical resources. The system then monitors resource availability and makes upcalls to the application when windows of tolerance are violated. In response to an upcall, applications make data type-specific changes in fidelity in order to maintain a given level of performance (e.g., lowering the sampling rate for audio in a networked audio application in response to a significant decrease in available network bandwidth).

2.3.2 Congestion Manager (CM)

The congestion manager (CM) provides integrated network flow management and an API for constructing network-aware applications [2]. In CM, applications can request callbacks that inform applications whenever transitions in network bandwidth levels occur. More specifically, applications can be informed when a certain network bandwidth becomes available or, if applications are already sending at some rate, when a certain network bandwidth is no longer sustainable. Explicit queries on network resource availability

can also be made. Using these queries, applications that want to be proactive about their adaptability can query the system for estimates on current bandwidth and round-trip times to any destination address. Using callbacks and explicit queries, applications can then make application-specific changes to provide good performance in response to variations in network resource availability.

2.3.3 ReMoS

The ReMoS system provides a query-based API for adaptivity based on network resource availability [25, 44]. In ReMoS, resource-aware applications must be proactive and make explicit queries in order to observe changes in network state and respond to them. To provide applications with this state information, two types of queries are supported: *flow queries* which provide bandwidth and latency information between source-destination pairs and *topology queries* which provide information on the logical topology of the network. Query data on network performance metrics is provided over application-specific timescales. They include both averages and the variance, the latter of which can be useful in some cases. Implementation is based on a client library that implements the API and network state information obtained through router queries using SNMP.

2.3.4 Chang and Karamcheti's Work

Chang and Karamcheti have implemented a system that partially automates the construction of adaptation code to make it easier for programmers to write adaptive applications [12]. In this system, adaptation is implemented through specification of all possible application behaviors given different levels of resource availability. (No explicit callback handlers or queries are written by the programmer.) A virtual execution environment is then used to emulate various resource configurations and determine application sensitivity to resource allocations according to programmer-specified quality of service metrics. The results from this sensitivity analysis is stored in a performance database. The database, in conjunction with runtime monitoring of user-specified QoS metrics, is then used to automatically transition applications between their different adaptation behaviors.

2.4 Multi-Agent Systems

Multi-agent systems (MAS) is a branch of distributed AI that studies systems where multiple, self-interested agents [1] interact and compete for scarce, shared resources while pursing selfish goals. Work in this area mainly concerns the design and analysis of mechanisms for mediating agent interactions along with agent strategies for maximizing some desired metric given such mechanisms. This work is relevant for market-based cluster resource management in systems where software agents are used to represent the interests of users (buyers), sellers of cluster resources, or both.

2.4.1 WALRAS

WALRUS is a system for computing resource allocations based on general equilibrium theory for distributed planning systems [98]. Systems are specified in terms of goods, agents, bidding strategies, and market configurations. Resource allocations are computed using an iterative process by holding n independent auctions, one for each good, until an equilibrium is reached. In each auction i, consumers and producers submit demand curves and supply curves for good i under the assumption that prices for all other goods are fixed. Auctions are repeated and agents adjust their demand and supply curves until all agents have no new bids to submit and prices for all the auctions are stable. Using WALRUS, the behavior of a wide variety of market configurations can be examined. However, direct applicability of WALRUS to cluster resource management remains questionable though for several reasons including unknown quality of the equilibrium solution and the time to convergence, its assumptions of gross substitutability and infinitely divisible goods, computational and communication complexity of an implementation, and its focus on static resource allocation problems which do not involve time.

2.4.2 Bidding Strategies in CDAs

Dave Cliff's work examines agent bidding strategies in a continuous double auction (CDA) [18], the market mechanism used in global financial markets such as the NASDAQ. He investigates the question of what is the minimal amount of intelligence that software agents need to trade so that an equilibrium price can be established. He presents zero-

[1] By agent, we mean an entity (e.g., a process, a software module, etc.) that acts according to its own knowledge and interest.

intelligence plus (ZIP) agents, simple stochastic agents that alter their bids over time using a simple form of machine learning. Bids are based on agent limit prices along with target profit margins which are computed based on the history of the market. In one example, profit margins are adapted using the Widrow-Hoff delta-rule from backpropagation in neural networks. Using this small amount of intelligence (i.e., no-loss making trades plus the delta rule), ZIP agents are shown to achieve rapid convergence to equilibrium prices in a CDA for a single good, even with only a small number of agents. However, as Cliff points out, these results may not hold for more complex markets. Furthermore, it is unclear how sensitive these results are to the particular parameterizations used (e.g., the learning rates, etc.) and agent populations where some fraction of the agents do not employ these strategies.

2.4.3 Bidding Strategies in Fragmented Markets

In [72, 73], Priest investigates agent bidding strategies in a fragmented market comprised of multiple auctions offering the same goods. He assumes English auctions with discriminatory pricing where bids cannot be retracted. He presents two algorithms for buyers interested in m goods, the first for auctions which clear simultaneously, the second for auctions have different clearing times. Both algorithms use m outstanding bids and periodic checks on current winning bids to determine when an agent has been outbid and new bids need to be made. Both algorithms compute new bids to minimize the incremental cost of obtaining the additional goods for auctions where the agent was outbid. In the first case, cost is based on entirely on current bids while in the second, cost also takes into account the expected utility of remaining in an auction which is necessary to avoid always moving to the newest auctions and potentially never terminating. Through simulation, Priest's algorithms demonstrate higher allocative efficiency when compared to the simple strategy of buyers bidding their reservation price in m random auctions. For market-based resource management, such algorithms could prove useful in computer systems with competitive sellers in a fragmented marketplace (e.g., Grid computing systems).

2.5 Pricing in Computer Networks

Pricing in computer networks addresses many of the same issues as our work. Again, systems are designed with price incentives to encourage socially efficient use of differential service mechanisms. Key differences with network pricing include distributed

ownership and management of resources and a large, existing systems infrastructure (i.e., the Internet) that demands incremental deployment. Research in network pricing can be divided into essentially two categories. The first category is economic-oriented work and focuses mainly on proving the existence of prices with various levels of goodness according to economic criteria. The second category is systems-oriented work and focuses mainly on the design of network pricing architectures that are consistent with today's Internet. Because our focus is on building real systems, we focus exclusively on the systems-oriented work.

2.5.1 Cocchi et al.'s Internet Pricing Work

Cocchi et al. have studied per-byte pricing policies for differentiated service networks. In their work, differentiated service is provided through the setting of two flags: a priority service flag and a no-drop flag. They compare flat per-byte charging, which uses fixed per-byte charges independent of priority flag settings, with priority-based charging, which uses per-byte charges based on the combination of priority flag settings. Workloads consisted of instances of four classes of applications (telnet, ftp, voice, e-mail), each of which had an assumed utility function which was used to set the priority flags. Using packet-level simulations on several simple network configurations, they show [19] that application utilities are universally higher using a simple priority-based scheme (per-byte charges of p plus p per priority flag) compared to flat pricing. Additionally, they also show [20] that prices for arbitrary priority-based pricing policies can almost always be found that (i) are Nash equilibria and (ii) leave all users better off in terms of individual utility compared to flat pricing. In both cases, how to actually compute the prices in the priority-based scheme is still unclear. In addition, their simplifying assumption of demand being independent of price also raises questions.

2.5.2 Edge Pricing

In [79], Shenker et al. make the case for a shift towards more of an architectural focus in network pricing research. The authors argue that too much focus has been paid towards the computation of optimal network prices in simplified models while many architectural and practical implementation issues have gone relatively unexplored. They claim that in the case of the actual Internet and competitive pricing between multiple service providers, the computation of optimal prices is beyond the scope of any practical

algorithm. Furthermore, they assert that price deviations from these theoretically optimal prices produce only second order deviations in total welfare anyway.

Edge pricing is a decentralized approach for network pricing in the Internet [79]. It is based on local pricing policies within service providers and bilateral agreements between service providers with billing at the edges of their networks. It uses a twofold approach to approximate congestion costs. First, it employs time-of-day quality of service pricing based on expected congestion costs. Second, it uses pricing based on source-destination pairs using expected paths as opposed to actual paths to avoid making pricing dependent on routing decisions. Other practical architectural issues discussed include pricing of multicast traffic and receiver-based charging (e.g., a web server sending a web page to a client).

2.5.3 Expected Capacity Allocation

Expected capacity allocation is a decentralized approach to network pricing in the Internet [17]. In this approach, users make service agreements with service providers to obtain network service. Service agreements specify usage profiles that describe resource needs and provide expectations on what type of service to expect. Service expectations are met through the collective efforts of service providers who make local provisioning decisions. Local provisioning decisions are made through service agreements with other service providers based on aggregate resource usage of subscribers. When adequate provisioning is done, sufficient capacity exists to meet user expectations most of the time. When capacity is exceeded (i.e., congestion occurs), packets which have exceeded their usage profiles are dropped in order to meet as many other expectations as possible. A simple implementation approach is proposed based on (i) tagging of packets as being within or outside their usage profiles and (ii) routers that selectively drop packets during congestion based on these tags. The approach is simple and can be incrementally deployed in today's Internet. However, several important issues still remain including support for receiver-based charging, charging for multicast traffic, and concrete feedback signals for non-TCP traffic.

2.5.4 INDEX

Edell et al. have conducted a two-year study that empirically analyzes price and quality elasticities for different qualities of service for Internet access [27, 90]. Through a series of pricing experiments with real users at the University of California at Berkeley, they make four key observations. First, they observe that given the ability to dynamically choose

amongst different qualities of service that users will take advantage of this capability. (The average user used 3.5 of the 5 available qualities of service.) Second, they observe that at all qualities of service, significant price elasticities exist. Third, they observe that as speed goes up, so does corresponding demand (e.g., a six-fold increase in average demand for 128 kbps vs 32 kbps). Finally, they observe that demand from individual users varies significantly (20:1). The high level conclusion is that there is a real demand for differentiated quality of service. Current ISP practice of using flat-rate pricing is shown to cause substantial social inefficiencies with light users subsidizing heavy users. INDEX demonstrates the advantages of alternative pricing schemes and demonstrates a proof-of-concept implementation of a network pricing system.

2.6 Parallel and Distributed Systems

Previous work in market-based resource management for parallel and distributed systems differs from our work mainly in their assumptions, goals, and scope. Previous work assumes competitive sellers pursuing selfish interests while our work assumes cooperative sellers optimizing for social welfare. Work has focused on using markets mainly as a systems organizational principle. In contrast, our work uses markets primarily to increase social welfare. Finally, previous work has consisted mainly of simulations. In the few efforts that produced system designs, few were implemented and none were ever deployed for production use. Our work, on the other hand, consists of both systems work (architecture, two prototypes) and simulations.

2.6.1 Miller and Drexler's Work

The elevator algorithm [57] uses a Vickrey auction [91] for fine-grain, market-based CPU scheduling. Jobs use bids to express CPU resource valuations and obtain CPU resources over time. Bids are time-varying and consist of an initial bid along with a rate which specifies a linear rate of increase over time. At the end of each scheduling quantum, a Vickrey auction is held to determine which job will be selected to run during the next scheduling quantum. The selected job is the job with the highest bid and is subsequently charged the second highest bid, as per the rules of the Vickrey auction. Compared to our work, the elevator algorithm is similar to our rate-based algorithm. Key differences in our algorithm include substantially less implementation overhead and bounds on "bids" for

CPU quantums.

In [57], Miller and Drexler propose a rent-based storage management algorithm. In this scheme, objects pay rent to storage owners in order to remain resident. Rent is determined based on supply and demand and is computed using an Nth price auction based on bids per unit time per unit of storage. Objects obtain funds to pay rent by charging other objects that refer to them retainer and per-usage fees. Over time, the objects collecting the largest fees, and hence delivering the most value, from referring objects are the ones that remain resident. All other objects are evicted whenever they cannot pay their rent. Compared to traditional storage management algorithms, rent-based storage management adds a significant amount of overhead and complexity. Therefore, it appears to be viable only when applied at sufficiently large time-scales and granularities.

2.6.2 Ferguson's Work

In Ferguson's load balancing economy [32], auctions are used to allocate CPU time and network bandwidth to jobs in a distributed system. Jobs are characterized by CPU requirements and bandwidth requirements, the latter of which reflects remote execution costs. Jobs are assigned budgets which are used to purchase CPU and bandwidth resources. Resources are purchased by participating in auctions which award use of resources to bidders with the highest bid amount per unit time. Which auctions to participate in is based on information in bulletin boards which post recent CPU and bandwidth prices (clearing prices) for nodes within some local neighborhood. Through simulation, Ferguson's main result shows that with mean response time as the metric, the load balancing economy performs comparably to traditional load balancing algorithms. No evaluation was made using social welfare as the metric.

In Ferguson's flow control economy [30], tatonement is used on network link prices to compute Pareto optimal bandwidth allocations for a set of competing virtual circuits (VCs). VCs are characterized by preference relations which capture diverse throughput/delay goals. VCs are assigned budgets which are used to purchase bandwidth on network links. Bandwidth is purchased by paying fixed prices computed using an iterative algorithm called tatonement. In tatonement, VCs compute demand sets for bandwidth on all links given current link prices. Link prices are then iteratively adjusted until all excess demand is eliminated. Through simulation, Ferguson's main result shows that tatonement converges to within 2% of equilibrium prices using 10-20 iterations on simple network con-

figurations. However, no claims are made on the quality of the equilibria nor is there any analysis of the algorithmic complexity required for their computation.

In Ferguson's storage economy [31], data replication is driven by local demand from transactions for access to data items. Transactions are read/write sequences on data items. Transactions are assigned budgets which are used to minimize execution time. Transaction cost/speed is the cost/speed of reading and writing data items which depends on whether data is accessed locally or remotely. Data is accessed locally or remotely depending on whether the executing node has local copies of the data. Nodes obtain local copies of data by leasing but are charged a cost proportional to the number of outstanding copies. Based on local demand, nodes decide to make local copies in order to maximize profit. Unlike rent-based storage management, the storage economy appears to be more practical given the state of current computer systems. Still, more work is needed in order for a practical implementation to emerge (e.g., reducing the $O(N^2)$ communication for data item price updates).

2.6.3 Spawn

Spawn uses Vickrey auctions for market-based resource allocation of dedicated CPU time in a distributed system [94]. Applications are tree-based, concurrent applications. Applications have funding rates associated with different nodes of the tree which are used to purchase CPU resources for tasks associated with leaves beneath those nodes. CPU resources are purchased by participating in Vickrey auctions for dedicated slices of CPU time which are held independently by each node. Which auctions to participate in is (presumably) determined randomly since no information is revealed through the Vickrey auction for balancing load. Through simulation and measurement on a prototype implementation, the authors' main result shows that application funding rates are effective in achieving proportional-share resource allocations. This result, however, is limited by Spawn's assumption of concurrent applications that do not communicate (i.e., no coscheduling).

2.6.4 Mariposa

The Mariposa distributed database system [83] uses market-based mechanisms for query execution. In this system, clients submit queries with time-dependent budgets $B(t)$ to brokers. Brokers compute execution plans to execute queries as collections of subqueries and contract this work out to servers. Subqueries are executed on servers that are selected

using either an expensive protocol, which solicits bids from all servers, or a cheap protocol, which selects specific servers based on previous state information. Subqueries using the expensive protocol are scheduled using either a greedy algorithm based on reducing cost or by allocating subbudgets to subqueries and choosing the cheapest, affordable servers in each step. Subqueries using the cheap protocol require no additional scheduling since subqueries are sent for execution without querying servers for bid prices.

Mariposa also uses market-based mechanisms for storage management. Like Ferguson's storage economy, replication is driven by client demand. Servers manage storage fragments which are either exclusive or read only. Servers execute subqueries on behalf of clients using local processing resources and local storage fragments. Bidding on subqueries and fragment replacement policies are based on local, selfish optimizations. Bidding is based on subquery resource requirements, current load, and fragment billing rates which are based on past usage. Storage management is based on fragment revenue history and attempts to keep the most profitable fragments in local storage. In contrast to our work, Mariposa uses market-based resource management primarily as a means to reduce system complexity. How successful it is in that respect, unfortunately, is unclear due to lack of a working implementation at significant scale.

2.6.5 Stoica's Microeconomic Parallel Scheduler

Stoica's microeconomic parallel scheduler [82] uses a first price auction for market-based resource allocation of dedicated CPU time for parallel jobs. In his system, jobs are assigned saving accounts which are used to purchase CPU resources. CPU resources are purchased by participating in a centralized first price auction that allocates CPU time on all processors. The winning job is the job with the highest bid per unit time per processor and is scheduled immediately if enough nodes are available. If enough nodes are not available, the winning job blocks and is charged the time it takes until enough free nodes are available. In other words, there is no backfilling and jobs are charged for head-of-line blocking. Through simulations, Stoica's main result is comparable, and in some cases better, mean system and user response times compared to FIFO and FIFO with backfill scheduling policies. He also shows that income rates are inversely proportional to mean response times and develops a practical architecture which is amenable to implementation. Compared to our work, this work differs mainly in its use of traditional performance metrics as opposed to social welfare in evaluating system performance.

2.6.6 Sairamesh's Work

Sairamesh [78] proposes an architecture for market-based resource management based on tatonement and a level of indirection between buyers and sellers. He assumes groups of buyers have preferences that can be represented using common utility functions. He proposes isolating buyers from the low-level details of the market by having agents represent their interests and participate in the tatonement process. He assumes aggregate buyer demand for individual agents changes slowly so tatonement between agents and sellers can be run on large timescales. He assumes agents implement admission control and tacitly assumes that agents implement fair-share resource allocations on resources obtained through tatonement. The resulting architecture, presumably, simplifies buyer interaction with the system since details of the tatonement process are relegated to agents who obtain resources on behalf of interested buyers.

To demonstrate these ideas, he examines three different examples of this architecture through simulation. The first example examines an economy where agents compete for buffer space and bandwidth on a single network link on behalf of classes of static network clients (i.e., constant demand). The second example extends the first example by examining an economy where network clients can arrive and leave dynamically causing fluctuations in resource demands for agent resources. The third economy extends the second example by examining an economy where multiple links are available for agents to use. Another example, involving clients competing for multiple servers, is also examined but this example is essentially isomorphic to the multiple link case in terms of algorithmic details. In all three cases, tatonement is used between agents and sellers to compute link buffer and bandwidth allocations.

The sole difference between this architecture and Ferguson's load balancing economy is the use of agents as a level of indirection between buyers and sellers. Presumably, this design decision simplifies interaction with the system for buyers, but at the same time it also constrains them. While it is plausible that agents could be implemented with utility functions that have the right shapes for significant numbers of applications, it is unlikely that their specific parameterizations of those utility functions will be the same across the board. To address this, agents would thus have to implement resource allocation mechanisms beyond the fair-share allocations that are tacitly assumed in this work. However, in doing so, they would simply have created another market-based allocation problem beyond the existing agent-seller interactions.

2.6.7 MarketNet

MarketNet [99, 100] advocates use of market-based resource management to improve distributed systems reliability and survivability. In MarketNet, distributed systems are decomposed into independent domains which manage disjoint sets of resources using domain-specific currencies and policies. Domain-specific currencies provide access to resources in associated domains. Domain-specific policies provide control over currency flows in to and out of a given domain. Since currency encapsulates resources rights and policies control currency flows, policies effectively control resource usage. The argument in MarketNet is that by properly limiting currency flows, resource usage can be controlled in such a way that the damage caused by denial-of-service and other hostile attacks can be constrained and thus mitigated. Compared to our work, MarketNet is based on significantly different assumptions and goals. In addition, no evaluation was done to test their hypotheses.

2.6.8 Popcorn

Popcorn [75] is a wide area infrastructure for market-based CPU scheduling of coarse-grain parallel applications written in the Java programming language. In this system, tasks in a parallel application are assigned exponentially-decaying bids which are used to purchase CPU time [2]. CPU time is then purchased from trusted sellers through a centralized market which implements CPU markets using three different market mechanisms: (i) a repeated Vickrey auction, (ii) a sealed-bid double auction (DA), and (iii) a repeated Clearinghouse double auction (k-DA). The system was implemented and was accessible to buyers and sellers through a Web browser. Simulations, however, provide the main results which mainly show that elementary properties of markets hold (e.g., prices respond to changes in supply and demand).

2.6.9 Geweke's Economic Batch Queue

Geweke's economic batch queue [36] uses a Vickrey auction for market-based resource allocation of CPU time for parallel jobs on a cluster. In this system, jobs are assigned bids which are used to purchase CPU resources through a centralized Vickrey auction that allocates CPU time on all nodes. Winning jobs are scheduled immediately regardless of node

[2] Bids are specified as a bid per Java operation to account for heterogeneity in CPU speeds.

availability through use of preemption – running jobs are preempted if they are outbid. The system was implemented through modification of the PBS batch queue system and was deployed on a real system. Unfortunately, demand was too limited to draw any significant conclusions. Instead, results were obtained mainly through simulation which essentially confirm previous results [32, 82] that higher bids yield smaller mean system and user response times. Preemption, the most interesting aspect of this system, was not evaluated at all. Compared to our work, this work differs mainly in its use of traditional performance metrics as opposed to social welfare in its evaluation.

Chapter 3

System Architecture

In this chapter, we present a system architecture for clusters that use market-based resource management. We articulate the three fundamental requirements that market-based systems must meet in order to deliver more value to users. Given these requirements, we examine the resource management problem from an economic perspective. We do this by describing the structure of markets for computational economies that can meet our requirements. We then examine the problem from a systems perspective. We do this by presenting a system architecture that can be used to realize these market structures and is amenable to implementation. Finally, we briefly discuss some systems implementation issues.

3.1 Three Fundamental Requirements

The key element that distinguishes market-based resource management from traditional resource management is extra information, specifically user valuations of the resources being allocated. The three fundamental requirements for market-based systems all revolve around how to obtain accurate valuations from users and how to ultimately translate these valuations into resource allocations which reflect these valuations. These requirements include: (i) providing a simple and natural way for users to express resource valuations, (ii) policies to translate valuations into resource allocations, and (iii) mechanisms to enforce these resource allocations.

3.1.1 Expressing Value

To optimize for value, a market-based system must first somehow determine user valuations of scarce resources being allocated to competing applications. Without this knowledge, there is no meaningful way of inferring how valuable resources are to competing applications when deciding how to allocate resources. Thus, the first requirement for market-based systems is a means to concretely express a user's personal valuation of scarce resources. There are three parts to this: the medium of expression, the functions used to specify value, and the entity (or entities) to which value is being assigned.

To make the notion of user value concrete, we first must have a common medium of expression to express value so that valuations can be handled in a uniform manner. In real systems, all users typically have an implicit notion of how important the execution of an application at some instant in time is to them. A user running a web browser likely values its execution less than an important simulation whose results are needed that hour, for example. To enable the system to handle valuations uniformly, user valuations must eventually be mapped to the same units so they can be compared. Typically, this is achieved through use of a common currency for exchange.

Second, we must specify the set of valid functions that can be used to specify value. Such functions might specify value as a function of time and resource allocations, for example. Resources allocated now are usually more valuable than resources allocated later and so valuations are almost always time-dependent. More of a scarce resource whose allocation is important to an application's performance usually delivers more value and so valuations are also dependent on the quantity of allocations. Using functions to specify value allows for precise specification of a user's valuation. In practice, however, approximations which are easier to reason about and specify are expected since users will likely prefer a simple and natural means of expression even if they are only approximations of their true valuations.

With common units to express value and functions to specify value over time, we must finally specify the entity or entities to which value is being assigned. In the base economy, these entities will usually be low-level resources such as slices of CPU time, network bandwidth, physical memory, and so forth. Users will assign valuations on these entities based on the expected impact that allocation of those resources will have on their applications. For example, in a network simulation, CPU allocations are usually the dominant factor in determining execution time and so a user running such an application would

typically specify a valuation for CPU time that decreases over time.

Ideally, the entities to which we are assigning value should be expressed in the vocabulary of the application (e.g., requests per second versus seconds of CPU time) in terms that the user deals with. For some applications, this is feasible when the mapping between application-specific performance metrics and low-level resource allocations that deliver those performance metrics is well-understood or easily measured. In other cases, this may not hold, and in those cases users must make approximations. For many cluster-based applications, we hypothesize that the basic resources critical for performance are well-known. Furthermore, we also hypothesize that by expressing value on even approximate mappings, significant gains in value can still be achieved.

3.1.2 Translating Value

With user valuations for shared resources in place, the second requirement for market-based resource management is the existence of policies which translate user valuations into resource allocations which are computed to optimize for value. The design of such policies involve trade-offs amongst several factors. From the user's point of view, the two most important factors are that policies be designed to elicit true valuations from the users and that they are simple enough so users can form a reasonable mental model of how the system behaves in response to user actions.

Policies that elicit true valuations from users are the most effective ones from an economic standpoint because they afford the system the best possible opportunity to optimize for value. A policy is said to be *incentive-compatible* when in deciding how to assign resource valuations, a user's optimum utility-maximizing behavior is to reveal true value. Assuming the hyperrational assumption on user behavior from game theory [38] holds, incentive-compatible strategies are attractive because they allow strong statements about user behavior to be made.

In practice, however, we do not know if hyperrationality holds nor do we have quantitative information on the impact strategic users have on overall value delivered. Moreover, the goal of incentive compatibility still must be balanced against other factors, such as providing a simple and natural means of expression, without which users still might express inaccurate valuations due to errors. In addition, given quantitative data on the impact of strategic users, a policy can still be tuned to keep it simple while potentially changing the rules to reduce the negative impact strategic users have on total value (e.g., bounding the

extent/number of reactions that strategic users can have to feedback, such as price changes).

Policies that enable users to form a reasonable mental model of system behavior is also important. While incentive compatibility is desirable to elicit true valuations, policies whose behavior is easy to reason about are desirable to elicit accurate valuations that reflect the user's intentions, regardless of whether or not the valuations being expressed are consistent with true valuations. The latter goal represents a key difference between policies that are optimal in theory and policies which are useful in practice. Policies that are useful in practice are used by real users with varying levels of sophistication. Here, we must balance often conflicting design goals of eliciting true and accurate valuations. It makes no sense to categorically require an incentive-compatible policy if its behavior is so complex to reason about and valuations are so complex to specify that users are unable to accurately express their true valuations to the system.

3.1.3 Enforcing Value

Differential valuations lead to differential allocations which must be enforced by the operating system to provide appropriate service guarantees. Users who pay for and place higher value on resources being contended for ought to receive allocations that reflect these valuations. Modern operating systems, however, are designed with traditional resource management algorithms which make implicit assumptions about user valuations. They optimize internally for global system-centric performance metrics which are embedded in their scheduling algorithms and thus do not have the requisite machinery to enforce differential valuations in flexible ways.

Fortunately, there has been a wealth of systems research on novel quality of service mechanisms for CPU time, networking, physical memory, I/O, spin-locks, and other shared resources. Substantially less work, however, has been done on policies which can effectively utilize these mechanisms. This, combined with the lack of a compelling need to control resource allocations precisely on a single user's desktop, where resources are usually abundant [1], has resulted in quality of service mechanisms making little headway into inclusion in mainstream operating systems. Leveraging the vast quality-of-service literature, market-based systems will thus need to provide new implementations of key operating system schedulers.

[1] This situation stands in marked contrast to cluster computing environments where demand can easily outstrip supply.

3.2 Market Structure

The structure of the market should enable the sale of computational goods which are meaningful for user applications. Sale of such goods should occur at or close to prices which reflect users' true utilities for allocation of these goods in order to maximize desirable social welfare metrics, such as aggregate utility delivered. Structuring a computational market this way involves first deciding what are the goods which are meaningful to user applications and second, how to implement economic mechanisms to map expressions of user utility to allocations of these goods and to ensure that these utilities reflect true user value.

3.2.1 Goods

Goods encapsulate computational resources which are sold to users to run their applications. At the lowest level, the market should provide basic goods which either directly or indirectly through use of third parties (e.g., brokers) enable users to make purchases to effect resource allocations for their applications to maximize their utility. Such goods provide specific resource allocations with various types of service guarantees which are relevant to the majority of the applications that are typically run on the system.

The goods in a computational economy for a cluster should be tailored around the resource allocation needs of applications that constitute the majority of system usage. Providing goods which meet the specific needs of every possible application is tantamount to trying to build a system which provides every possible type of quality of service across all possible resources simultaneously. If common patterns of usage are known, the system should focus its efforts on those cases, since optimizing for the majority of the cases will deliver most of the added value.

Goods should be chosen to accommodate sharing, which in cluster computing environments is often fundamental. Many organizations that use clusters for computing lack the resources or may not find it economical to purchase, construct, and maintain large clusters dedicated to running single applications. Depending on the specific installation, sharing of cluster resources occurs in either time, space (i.e., dedicated access), or both. Even in supercomputing environments where applications are often scheduled on dedicated resources, sharing still occurs in space. Such sharing becomes relevant for economies whenever demand exceeds supply and queues build up in batch submission systems.

Specifically, goods in a computational economy are resource allocations with service guarantees over a well-defined period of time. Unlike tangible goods where ownership is completely transferred from a seller to a buyer, computational goods simply represent the right to use a shared resource where service guarantees determine how resource sharing is done in time and space. User participation in the economy is thus more akin to renting, where prices reflect competition for housing and the quality of the rooms, one dimension of which specifies the level of sharing.

Service guarantees for resource allocations vary mainly in the quantity of the allocation and over what period of time that allocation is delivered. This applies for both low-level resource allocations such as CPU allocations as well as high-level application-level resource allocations (e.g., a resource allocation to support a specific web server hits/second rate). This also applies for markets which support goods such as options and futures [47]. In most cases, we are concerned with the level of sharing and when competition occurs, what fraction of the resources is obtained.

Users purchase goods that encapsulate computational resource allocations for their applications. It is important to note, however, that the purchase of these goods is not necessarily a one-time activity that occurs at the start of job execution. Users have different expectations on how a system behaves during time-shared usage versus dedicated batch usage. Allocation of these goods must be consistent with what users expect. Goods for time-shared allocations, for example, cannot provide hard guarantees on allocations for all running applications for long periods of time without significantly impacting the interactivity of the system. For time-shared applications, multiple goods may need to purchased over time to deliver the resources to execute applications to completion.

The level of abstraction at which users, or agents acting on their behalf, interact with goods in the in economy is also important. Users care about application performance and so goods must be offered that users can reason about to make appropriate purchasing decisions. Choosing an appropriate level of abstraction varies both with the sophistication of the users and the nature of the application. At the lowest level, the economy should provide markets for basic computational resources that have the most significant impact on application performance for the majority of the workload.

Higher-level goods built on these basic goods can be provided by entities, such as application-specific libraries, agents, or third-party brokers which create secondary markets. An application-specific library could translate application-level performance metrics

into required goods with specific service requirements. Agents could be used to monitor application performance and dynamically adjust service allocations to achieve desired performance. Finally, third-party brokers, acting as buyers and sellers, could purchase low-level goods and resell them to provide goods at a higher level of abstraction.

The process of choosing the lowest level goods must be done with knowledge of the anticipated workload. Because of the complexity and infeasibility of providing goods for every resource, combination of resources, and with every type of service guarantee, we will have to make decisions, and sometimes those decisions will prove to either be inaccurate or change over time. In those instances, providing appropriate secondary markets with third-party brokers may be enough to solve the problem by creating goods that address the changes. In other cases, more substantial changes may be required (e.g., offering low-level goods on different resources). Ideally, in those cases, the interfaces presented to users to interact with the economy should stay the same; only the goods and their associated service guarantees should change.

3.2.2 Economic Mechanisms

Given a collection of different types of goods which make sense for the majority of a system's workload, the system must provide markets where exchange for these goods can occur and implement economic mechanisms which specify the rules by which such exchange occurs. Economic mechanisms enable simple and natural expression of resource valuations to make the system easy to use and to reduce errors. In addition, mechanisms ideally should be incentive-compatible, meaning that from a game theoretic perspective, a user's optimal strategy is to reveal his true valuation of the resource.

Economic mechanisms that meet the design goals of simple and natural expression and incentive compatibility allow the system to improve its resource allocation policies in order to maximize value delivered to users. The primary purpose of the mechanisms is to obtain accurate and true valuations of the goods. Given such valuations, the system can optimize for a target social welfare metric, an arbitrary function of user utility functions. Without accurate estimates of utility functions, it will be difficult to perform optimal maximization of social welfare, hence our goals of simple and natural expression and incentive compatibility.

There are many different economic mechanisms for implementing exchange in a market. Scalar auctions (English, Dutch, Vickrey, CDA, etc.) involve buyers and sellers

involved in one or more phases of buyer bid and seller reservation price adjustments leading to one or more winners who are awarded the good. Combinatorial auctions extend this to cases involving combinations of goods, often using completely different algorithms. For markets with multiple goods whose prices are not independent, general equilibrium approaches such as tatonement exist which provide economically efficient solutions given appropriate preconditions.

Choosing an appropriate mechanism for a cluster-based computational economy depends not only on the mechanism's economic properties (normally the binary decision of whether the mechanism is incentive-compatible or not) but also a collection of other factors. These factors include enabling a simple and natural means of expressing value, consistency with expectations for given computing style (e.g., time-shared versus batch), implementation complexity, and computational complexity. Thus, while certain well-studied mechanisms are theoretically quite elegant, the reality of implementing and deploying such mechanisms in real computer systems demands that we take more pragmatic approach which attempts to strike a balance amongst competing requirements.

Expressing valuations of computational goods must be simple and natural for either the user or some entity acting on the user's behalf to reason about. Users may express valuations and interact directly with the market's low-level mechanisms for purchasing goods. On the other hand, application-specific libraries, agents, or third-party brokers may be used to provide a simpler interface for the user. Ultimately though, some entity must interact with the economic mechanisms for markets for basic computational goods and in these cases, the mechanisms need to be simple enough so that programmers can reason about the behavior of these mechanisms to provide higher level services.

Consistency with expectations for a given computing style involve both goods and mechanisms. Goods need to be chosen carefully to ensure that expectations are met. For example, we cannot allow all time-shared goods to have hard resource allocations which are guaranteed for long periods of time without impacting interactive use. In addition, we do not necessarily want to force applications to make large numbers of fine-grained purchases. In the case of fine-grain goods, mechanisms should be designed to minimize the amount of communication required. Another example of a mismatch in expectations is using an auction with long clearing times to allocate computational goods for interactive, time-shared use. The use of market-based allocation should not violate the basic expectations that users already have about how the system behaves.

Implementation and computational complexity of economic mechanisms in distributed systems has received relatively little attention in previous work to date, due to most studies being largely theoretical in nature. Overhead due to large computational costs and communication overheads, scalability of the economic mechanism implementation, and behavior in the presence of node and network faults are the main issues. These issues are all relevant because without an efficient and robust implementation, the benefits of having market-based resource management could easily be diminished by the overheads and fragility introduced by added system complexity.

Finally, it is important to note that in this dissertation the focus is on using market-based mechanisms to optimize for total value delivered. Thus, the sellers in our economy are not designed solely to maximize their profits, unless such a strategy would be optimal in terms of optimizing the desired social welfare metric. In this work, scenarios such as monopoly pricing where some resources go unused even when there is demand for them due to a pricing structure do not occur. In addition, sellers in our economy are cooperative rather than competitive and are processes with system-level privileges (i.e., running as `root`) as opposed to user processes.

Having the option to employ cooperative strategies amongst sellers opens up the possibility of improving value delivered since the space of cooperative strategies is a strict superset of the space of all competitive strategies. A similar argument can be made for sellers optimizing for value delivered as opposed to just maximizing their revenue. Having sellers with system-level privileges is essentially a way to prevent arbitrary third-party brokers from appearing which might violate these assumptions we make about sellers. While users still have the flexibility to design application-specific libraries and agents to assist in their interaction with the market, they are disallowed from creating third-party brokers which act for profit. Ensuring that all sellers optimize for value delivered and are cooperative simplifies the design and allows us to make assumptions that would not hold in general.

3.3 Architecture

This section presents a system architecture for market-based cluster resource management that addresses the requirements in Section 3.1 and can be used to implement market structures as described in Section 3.2 for goods that are meaningful to user applications. The architecture consists of five layers (Figure 3.1): *resources*, computational resources such

as CPU, physical memory, and disk which are packaged as goods that encapsulate shares, proportional-share resource allocations; *resource managers*, operating system entities which enforce resource allocations; *economic allocators*, sellers which translates competing valuations of shared resources into resource allocations and charge users for resource usage; *access modules*, the means by which users assign resource valuations and run applications on the system; and *users and applications*, the buyers which use the system and ultimately drive the design decisions in all other layers.

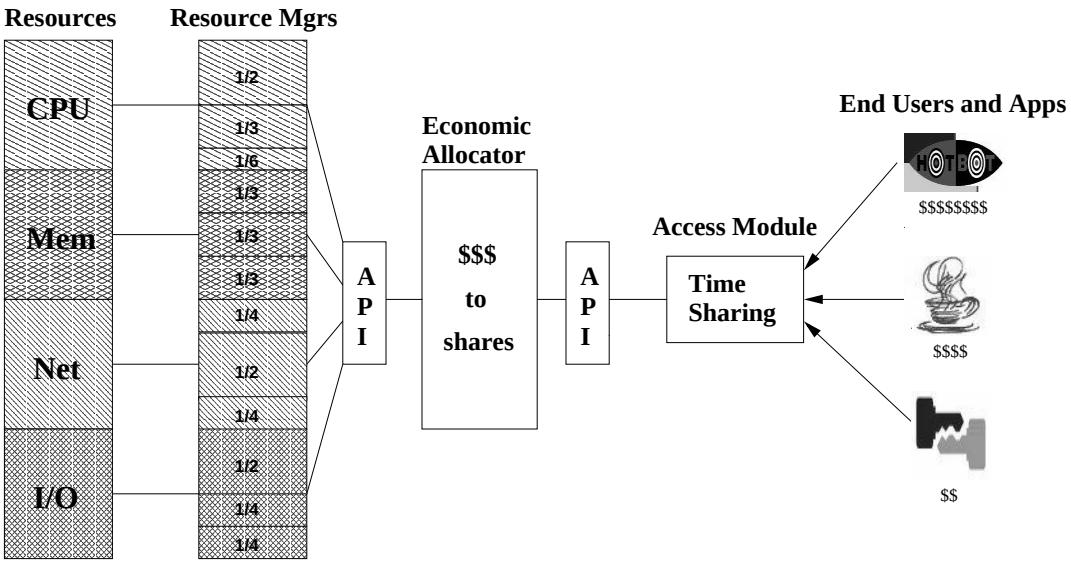

Figure 3.1: **End-to-end System Architecture.** End users (buyers), running a wide range of applications, assign valuations for goods that encapsulate resource allocations to use for their applications, which run remotely on the cluster using the appropriate access module. Access modules discover the economic allocators (sellers) with the best prices, purchase goods on behalf of users, and remotely execute jobs through the economic allocator. The economic allocator translates competing jobs' valuations, expressed in a common currency, into goods that encapsulate resource shares which reflect competing users' valuations for the shared resource. Resource share allocations are enforced using resource managers, one per resource type.

3.3.1 Resources

Resources are the computational resources that are controlled by the resource managers and allocated to applications to deliver value to users. Operating systems manage a variety of resources, all of which are candidates for allocation within the computational

economy. The economy, on the other hand, focuses exclusively on scarce resources, specifically those which have significant implications on application level performance for the majority of the system workload. If a resource is not scarce, an economy is unnecessary since there is always sufficient supply to satisfy demand. If a resources is scarce, then that resource is relevant only if allocation of that resource has a substantial impact on application performance. In all other cases, resource allocations do not have a significant impact on overall performance of the system and thus any improvements made there will translate into, at best, small overall improvements in social welfare.

We view all computational resources in a cluster as share-based allocations over some period of time according to the proportional-share scheduling model. In this model, each resource has a collection of tickets that encapsulate the aggregate resource rights to that resource. A process holding t tickets on a resource with T total tickets is guaranteed an allocation of t/T of that resource as long the ratio t/T remains unchanged. Applying this model to real computing resources, we have proportional time-shared slices of CPU time, space-shared allocation of physical memory and disk space, and proportional shares of networking and I/O bandwidth. Resource shares and bundles of shares across different resources provide a simple and flexible substrate which can be used to construct a variety of different types of goods for a computational economy.

3.3.2 Resource Managers

Resource managers are the operating system entities which provide resource allocations with associated performance guarantees for goods in the computational economy. Numerous algorithms to implement such entities already exist for a variety of different types of guarantees and for computational resources such as CPU time [11, 40, 51, 95, 96], memory [45, 93], network bandwidth/latency [24, 41, 93, 102] and I/O bandwidth [9, 80, 93]. The quality of service literature is rich with the design, analysis, and implementation of such algorithms. The deployment of such algorithms in real systems, unfortunately, has been limited due largely to the lack of policies to control how applications can make use of these quality of service mechanisms. In the absence of proper incentives, there is nothing aside from social conventions that will stop every user from claiming that his/her application needs the highest quality of service.

In our architecture, we use goods that encapsulate proportional-share resource allocations and therefore our resource managers must enforce the proportional-share scheduling

contract. This process entails guaranteeing that each competing process i, holding t_i tickets on a resource with T total tickets, obtains a t_i/T share allocation of that resource over time as long as t_i/T remains unchanged. Proportional-share algorithms to enforce these types of allocations with different trade-offs between allocation precision and overhead already exist for allocation of CPU time [11, 54, 63, 95, 96], physical memory [93], network bandwidth [24, 93], and I/O bandwidth [93]. [2] Concrete implementations of computational economies can make direct use of these algorithms.

3.3.3 Economic Allocators

Economic allocators are the sellers, the entities which provide the goods and implement the economic mechanisms of the market. Economic allocators have several responsibilities. First, they are responsible for announcing their availability and making relevant details of their state available to access modules so that access modules can match buyers with appropriate sellers. Second, they are responsible for translating user valuations of the resources into goods with specific resource allocations according to the rules of the economic mechanism used. Third, given these resource allocations, the economic allocator must use the resource managers to enforce the service guarantees associated with the allocations.

Economic allocators must announce their availability in order for users to find and eventually use their resources. Furthermore, they must also make relevant details (e.g., current prices, goods currently available) available so that users can not only be aware of their existence but also their state so they can decide which sellers to perform exchanges with. Providing this functionality in a scalable and robust manner is no different from providing the same functionality in traditional resource management systems where the problems of discovering what resources are available and what their state (e.g., current load) is also exist. Because solutions to these problems have been studied in previous systems, we simply borrow and adapt existing mechanisms rather than try to develop entirely new ones.

An economic allocator's main purpose is translating user valuations into goods with specific resource allocations and charging users while trying to maximize some social welfare metric. This functionality, which can vary significantly in complexity depending on the economic mechanism used, lies at the heart of the architecture. It is this func-

[2] The performance implications of concurrent proportional-share allocations of different resources has not well-studied, however.

tionality that provides the main distinction between market-based systems and traditional systems. Traditional systems allocate resources according to policies which optimize for system-centric performance metrics which make implicit assumptions about what users value. Market-based systems, on the other hand, make user value explicit and use these as inputs to economic-based resource allocators that compute and enforce resource allocations while optimizing for arbitrary functions of competing users' valuations of the resources.

With resource allocations computed, the economic allocator must enforce these resource allocations using the mechanisms provided by the resource managers, in our case proportional-share schedulers. The service guarantees associated with the goods determine the allocations and the timescales over which these allocations are enforced and also whether queueing is needed to prevent overselling of resources. On one extreme, time-sharing using rate-based resource allocation uses allocations that can change with very little delay, where allocations must be continuously renewed, and are guaranteed at the granularity of a small number of scheduling quantums. On the other extreme, batch processing usually involves resource allocations for large blocks of dedicated CPU time. The longer allocations are guaranteed for, the more likely queueing will need to occur due to the system's commitment to deliver the resource allocations associated with goods already purchased by a running application.

3.3.4 Access Modules

Access modules provide the means by which users express value to the system and access cluster resources to run applications. They encapsulate the details of discovering sellers offering the best prices, the protocol to interact with sellers to obtain resource allocations, and the remote execution protocol to perform remote computations on those resources. Subsets of this functionality can already be found in existing and previous systems, albeit with certain limitations. Examples include `rsh/ssh` for interactive remote execution of sequential applications, custom remote execution environments such as GLUnix [37] and Score-D [46] for remote execution of parallel applications, and batch queue systems such as PBS for running large numbers of sequential and/or parallel jobs. In our architecture, users state the types of goods their applications need and their valuations for these goods and leave the rest of the details to the system.

Access modules automate the process of keeping track of sellers and monitoring their state to provide users with access to sellers with the best prices. Such automation is

similar in functionality to services provided by computerized markets for securities such as the NASDAQ in which the best ask prices for all sellers (Level 2 quotes) for a given stock are available. It is also similar to traditional resource management systems which typically use a daemon to keep track of load and resource availability on all the nodes in a cluster. In both cases, we are essentially solving discovery and information dissemination problems, albeit with slightly different requirements. Because these problems have been studied in the past, we simply leverage existing work in this aspect of the system.

Once appropriate sellers have been found (i.e., typically the sellers with the lowest prices), users must interact with these sellers to purchase computational goods to run their applications. The specifics of these interactions depend on the implementation of the economic mechanisms, details which should be largely hidden from users. The only responsibility that the user should have is to express a valuation to the system that is meaningful for the economic mechanism used in the market. Assigning value may be as simple as specifying a scalar value that is passed interactively on the command line or may involve more complex specifications (e.g., specifying a utility function). These valuations close the loop, in that future workload demand and valuation depends on past performance.

With goods purchased from appropriate sellers, applications must then be executed remotely using the resources encapsulated by those goods. We have chosen to place this functionality in the access module. In this model, the process of purchasing goods and running an application that uses those goods occurs immediately in a single step. The main reason for this design decision is our lack of support for goods such as options and futures on computational resources. In most cases, we expect users to purchase goods and use them immediately to run their applications. Given such an assumption, it makes little sense to separate the process of good purchasing and claiming those goods to deliver resources to a user's application. Another important reason for this decision is that it deters users from buying goods and creating third-party sellers. With all sellers created and managed by the system, we preserve our assumptions of social welfare maximizing behavior and cooperative behavior amongst sellers.

3.3.5 End Users and Applications

End users and applications are the two most important parts of the end-to-end picture because it is the users' resource valuation distributions and application resource demands over time that drive the design decisions at the lower layers of the system. Users,

along with their application resource demands, determine what the appropriate goods and mechanisms are in the computational economy. Their distributions of valuations over the goods, along with their demands for these goods relative to supply over time, then determine whether market-based resource management can be beneficial compared to traditional resource management and the extent to which social welfare can be improved.

For clusters used primarily for scientific computing and simulation, especially in academic and research environments, a significant fraction of system usage consists of long-running, compute-intensive applications which are well-suited for resource allocation via goods that encapsulate shares of CPU time. Since running time is often directly related to an application's CPU allocation, CPU shares are easy for users to reason about. As long as there are no significant externalities (e.g., memory is also limited and paging occurs), then using a market for CPU shares to mediate access cluster resources should be sufficient to effectively manage competition to scarce CPU resources.

How effective these CPU markets are in improving social welfare will depend on the distributions on CPU valuations and demand relative to supply over time. If all valuations are identical, then allowing users to express valuations to the system yields no new information, and thus a fair-share algorithm should perform just as well as a market-based one. In general, however, we expect there will be variation in user valuations for CPU time. The more variation, the more opportunity market-based systems have for optimizations. Even if all valuations are the same, however, market-based systems still can improve performance through their use of a concrete feedback signal (prices) which captures the degree of competition for scarce resources and which cannot be ignored. If demand exceeds supply, regardless of the distribution on CPU time valuations, a market-based system can still outperform a non-market-based one through use of feedback, which alters user behavior and keeps the system in a comfortable operating regime.

3.4 Implementation Issues

Market-based systems must effectively address practical systems implementation issues in order to fully realize their advantages in delivering higher social welfare compared to traditional systems. Among these issues, market-based systems must avoid introducing substantial computational and communication overheads. They must either maintain or improve the scalability and robustness of the system compared to traditional systems. Check-

point/restart and process migration, techniques commonly used to improve fault-tolerance and load distribution, may need to be employed to achieve maximal improvements in social welfare. And finally, trust and mechanisms to address the lack of trust are also important.

3.4.1 Computational and Communication Complexity

The resources necessary to implement market-based resource management is an important issue because the more overhead needed to implement the market structure, the fewer resources are available for allocation to user applications. The structure of the market, the goods and the economic mechanisms, is the main determining factor. Specifically, we would like to avoid having to use complex algorithms that involve complex computations, multiple rounds of communication with many distributed entities, and unknown bounds on time to convergence. While such algorithms may ultimately prove to be the optimal ones, it is prudent to first consider and thoroughly evaluate the efficacy of the simplest possible algorithms that could help to substantially improve social welfare. This is the approach taken in this dissertation.

3.4.2 Scalability and Robustness

Using market-based algorithms in place of traditional algorithms for resource management should not adversely affect the scalability of the system. It makes no sense to replace traditional algorithms that scale well with market-based ones if the market-based system cannot scale to handle the very types of systems whose performance we were trying to improve. Given a system that has some scalability behavior with a traditional algorithm, the switch to a market-based algorithm should either maintain those properties or improve upon on them. One of the much touted properties of market-based algorithms is their amenability to decentralized implementation. While future work may attempt to exploit this property in ways which allow market-based implementations to scale in new ways compared to traditional algorithms, in this dissertation, we seek primarily to maintain the status quo using well-known techniques.

Market-based algorithms should also either maintain or improve system behavior in the presence of node and network faults. In large cluster-based computer systems, failures are inevitable and any robust resource management system must be able to cope with these types of failures in a manner that allows the system to continue operation. Ideally, we seek to achieve graceful degradation in performance when nodes become unavailable

rather than an abrupt disruption in system service. This goal becomes especially pertinent when considering implementations of certain market-based algorithms such as those based on general equilibrium theory (e.g., tatonement) which require many rounds of communication and coordination amongst multiple entities, which may fail. Implementation of such algorithms is already complex but becomes even more difficult to reason about when the algorithm must be robust against faults. As with scalability, this dissertation does not focus on new techniques to improve robustness against node and network faults. Rather, we simply attempt to maintain similar levels of robustness achieved by traditional algorithms.

3.4.3 Preemption

Preemption allows a resource management system to mitigate the effect of bad scheduling decisions in light of new information, such as the arrival of jobs with substantially higher resource valuations than the applications currently occupying system resources. In situations where the service guarantees associated with goods permit reduced or temporarily suspended resource allocations, the ability to preempt creates a new class of opportunities to help improve social welfare. Market-based systems that support preemption improve social welfare by evaluating runnable jobs and using their estimated impact on social welfare along with preemption costs to decide which jobs to keep running and which jobs to preempt.

Mechanisms to implement preemption are abundant in the systems research literature. Most of these mechanisms were designed with the intent of improving application fault-tolerance while others were designed for managing load imbalance. Whatever the case, these mechanisms have direct applicability for our purposes. Checkpoint/restart [33, 53, 69, 70, 74, 86, 101] and process migration [26, 87] are the two main types of preemption and essentially involve taking snapshots of application state and recreating that state at a later time, potentially on different resources. Implementations typically involve non-trivial systems level machinery in order to provide general-purpose use but have successfully been implemented in research systems.

Another mechanism for a limited form of preemption and which is available in modern operating systems is explicit process suspension. In Unix operating systems, for example, the runnability of a process can be explicitly controlled at user-level by sending the process SIGSTOP (suspend) and SIGCONT (continue) signals. While suspended processes still consume some resources while suspended, they also completely refrain from using others during the duration of their suspension and this may enable optimizations when such

resources are scarce. CPU time and physical memory, once physical memory pages have been paged out, are two examples of resources that are not consumed while suspended. Virtual memory, conversely, is an example of one that is.

3.4.4 Trust

Market-based cluster resource management makes the issue of trust tractable by assuming that all resources are controlled by a single trustable entity or a collection of trustable entities. When such trust is in place, buyers need not concern themselves with seller integrity, specifically whether the goods delivered have service guarantees that are consistent with the goods that were purchased. Without trust, additional mechanisms would be required to verify key properties of delivered goods. While implementation of such mechanisms is feasible for most important properties, the cost of deploying such mechanisms in real systems may adversely impact system performance since utilizing those mechanisms consumes resources.

One example of an important property that warrants verification in systems where trust cannot be assumed is integrity. Integrity concerns whether the goods delivered, minus the service guarantees, are indeed the goods that were purchased. An integrity check for computation, for example, might involve purchasing multiple goods for the same application and verifying identical end results. Such checks, unfortunately, are expensive and also do not generalize to other computational resources such bandwidth, except in those cases where the "application" being run is a bandwidth benchmark designed specifically for the purpose of such checks.

A more efficient class of mechanisms for checking seller integrity involves execution of integrity check applications at randomized times. Rather than apply checks for all applications, this class of mechanisms uses randomization to reduce the resources consumed by integrity checks at the cost of potentially missing some wrongdoing. Because sellers lack the knowledge of the random number generator used to compute when integrity checks run and assuming integrity-checks can check all important properties, such a scheme may be sufficient to prevent sellers from trying to swindle users while using only a small amount of resources. [3] Integrity checks might include running multiple applications and checking

[3]Such a scheme might be used for integrity checks when computational economies span multiple administrative domains. Grid computing systems are a good example of systems that could benefit from market-based resource management and where integrity checks are essential since sellers, while not necessarily completely untrustworthy, are neither completely trustable either.

their results to verify integrity or running benchmark applications to verify that service guarantees are being met.

Chapter 4

Prototype Implementations

In this chapter, we describe the design and implementation of two prototype systems we have built that use market-based resource management. Based on the system architecture in Chapter 3, these prototypes demonstrate that our architecture is both natural and feasible to realize through example. We begin our discussion of these prototypes by first describing REXEC, a remote execution system that provides the distributed systems infrastructure needed to build computational economies for clusters. With REXEC as the foundation, we then present the two prototype systems: (i) a time-shared computational economy for proportional-share CPU time and (ii) a batch computational economy for dedicated CPU time. Both systems were implemented and deployed on subsets of the UC Berkeley Millennium cluster, a 300+ CPU cluster of symmetric multiprocessors, and were subjected to use by a real user community.

4.1 REXEC Remote Execution System

REXEC is a remote execution system for running parallel and sequential jobs on clusters and was built to provide the distributed systems infrastructure for building computational economies. Building on previous work in remote execution systems and practical experience with a large research cluster, it provides a myriad of useful features which are used to provide scalable, robust, and flexible remote execution. Its implementation consists of a collection of user-level programs, client programs and daemons, along with only small modifications to the Linux kernel [1]. The system is currently installed and has been in

[1] The kernel changes are minimal. Running REXEC on an unmodified kernel still delivers most of its value. The only difference involves cleanup in the special case where a process in a process hierarchy becomes

production use for over a year on the entire UC Berkeley Millennium cluster, a 300+ CPU cluster of symmetric multiprocessors.

4.1.1 Design Goals and Assumptions

In this section, we state our design goals and the assumptions we made in designing REXEC. Our design goals are largely based on several years of practical experience as users of the GLUnix remote execution system [37] on the Berkeley NOW cluster, a 115-node cluster of Sun UltraSPARC machines. They are also based on a comprehensive examination of previous systems work in remote execution and our desire to combine and extend the most useful features from this body of work in a unified system. Our assumptions are typical of remote execution systems and not overly restricting. Modern clusters built using off-the-shelf hardware and contemporary operating systems are easily constructed to satisfy these assumptions.

Goals

High availability. The system should be highly available and provide graceful degradation of service in the presence of failures. No single point of failure should exist, nor should there be any artificial dependencies between failure in one part of the remote execution system and a perfectly functional node (e.g., by having a centralized system coordinator). Any node that is functional and running the REXEC system software should be available to run applications through the REXEC system.

Scalability. Performance of the remote execution system should scale as the size of the system grows. More specifically, as more nodes are added and more applications are run on the system, remote execution overhead should scale gracefully.

Transparent remote execution. Execution on remote nodes should be as transparent as possible, given the implementation approach (e.g., kernel-level implementations can achieve greater transparency than user-level implementations). At a minimum, signals, `stdin`, `stdout`, `stderr`, and the user's local environment (current working directory, user ID, group ID, environment variables) should be propagated. In addition, local job control should be applicable for controlling remote jobs. An implementation of this core set of

inherited by `init` and cannot be killed given the PID of the root of the hierarchy. In this case, `C-c` without the kernel modification could leave straggler processes.

features provides sufficient transparency for a large number of applications. File access is another important issue and is discussed later in this section.

Minimal use of static configuration files. The system should rely on as few static configuration files as possible. In particular, static files related to cluster configuration should be avoided. In our experience, node membership in a large cluster varies over time. Membership changes as machines are added, removed, crash, rebooted, shut down, etc. Having to manually update configuration files every time a change occurs is error-prone and inefficient and does not scale. The configuration of the cluster should be discovered dynamically.

Decoupled discovery and selection. The process of discovering which nodes are in the cluster and what their state is should be separated from the selection of which nodes to run an application on. This gives users flexibility to apply their own criteria to select which nodes are best suited for running their applications. For example, one user might want to select the nodes with the lowest CPU utilization, another might want to select nodes based on physical memory available, and yet another might want to run on a specific set of nodes independent of their state.

Well-defined failure and cleanup models. The system should provide well-defined models for failure and cleanup. Remote processes controlled through a local point of control introduce new failure modes (e.g., network connection failure from a local proxy process to a remote process). Remotely controlled parallel or distributed processes add additional failure modes (e.g., one process in a n process program crashes). In addition, there are practical issues as well that arise due to user-level implementations (e.g., cleanup for remote processes that fork, child processes whose parents may exit causing the child to be inherited by `init`, etc.). A user-level remote execution system that supports parallel jobs should extend the failure and cleanup model of local sequential execution to account for these differences.

Parallel and distributed program support. The remote execution environment should provide a minimal set of hooks that allow parallel and distributed runtime environments to be built. For example, when running a parallel program, each remote process in the program often needs to know how many processes comprise the program, that process's rank in an ordering of those processes, and information which can be used to resolve network addresses of those processes so communication can be established.

Security. The system should provide user authentication and encryption of all communication. For clusters that are unprotected by firewalls, strong authentication prevents outside attackers from launching brute-force attacks on weak security or reverse engineering and exploiting schemes that rely on security by obscurity (e.g., GLUnix authentication was based simply on the UID passed from the client process to the master). For internal attackers, encryption prevents users from snooping the network to discover passwords, an increasingly common occurrence in large public institutions.

Assumptions

Uniform file pathnames. We assume that all shared files are accessible on all nodes using the same pathnames. We also assume that most local files on each node are also accessible using the same pathnames (e.g., `/bin/ls`). Clusters today are built with nodes that run mainstream operating systems (e.g., Linux, Windows). Since none of these operating systems implements a global cluster filesystem (see Sprite [67] or Solaris MC [52, 81]), each node typically uses local filesystems and some of number of network file systems which are attached at common mount points. Our system, which uses `ext2fs` for local filesystems and NFS filesystems for network file access, has this configuration.

All nodes run the same OS with compatible software configurations. We assume all nodes in the cluster run compatible versions of the operating system and have compatible software configurations. Compatible versions means that application binaries are supported on different versions of the operating system (e.g., Linux 2.2.5 versus Linux 2.2.10). Compatible software configurations means that all the necessary support for applications is available for applications. Having the same version of the operating system is not sufficient (e.g., shared library version mismatches). We do not attempt to address heterogeneity issues that Grid computing systems [34, 43] or systems like PVM [85] attempt to address. In our system, all nodes run the same, slightly modified version of the Linux 2.2.5 operating system under the same configuration (identical RPMs installed, same mount points for NFS filesystems, etc.).

Common user ID/account database. We assume each user has a unique user ID and an account which is the same on all the nodes in the cluster. This is implemented by having all nodes in the cluster use the same NIS domain. Assuming unique user IDs and accounts on all cluster nodes is a reasonable assumption for a system on the scale of a machine

room, building, or campus. Again, we do not attempt to address the heterogeneity and cross administrative domain issues of wide area Grid computing systems. Requiring a user account on all the cluster nodes is not terribly restrictive and can be implemented fairly easily on the class of systems we address. Furthermore, it does not necessarily preclude a user without an account from accessing resources on the cluster. Access to the cluster could, for example, be made available through well-defined services for popular applications whose server-side implementations run using a real user account. Alternatively, with proper security, guest accounts could also be offered.

4.1.2 System Architecture

The REXEC system is based on a simple system architecture that revolves around three main programs: `rexecd`, a daemon which runs on each cluster node; `rexec`, a client program that users run to execute jobs using REXEC; and `vexecd`, a replicated daemon which provides node discovery and selection services (Figure 4.1). A fourth program, `millstat`, provides details on the state of the entire cluster, including how many and which nodes are available along with their state (e.g., CPU load). The basic mode of operation is users use `rexec` client to run programs. They use information from specific `vexecd` servers to obtain identities of the "best" nodes (e.g., lowest CPU load). Using this information, they then establish connections with `rexecd` daemons on those nodes and remotely execute the user's program. In the next section, we go into detail on how REXEC does all of this in a scalable, robust, and flexible way.

4.1.3 Key Features

Building on previous work in remote execution systems and practical experience with the Berkeley NOW cluster, REXEC provides a myriad of features which improve its scalability, robustness, and flexibility. The key features of REXEC include decentralized control, transparent remote execution, dynamic cluster membership, decoupled node discovery and selection, a well-defined error and cleanup model, support for sequential as well as parallel and distributed programs, and user authentication and encryption. In this section, we provide details on the implementation of these features and show how these features are used to address our design goals.

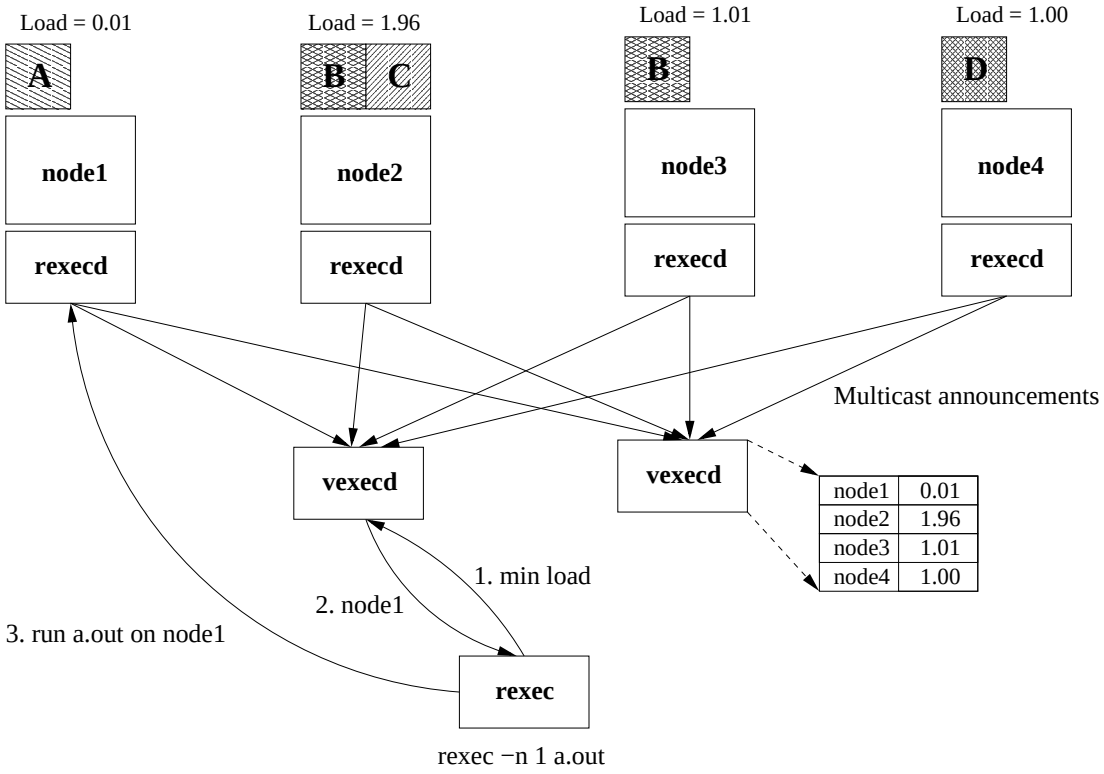

Figure 4.1: **The REXEC Remote Execution Environment.** The system consists of three main programs: `rexecd`, a daemon which runs on each cluster node; `rexec`, a client program that users run to execute jobs using REXEC; and `vexecd`, a replicated daemon which provides node discovery and selection services. Users run jobs in REXEC using the `rexec` client which performs two functions: (i) selection of which nodes to run on based on user preferences and (ii) remote execution of the user's application on those nodes through encrypted TCP connections to node `rexecd` daemons. In this example, there are four nodes in the system: node1, node2, node3, and node4 and two instances of `vexecd`, each of which implements a lowest CPU load policy. A user wishes to run some program `a.out` on the node with the lowest CPU load. Contacting a `vexecd` daemon, `rexec` obtains the address of the machine with the lowest CPU load, node1. `rexec` then establishes a TCP connection directly to that node to run `a.out` remotely.

Decentralized Control

REXEC uses decentralized control for graceful scaling of system overhead. As clusters scale up in size, demand also tends to scale to match the increase in system capacity. More demand usually means more applications are running concurrently on the cluster which in turn means that more instances of the `rexec` client are running. More instances of `rexec` client implies more computation and resources are needed to manage the remote interactions between `rexec` clients and `rexecd` daemons. As the number of these interactions increases, this can have adverse effects on system scalability.

REXEC distributes the computation and resources for client to daemon interactions by having each instance of the `rexec` client use local resources on the client and direct TCP connections to manage its interactions with `rexecd` daemons. By using direct client to daemon connections, computation and resources for remote execution are distributed across many `rexec` instances and node `rexecd` daemons. With a large number of nodes, having a centralized entity act as intermediary between users and cluster nodes can easily become a bottleneck as single node resources become an issue (e.g., kernel limitations on # of sockets). Our scheme avoids this problem by distributing this work, thereby increasing system scalability.

In addition to scalability, our decentralized design by definition avoids single points of failure. By freeing users from depending on intermediate entities to access cluster nodes, we ensure that any functional node in the system which is reachable over the network and running an `rexecd` daemon can always be used to run user applications. This approach is a significant improvement over previous systems such as GLUnix [37] and SCore-D [46] where failure of a centralized coordinator causes all cluster resources to become unavailable. In practice, centralized entities with no backup or failover capabilities can significantly decrease system availability.

Transparent Remote Execution

REXEC provides transparent remote execution which allows processes running on remote nodes to execute and be controlled as if they were running locally. REXEC uses four mechanisms to accomplish this: (i) propagation and recreation of the user's local environment on remote nodes, (ii) forwarding of local signals to remote processes, (iii) forwarding of `stdin`, `stdout`, and `stderr` between the `rexec` client and remote processes

and (iv) local job control to control remote processes.

First, REXEC propagates and recreates the user's local environment on each remote node. To do this, the `rexec` client packages up the user's local environment including the current working directory, environment variables, user ID, group ID, program to execute, and its command line arguments. Then, as part of the remote execution protocol, it sends this information to each `rexecd` on each node that has been selected to run the user's application. Upon receiving a new connection request from an `rexec` client, each `rexecd` spawns a new client thread to manage that particular `rexec` instance (Figure 4.2). Before forking and execing the user's program, this thread then unpackages the user's local environment and recreates it using standard C library functions (e.g., `chdir`, `setenv`, `setuid`). Each remotely forked process will then execute with the same local environment as the machine where the `rexec` client is running.

Second, REXEC forwards locally delivered signals to each remote process. REXEC does not implement a true global PID space. Instead, to send a signal to remote processes, users send the signal to the local `rexec` client which acts as a proxy to all remote processes. Upon receiving a signal, `rexec` forwards that signal to an `rexecd` thread on each node the application is running on. That thread will then send the appropriate signal to the local process running on that node. (This thread also is also responsible for forwarding `stdin`). REXEC currently forwards most of the 30 or so Unix signals supported in Linux with a few exceptions. `SIGUSR1` and `SIGUSR2` are used by the pthreads library and so these signals cannot be caught without breaking the pthreads implementation. `SIGINT` and `SIGTSTP`, which are related to job control, are handled differently as discussed later. Finally, because REXEC is a user-level implementation, `SIGKILL` (if the `rexec` clients crashes, the same effect is achieved through REXEC's error detection and cleanup mechanisms as discussed later) and `SIGSTOP` on the `rexec` client cannot be caught.

Third, REXEC forwards `stdin` from the `rexec` client to remote processes and forwards `stdout` and `stderr` from remote processes to the `rexec` client. For `stdin`, there is an additional thread in `rexec` that is responsible solely for forwarding `stdin` to remote processes. (`stdin` is broadcast to all remote processes.) On the remote side, the signals and `stdin` forwarding thread receives `stdin` messages and forwards them to remote processes through a Unix pipe. For `stdout` and `stderr`, `rexecd` has per-rexec client `stdout` and `stderr` threads which use Unix pipes to read output from the remote process and forward

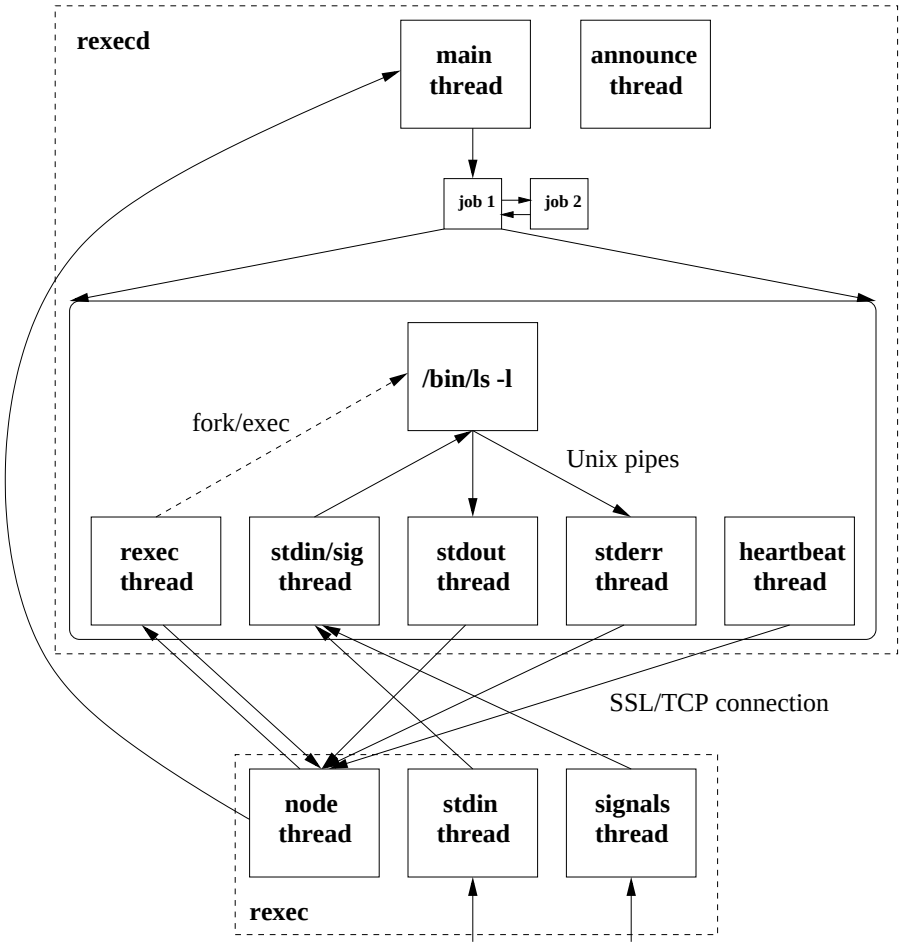

Figure 4.2: **Internal Thread Structure and Data Flows for `rexec` and `rexecd`.** `rexec` consists of a stdin thread for forwarding of `stdin`, a signals thread for forwarding of signals, and one node thread per node for managing remote process execution including printing remotely forwarded `stdout` and `stderr` to the user's local terminal, receiving heartbeat packets (and `rexec` client monitoring of the SSL/TCP connection), and receiving the exit status of the remote process. (In this example, `rexec` is running '`/bin/ls -l`' on a single node so there is only one node thread.) `rexecd` consists of a main thread which creates new threads for new `rexec` clients and maintains a list of running jobs, an announce thread for sending multicast state announcements, and a collection of per-rexec-client threads. These per-rexec-client threads include a main `rexec` thread for the client, the user process forked and execed by the main `rexec` thread, a `stdin`/signals thread for forwarding `stdin`/signals from `rexec` to the user process, `stdout` and `stderr` threads for forwarding `stdout` and `stderr` from the user process to `rexec`, and a heartbeat thread for sending of periodic heartbeat packets to detect failures in the SSL/TCP connection between `rexec` and `rexecd`.

this data back to the `rexec` client [2]. For each remote node, `rexec` has a thread which, among other things, reads remote `stdout` and `stderr` and prints output locally to the user's terminal. Each line of `stdout` and `stderr` is prefixed by the node number of the remote process that printed it (e.g., node 0 that prints "hello world" appears as "(0) hello world").

Fourth, REXEC allow users to use standard job control to control remote processes in their applications. Just as we use the `rexec` client as a proxy to deliver signals, we can use it as a proxy which can be used to control remote processes through job control. More specifically, REXEC supports `C-c`, `C-z`, `fg`, and `bg` commands from the shell to deliver `SIGINT`, `SIGTSTP`, and `SIGCONT` signals. Because REXEC is implemented mainly at user-level, job control and the delivery of the appropriate signals sometimes requires approximating the desired effect. For example, typing `C-z` to a local process delivers `SIGTSTP` to the process which suspends execution of that process. Delivering `SIGTSTP` to a remote process, however, is not meaningful since that process is not attached to a terminal. Thus, for `SIGTSTP (C-z)` and `SIGINT (C-c)`, `rexec` catches these signals and delivers `SIGSTOP` and `SIGKILL` respectively.

Our liberal use of threads, in particular on the client side, greatly simplified the implementation but also has implications for scaling. Finite operating system resources (e.g., fixed-size process table, # of open file descriptors) combined with large numbers of `rexec` clients running on front-end machines can lead to taxing of machine resources. The scalability of front-end operating system algorithms, such as thread scheduling, could also become bottlenecks, since local operating system schedulers were not designed to accommodate efficient scheduling of large numbers of runnable processes. As the number of clients grows, slowdowns and eventually failure will occur when resource limits are reached. One solution to this is to simply increase operating system limits at kernel compile time. This helps alleviate the resource limits problem but does not address the scalability of operating system algorithms. To address both resource limits and algorithm scalability, a hardware alternative is to simply add more front-end machines and use a hardware load-balancer (e.g., Cisco's Local Director [16]) to distribute user logins across a set of front-end machines.

An alternative implementation strategy, which requires no additional hardware, is to use an event-driven architecture based on I/O multiplexing. Rather than have three

[2] Note that due to the way `stdout` is buffered, applications that run on REXEC must use `fflush` statements if they want `stdout` to be written immediately to the `stdout` pipe so `stdout` is displayed immediately on the user's terminal.

threads per remote node, the `rexec` client could instead use one thread for each thread's function (node thread, `stdin` thread, signals thread) and use I/O multiplexing via the `select` system call to multiplex requests and replies for communication with `rexecd` threads running on remote nodes. Using this strategy, resource demand on front-end nodes from `rexec` clients would scale with the number of running applications, as opposed to the number of applications scaled by their individual degrees of parallelism. The use of I/O multiplexing to handle communication across very large numbers of concurrent network connections has been successfully used in a number of high profile projects, including the Apache web server and Harvest web cache. In addition, recent research [6] has also shown how the implementation of `select` can be scaled to efficiently handle I/O multiplexing across hundreds or thousands of file descriptors. Given these previous successes, it seems likely that I/O multiplexing could also be used successfully for our purposes.

Dynamic Cluster Membership

REXEC uses a dynamic cluster membership service that avoids use of static configuration files. In our experience with large clusters of computers, we have found that over time the set of nodes in the cluster that are available to perform useful work varies for a number of reasons. Adding nodes, removing nodes, bringing machines down, rebooting machines, and so forth are fairly common events in our system. Thus, rather than use static configuration files to define the set of nodes in the cluster we use a dynamic cluster membership service that discovers nodes as they join and leave the cluster.

This cluster membership service is built around a well-known cluster multicast address. By using a well-known multicast address, processes on different nodes can communicate with interested receivers without explicitly naming them. Senders who wish to send some information simply send it on the multicast address with a unique message type. Interested parties can elect to receive and interpret information of interest by examining incoming message types. If necessary, processes can even use the multicast channel to bootstrap point-to-point connections (e.g., by sending a multicast request for a server's point-to-point address). The Information Bus [65], an architecture for publish/subscribe services, uses a similar but more general model.

To implement the cluster membership service, we first needed a way for nodes to be added to the cluster membership when they are available and to be deleted from the cluster membership when they are unavailable. To do this, `rexecd` daemons multicast

announcement packets which include the IP address of the node (a TCP port could be included as well but currently `rexecd` daemons listen on a well-known port) along with other configuration information (e.g., number of CPUs) and state information (e.g., number of jobs REXEC is controlling, load, etc.). These announcement packets are sent periodically (once every minute) and also whenever a significant change in state observed by `rexecd` occurs. Currently, the state changes we send announcements for are job arrivals and job departures.

We maintain approximate membership of the cluster by using the reception of an announcement packet as a sign that a node is available and the lack of reception over a small multiple of the periodic announcement interval as a sign that a node is unavailable. The membership is approximate because information is necessarily delayed and because we use unreliable UDP/IP multicast as our transport protocol. Using the announcement packets, replicated `vexecd` daemons discover and maintain the node membership in the cluster by caching and timing out node announcement information. Unlike V [84, 87], which used a similar multicast-based protocol for transmitting load information, we use separate daemons to maintain the membership and load information as opposed to having every node in the cluster do so. Our primary reason for doing this is to avoid context switches on cluster nodes that are running user applications. If each `rexecd` received all multicast packets, then `rexecd` would need to be scheduled to handle the reception of these packets.

Failure detection based on timeouts can be slow unless the processes maintaining membership views are actively trying to reach all nodes in their view. `vexecd` daemons maintain views on cluster membership and provide services based on this information to clients. However, they do not actively make connections to the nodes whose addresses they are providing. Because of this, `vexecd`s may continue to send addresses of failed nodes to future `rexec` clients even when previous `rexec` clients have already detected that those nodes have failed. The end result is that `rexec` clients may fail repeatedly while trying to contact failed nodes until the `vexecd`s time those nodes out. To avoid this, REXEC uses an optimization where whenever an `rexec` client detects that a node is down, it sends a hint on the cluster multicast channel indicating such. Upon receiving a down hint, `vexecd`s will immediately remove that node from their membership. Thus, time-out delays are eliminated.

Decoupled Discovery and Selection

REXEC decouples node discovery from the selection of which nodes an application should run on. As discussed in the previous section, replicated `vexecd` daemons are responsible for discovering and maintaining the node membership of the cluster. Within one state announcement period, a new `vexecd` discovers the entire instantaneous membership of the cluster. With multiple `vexecd` daemons keeping track of all the nodes, their configuration and state, a selection policy is simply a mapping that applies some criteria to that list of available hosts and returns a set of hosts.

Because users may have different criteria in how they want nodes to be selected for their applications, discovery and selection are decoupled. The `vexecd` daemons which do discovery can implement any number of selection policies. The idea with `vexecd`s implementing selection services is that we envision that most users will probably choose from a small set of policies in deciding where to run their applications. In a community composed largely of scientific computing users, for example, lowest CPU load may be the most common criteria.

`vexecd`s in our system precompute and cache orderings on the list of available nodes so clients can quickly obtain the results of common selection policies. Under most circumstances, users will contact prewritten `vexecd` daemons asking for the n "best" nodes, where best is defined according to some selection criteria. The `vexecd`s simply return the top n nodes on their ordered list, which is recomputed each time a state change occurs with adjustments. `vexecd`s are free to implement policies with more complexity as well (e.g., return n nodes with lowest average CPU load over last three days). Users are also free to implement their own selection policies (e.g., by writing their own `vexecd`). As anyone is able to receive CPU state announcement messages from `rexecd`s, such an implementation would simply required listening on the cluster multicast address, receiving announcement messages, and applying a policy.

Since users will want to use `vexecd`s based on the selection policies the `vexecd`s implement, the discovery of `vexecd` daemons and use of their services cannot be completely transparent. More information is needed from the user either in the form of a list of suitable `vexecd` servers or a criteria which is expressed in a way that the system can automatically discover which `vexecd` daemons implement that criteria. Currently, we take the former approach. Users specify a list of suitable `vexecd` daemons using an environment variable (`VEXEC_SVRS`, a space-delimited list of hostnames where `vexecd` daemons run). The `rexec`

client, upon seeing a non-null `VEXEC_SVRS`, will try each daemon in turn until it succeeds or exhausts the list.

Discovery of the hostnames of the machines that implement suitable `vexecd` daemons can be done through out-of-band means (e.g., a system administrator uses a web page to post a list of hostnames that are designated to run `vexecd`s with certain policies) or it can be done semi-automatically. We offer both approaches. The former is self-explanatory. The latter involves using the cluster IP multicast channel to multicast to all `vexecd` daemons in the system asking them what selection policy they implement. Each `vexecd`, upon receiving a such a request, returns a string that provides a textual description of its policy which the user can then use to construct a suitable list for setting the `VEXEC_SVRS` environment variable.

Error and Cleanup Model

REXEC provides a well-defined error handling and cleanup model for applications. If an error occurs on the `rexec` client, in any of the remote processes, or on any of the TCP connections between the `rexec` client and any of the remote `rexecd` daemons, the entire application exits, all resources are reclaimed, and a textual error message is printed to the user. A common shortcoming in many previous remote execution systems, especially those that support parallel execution, is lack of a precise error and cleanup model and insufficient implementations of remote cleanup. Just as point-to-point facilities such as `ssh` often fail to perform appropriate cleanup (e.g., local `C-c` on the client does not always kill the remote process), many user-level remote execution environments also fail to properly relinquish resources on remote nodes. REXEC addresses this by defining a model, addressing the new failure modes associated with remote execution and parallel and distributed programs, and providing a robust implementation.

Transparent remote execution on a cluster and support for parallel and distributed applications introduces new failure modes in the execution of user applications. With remote process control being mediated through use of a local proxy, the `rexec` client, there are two obvious new modes of failure. First, the `rexec` client can fail and second the network connection between the `rexec` client and a remote `rexecd` can fail. Because the `rexec` client serves as the point of control for the user and logically represents the remote process(es) itself, we interpret failure of the `rexec` client as failure of the application and thus we cleanup. When the network connection from `rexec` to the `rexecd` fails, this prevents the

user from controlling the remote process and thus we view this as a failure as well.

Support for parallel and distributed programs introduces yet another failure mode: individual failure of processes in the program. Here, we take the position that if any process in the program fails, the entire application is aborted. The rationale behind this is that for all but the most trivially parallelizable and distributed programs, there will be at least some communication between processes and thus failure of one process usually means failure of the entire application. Of course, this will not be the case for every parallel or distributed program but we feel it is a reasonable model to present to users for a large class of programs.

In general, there are many potential error and cleanup models the system could support. However, only a handful of them make practical sense for real applications. For example, another useful failure model which we considered supporting but currently do not implement is the model where all processes are completely decoupled and we leave it up to the application to deal with failures. Such a model might be appropriate, for example, for a parallel application with its own error detection and the ability to grow and shrink based on resource availability and faults.

The implementation of the error and cleanup model is done mainly at user-level but also involves some small kernel modifications. At user-level, we need to monitor remote process failures locally on remote nodes and we need to monitor network failures. To determine whether a process failed, we have a thread in `rexecd` per remote process on that `rexecd` which captures the return code returned from the process. Examining the return code, this thread then performs a check to see if the process exited normally or abnormally. If it exited with an abnormal exit code, we interpret this as an error.

To determine whether a network connection has failed or not, we rely on system call error codes returned from network related system calls. In cases where processes fail but the machine's operating system is still functioning properly, network errors are detected quickly due to the OS's implementation of the TCP protocol. However, to ensure timely detection of network failures in cases where data is not sent for long periods of time (e.g., compute bound jobs) and a machine crashes, we use a heartbeat thread on the remote node. Since this heartbeat sends periodic heartbeats from `rexecd` to `rexec`, if the network connection has failed, these send failures will be detected in a timely manner [3]. Note that by interpreting network failures as failures which cause cleanup, we also cover the case

[3]TCP keepalive packets were not an option due to excessively high timeout values on the order of hours and the inability to set timeouts on a per-connection basis.

where the `rexec` client is sent a `SIGKILL` signal which it cannot catch due to our user-level implementation. In this case, `rexec` exits, its TCP connections are closed, and this causes remote `rexecd` daemons to perform cleanup.

REXEC uses kernel modifications to ensure that remote resources are reclaimed when performing remote cleanup. Specifically, we wanted to address cleanup issues for remote processes, including those that fork arbitrary process trees and have multiple threads. In these cases, keeping track of the process ID of the root of the process tree and sending a `SIGKILL` signal to that process is not enough to perform cleanup. Other descendents of that process may still be computing. Thus, we added a new system call to specify the root of a process tree (and that all descendents of that process should inherit the fact that they are part of the same tree) and a system call to deliver signals to all members of that tree, regardless of changes in process group, intermediate parents exiting causing their children to be inherited by `init`, and so forth. When performing cleanup, REXEC simply sends a `SIGKILL` to all processes in the logical tree, which results in all resources for all threads/processes in the process tree to be freed up. (Note also that we modified the `wait` system call to deal with process trees so a remote thread waiting for the remote process to exit does not return until all processes in the logical tree have exited.)

Parallel and Distributed Applications

REXEC supports parallel and distributed applications by allowing users to launch and control multiple instances of the same program on multiple nodes and by providing a set of hooks that allow parallel runtime environments to be built. Starting #nodes instances of the same program is accomplished by adding a -n #nodes switch to the `rexec` client program which allows the user to specify a program should be run on #nodes nodes of the cluster.

The hooks we provide for runtime environments are a fairly minimal set. Each remote process has four environment variables set by REXEC: `REXEC_GPID`, `REXEC_PAR_DEGREE`, `REXEC_MY_VNN`, and `REXEC_SVRS`. `REXEC_GPID` is a globally unique identifier for a particular execution of a user's application. It is implemented as a 64-bit concatenation of the 32-bit IP address of the interface the `rexec` client uses to communicate with `rexecd`s and the local 32-bit process ID of the `rexec` client. `REXEC_PAR_DEGREE` is a 32-bit integer which specifies the number of nodes the application is running on. Within an n node program, REXEC assigns an ordering on the nodes from 0 to $n-1$. On each node, `REXEC_MY_VNN`

specifies the position of that node in that ordering. Finally, `REXEC_SVRS` contains a list of `REXEC_PAR_DEGREE` hostnames (or IP addresses) for each of the nodes the user's application is running on.

Using these four environment variables, a parallel runtime environment, for example, would have enough information to identify which nodes should work on various parts of a dataset and to use whatever means employed to discover network addresses of peer endpoints. An example of how REXEC's parallel and distributed application support is used in practice is given in Section 4.1.4, where we discuss an REXEC-based MPI runtime.

Authentication And Encryption

REXEC provides user authentication and encryption of all communication between `rexec` clients and `rexecd` daemons. More specifically, REXEC uses the SSLeay version 0.9.0b implementation of the Secure Socket Layer (SSL) protocol [35] for authentication and encryption of all TCP connections between these entities. Each user has a private key, encrypted with 3DES, and a certificate containing the user's identity and a public key that is signed by a well-known certificate authority who verifies user identities. In our system, we use a single trusted certificate authority for certificate signing and use usernames as identifies in certificates.

Each time a user wants to run an application using REXEC, the user invokes the `rexec` client on the command line and types in a passphrase which decrypts the user's private key. The system then performs a handshake between the `rexec` client and `rexecd`, negotiates a cipher, uses a Diffie-Hellman key exchange to establish a session key, uses RSA to verify that the user's certificate was signed by the trusted certificate authority, and checks that the username in the certificate exists and that it matches that of corresponding local user ID that was propagated from the `rexec` client. Once the user's identity has been established, all communication over the corresponding TCP connection is encrypted with 3DES using the shared session key.

4.1.4 REXEC Applications

In this section, we present two examples of how REXEC has been applied to provide remote execution facilities to users. In the first example, we describe how the REXEC system is used in its basic form to provide remote execution for parallel and sequential

jobs. In the second example, we describe an MPI implementation using a fast communication layer on Myrinet that uses REXEC as its underlying remote execution facility. Using REXEC as the remote execution layer in this case is a substantial improvement over the default configuration which uses multiple `rsh/ssh` connections.

Parallel and Sequential Jobs

The `rexec` client provides the minimal amount of support needed to transparently run and control parallel and sequential programs on a cluster. Users run the `rexec` client as follows: `rexec -n #nodes progname arg1 arg2 .. argn`, where `#nodes` is the number of nodes the program should be executed on and `progname arg1 arg2 .. argn` is the command line the user would type to run program `progname` with arguments `arg1, arg2, .., argn` on a single node. Node selection is done through use of `vexecd` daemons by specifying a list of suitable `vexecd` daemons through the `VEXEC_SVRS` environment variable. Alternatively, if the user wants to run an application on a specific set of nodes, the user can set the `REXEC_SVRS` environment variable. A non-null `REXEC_SVRS` always takes precedence over `VEXEC_SVRS`. Parallel and distributed programs can be launched using the basic `rexec` client. It is the responsibility of runtime layers or the application to make use of REXEC's environment variable support for parallel and distributed programs to decide how data and computation should be partitioned and how communication between processes is established.

MPI/GM on Myrinet

Using REXEC's basic hooks for parallel and distributed programs, we have modified Myricom's MPI implementation over the GM (MPI/GM) fast communication layer [61] to use REXEC as its underlying remote execution mechanism. The original version of MPI/GM was configured by default to use `rsh` with static node/port number configuration files for remote execution to each node. With our modifications, users are able to run MPI programs on the cluster with GM while obtaining all the benefits of the REXEC system. Running an MPI program on multiple nodes is no different from running a normal application using REXEC. Two practical examples of problems we solve are elimination of static host configuration files and elimination of GM ports conflicts due to straggler processes, a problem which was immediately evident once we tried to use `C-c` to kill an MPI/GM program.

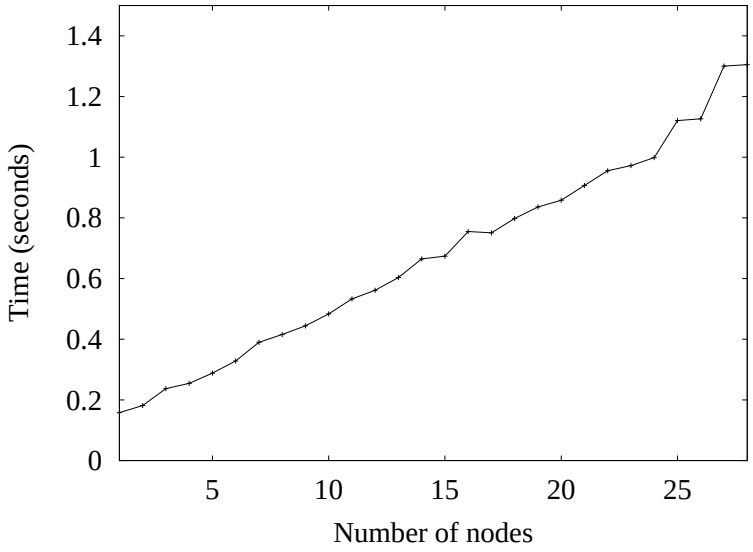

Figure 4.3: **Measured Execution Time to Run a Null Parallel Program with REXEC as a Function of Number of Nodes.** This graph shows the measured execution time of a parallel program that starts and immediately exits on multiple nodes of the cluster. The measurements illustrate the basic costs associated with running jobs through the REXEC system. The start-up and cleanup cost for a running a single node program with REXEC is 158 ms. As the number of nodes increases, total execution time scales linearly with an average per-node cost of 42.8 ms. (Note that we have not aggressively pursued optimizations so absolute performance numbers can still be improved.)

Modifying the MPI ADI layer to use REXEC required mapping REXEC environment variables to MPI variables and addressing port (communication endpoint) naming issues. MPI, like other parallel runtime environments, provides a set of library functions that allow each process in a program to obtain the total number of processes in the program and that process's unique ID according to some naming of those processes. These variables are most often used to decide how data or computation should be partitioned amongst multiple processes in a parallel program. For MPI/GM, these variables are the MPI size and rank and they are easily set for each process by simply setting them to the values of the `REXEC_PAR_DEGREE` and `REXEC_MY_VNN` environment variables.

In MPI/GM, each process uses a single port to communicate with all other processes in the program. Each node, or more specifically each Myrinet LANai4 network interface card, can support up to seven user ports ranging from 0 to 6. A port address is a combination of a hostname and an integer port number. The default configuration for MPI/GM relies on each user having a configuration file which specifies which port number is used by each process in the user's MPI program. With multiple programs running on the cluster, though, this manual configuration forces the user to deal with port name conflicts due to other programs competing for and using the same GM resources.

We solved the naming problem by first extending the GM library to support a function that binds to any free GM port and returns that port number to the application. This allows us to avoid having to explicitly specify port numbers. However, doing this now meant that each process in unaware of which port numbers its peer processes are bound to. Thus, the next thing we did was to allow peer processes to exchange names through bind and lookup operations on `enameserver`, a centralized name server used in the Berkeley NOW project. To do this, we took advantage of the two remaining environment variables, `REXEC_GPID` and `REXEC_SVRS`. The nameserver allows processes to bind (key,value) pairs and do to value lookups based on keys. We use the concatenation of `REXEC_GPID` and a process's rank separated by a ':' for keys. Because `REXEC_GPID` is unique throughout the system, this ensures that a n node program uses n unique keys. For values, we use the GM port numbers returned from the port creation function we added. Upon binding a (key,value) pair for its port, each process does a lookup on port numbers for each of its peers. The results of these lookups are then saved and used later for communication.

4.1.5 Related Work

Research efforts in remote execution environments for clusters have been going on for over a decade. Each effort has succeeded in addressing and to various extents solving different subsets of the key problems in remote execution systems. None of these systems, however, has addressed the range of problems that REXEC does. Built on previous work and practical experience with a large-scale research cluster, REXEC addresses a wide range of practical needs while providing useful features which address important issues such as error handling and cleanup, high availability, and dynamic cluster configuration. To accomplish its goals, REXEC is implemented at user-level on a commodity operating system with small modifications to the OS kernel. Such an approach is an example of one of three distinct implementation strategies: (i) user-level approaches (ii) modification of existing operating systems, and (iii) completely new distributed operating systems.

GLUnix [37], SCore-D [46], Sidle [50], Butler [62], HetNOS [8], and Load Sharing Facility (LSF) [103] are examples of user-level implementations. Compared to REXEC, each of these systems implements a subset of REXEC's features. GLUnix and Score-D, for example, are the only two systems in the list that support parallel programs. However, both of them also rely on centralized control and manually updated cluster configuration. Butler and LSF support different forms of replicated discovery and selection. However, neither supports an error and cleanup model as extensive as REXEC or strong authentication and encryption. One notable feature that has been implemented in some of these systems which REXEC currently does not support is a programmatic interface to the system. GLUnix, Butler, HetNOS, and LSF, for example, allow users to write applications which link with a C library of remote execution related functions. Using this model, applications such as a shell which automatically decides whether to execute a job locally or remotely have been developed.

MOSIX [7], NEST [1], COCANET Unix [77], and Solaris MC [52, 81] are examples of modifying and extending an existing operating system. Again, each of the systems supports only a subset of REXEC's features. NEST, for example, supports transparent remote execution but does not support features such as dynamic cluster membership or parallel and distributed application support. MOSIX, for example, provides transparent remote execution but does so in a fairly limited context which is mainly targeted for load balancing amongst a set of desktop machines to exploit idle time. One notable feature supported by MOSIX which REXEC currently does not support is process migration. Mechanisms to

implement it, however, are well-known [26, 53, 70, 87] in both user-level and kernel-level implementations and under various constraints. Another notable difference between REXEC and these kernel-level implementations is the degree of transparency in the remote execution system. Kernel-level implementations can achieve greater levels of transparency than user-level approaches. Solaris MC, for example, implements a true single system image with real global PIDs, a global `/proc` filesystem, and a global cluster-wide filesystem.

Sprite [67], V [84, 87], and LOCUS [97] are examples of completely new distributed operating systems which support transparent remote execution. Like the other systems described, these distributed operating systems also support only a subset of REXEC's features. V, for example, supports a publish-based, decentralized state announcement scheme very much like REXEC. On the hand, V does not support parallel applications, does not support flexible selection policies, nor does it implement strong authentication and encryption. Like MOSIX, all three of these systems support process migration which REXEC currently does not implement. In addition, like the kernel-level implementations previously described, these new operating systems also achieve greater levels of transparency due to implementations at the operating system level and, in the case of Sprite and LOCUS, cluster-wide global filesystems.

4.2 Time-shared Computational Economy

The first prototype we implemented is a computational economy for proportional-share CPU time [14]. Built on the REXEC system, this prototype lets users assign CPU valuations to their jobs which influence time-shared CPU allocations in an intuitive way. Valuations are expressed as rates which specify the maximum amount the user is willing to pay per minute of wall-clock time. Nodes keep track of rates for jobs competing for their resources. Based on individual rates and the sum of all competing rates, nodes assign proportional-share resource allocations with larger rates translating into larger fractions of the CPU. The end result is that important jobs finish faster, and consequently the system delivers higher performance since finishing important jobs earlier results in larger contributions to overall value. In this section, we provide details on the implementation of this system in the context of the system architecture presented in Chapter 3.

4.2.1 End Users and Applications

End users and applications in this system are drawn from two main sources: users affiliated with the UC Berkeley Millennium Project [89] and users/students in the UC Berkeley computer science (CS) department. In both cases, workloads are comprised mainly of long-running, compute-intensive applications. For Millennium Project users, such workloads are consistent with computer simulation and modeling, the driving force behind project group activity in Millennium. For CS department users, such workloads are only partially consistent. Workloads from CS department users are more diverse. However, the major resource consumers are still compute-intensive applications, such as network and systems simulations. Thus, in our efforts, we focus solely on market-based CPU allocation since that presents the largest opportunity for improvements in total value delivered.

4.2.2 Access Module

Users access the time-shared computational economy using modified versions of the REXEC client programs, `rexec` and `millstat`. Only three modifications were required. First, we modified `rexec` to take a new command line switch (`-r maxrate`) which specifies the maximum amount the user is willing to pay per wall-clock minute. [4] Second, we modified `rexec` to use a minimum node rate selection policy for selection since users will generally want to run on the nodes that have the least demand for their processors. Finally, we modified `millstat` to report rate information for all nodes so users can view the state of all the sellers in the economy before submitting their jobs.

Given these modifications to the client programs and the base functionality provided by the REXEC system, most of the requirements for the time-shared access module were already met. The few remaining ones required little or no modifications to REXEC. Seller discovery and selection of the lowest price seller were straightforward to implement by extending the `vexecd` daemons to maintain node rate information and provide a selection policy based on rates. The protocol to interact with sellers to obtain resource allocations was equally straightforward to implement by simply augmenting existing packets used by

[4] Rates are sent to node economic allocators just once as an optimization. Logically, jobs continuously purchase many fine-grain goods over time during their execution. Each of these goods encapsulates shares of CPU time with service guarantees that hold only over a small interval of time (e.g., multiple scheduling quantums) since node rates may change. Sending the rate once is simply a way of avoiding the extra communication needed to express (i) the repeated intent of purchasing the same good again, (ii) the new intent to purchase and repeatedly purchase the best possible good given a change in node rate.

REXEC with rate information. Finally, since we are using REXEC directly for remote execution, no additional remote execution protocol was required.

4.2.3 Economic Allocator

Leveraging the REXEC system, we use modified `rexecd` daemons as the economic allocators in the time-shared computational economy (Figure 4.4). In the new system, each `rexecd` now maintains a node rate $R = \sum_{i=1}^{n} r_i$ (r_i's are rates for jobs on that node) that reflects the aggregate demand for CPU time on that `rexecd`'s node. Each `rexecd` computes CPU share allocations, r_i/R, based on individual job rates and its own node rate. CPU share allocations are mapped onto ticket assignments using a system call interface to a CPU stride scheduler which enforces these allocations. Finally, to provide users with feedback, `rexecd`'s send multicast announcement packets which now also include the current node rate.

Feedback is implemented through node rate announcements but is made meaningful through rate-based charging. Without some kind of charging, users would observe the feedback signal but would have no incentives to respond to it. By implementing charging, observing and responding to feedback is important since future allocations and charges are directly related to the current state of the system (i.e., market conditions). In our system, charging mirrors rates so a job with rate r_i is charged r_i credits per wall-clock minute (Figure 4.5). The only exception to this rule is when there is no competition, in which case no charges are incurred. Given our charging algorithm, users are now faced with price incentives when deciding whether to submit a job or not. Specifically, high rates discourage submission of new jobs since those jobs would likely get only a small fraction of the CPU while still being charged their full rates. [5]

To implement charging, `rexecd`'s rely on a centralized bank service that uses SSL authentication for all bank transactions. Each user in the system is given an account in the bank. The bank uses a Berkeley DB database to keep track of these accounts and can perform authenticated credit/debit transactions against them. Only processes that can au-

[5] Locking users into a single rate once their job is submitted also influences user behavior. When node rates are high, an impatient user might consider assigning a comparably high rate so his/her job can immediately get assigned a large fraction of the CPU. However, if the rate drops shortly thereafter, that job will be locked into that rate for the rest of its execution, even if the total rate of all competing jobs is small. A better decision may have been to wait until the node rates decreased so a smaller rate could be assigned which would result in a comparable CPU allocation. If rates are allowed to change over time, this scenario does not occur.

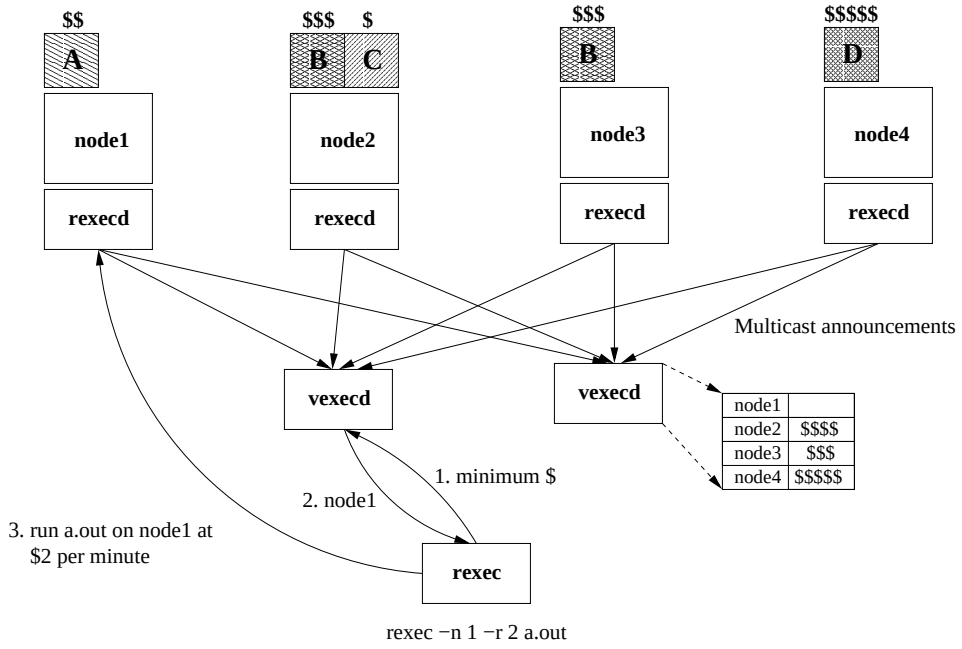

Figure 4.4: **Time-shared Computational Economy.** The organization of the time-shared computational economy essentially mirrors that of the base REXEC remote execution system. The key difference with the economy is its use of rates: job rates which specify a job's maximum willingness to pay per minute and node rates which reflects the aggregate demand for CPU time for the different nodes. In this system, users express CPU valuations as rates using a new command line switch in the `rexec` client. Each node in the system has a node rate $R = \sum_{i=1}^{n} r_i$ which is equal to the sum of all rates for jobs running on that node. `rexecd` daemons compute and enforce proportional-share CPU allocations for each job as r_i/R. Thus users generally want to run on nodes with the lowest rates since this gives them the maximum share of the CPU. To provide users with this information, `rexecd` daemons multicast announcement packets which include their current node rate. `vexecd` daemons listen for these announcements and can then be queried to select the nodes with the lowest node rates when running a new job.

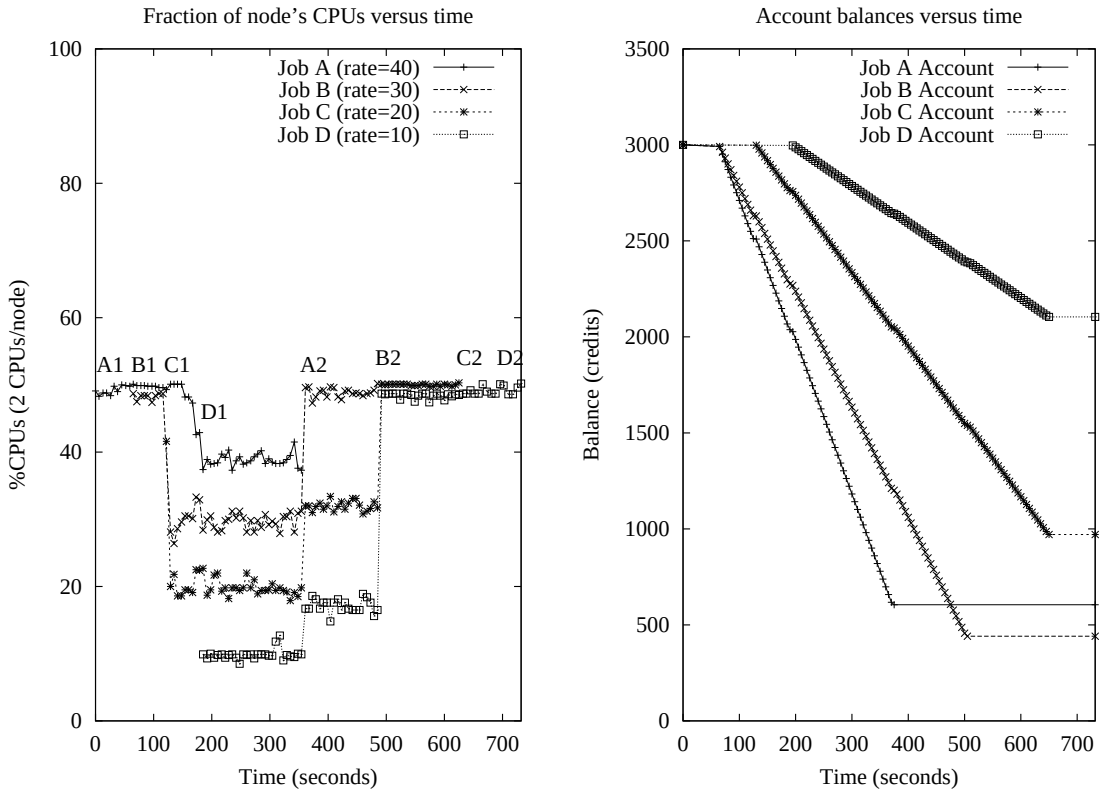

Figure 4.5: **Measured CPU allocation and Charging on a Single Node.** In this scenario, jobs A, B, C, and D are willing to pay rates of 40, 30, 20, and 10 respectively to compete for CPU resources on a single node. Each node has two CPUs. Job arrival times are labeled A1, B1, C1, and D1. Job departure departure times are labeled A2, B2, C2, and D2. At any instant in time (e.g., interval C1-B2), we see that the relative CPU allocation of any two jobs is given by the ratio of the rates they are willing to pay. (Note that since all the jobs are single threaded, when two jobs compete, each receives 50% of the node's CPUs, i.e., one CPU.) When multiple jobs compete, each job is charged based on the rate it is willing to pay and is given an allocation in proportion to its rate. When a single job demands the resources, it is given all of the resources and is charged nothing (e.g, intervals A1-B1, C2-D2).

thenticate as `root`, such as `rexecd`, are allowed to execute transactions on accounts. Strong authentication is used to prevent unauthorized access to user bank accounts which would undermine the effectiveness of the computational economy. `rexecd`'s use authenticated transactions to periodically debit accounts for all running jobs.

Since charging is not something traditional systems do, robustness and scalability in the charging mechanism are relevant issues which we must address. While we do not anticipate frequent failures of the bank service, especially given its simplicity, it is still possible that it can become unavailable and, in some cases, even needs to be restarted on a different node. We handle failures in the bank service using queueing and timeout/retry on bank transactions when the bank becomes unavailable. We handle changes in the bank's address by using a multicast-based discovery protocol to dynamically discover the address of the bank. Scalability and overhead are also important. We address this mainly by debiting accounts at long periodic intervals. By reducing the number of bank transactions, this reduces overhead per node and also avoids having the bank become a bottleneck.

4.2.4 Resource Managers

To enforce CPU resource allocations based on value, we modified the Linux 2.2.5 kernel by replacing the standard priority scheduler with an implementation of stride scheduling, a deterministic, proportional-share scheduling algorithm [96]. We did so primarily because stride schedulers provide much better precision when enforcing CPU allocations and such precision is important when enforcing the proportional-share allocations computed by the economic allocator in the computational economy. Building on the basic stride scheduling algorithm, we also added two extensions, one to support stride scheduling on SMPs and another to properly handle multithreaded and multiprocess jobs. Before covering those extensions, though, we first summarize the basic stride scheduling algorithm.

In stride scheduling, resource rights to a shared resource are represented by T tickets which are allocated across n jobs. Each job i has t_i tickets ($\sum_{i=1}^{n} t_i = T$) which represents its resource allocation; a stride s_i, the interval of virtual time (measured in CPU quantums) job i has to wait until being scheduled again; and a pass p_i, the virtual time when the job will next be scheduled. The algorithm works by picking the job with lowest pass, running it for a scheduling quantum, incrementing its pass by its stride, and iterating (see Figure 4.6). Strides are inversely proportional to tickets so jobs with more tickets have shorter strides, are scheduled more frequently, and obtain a greater fraction of the resource.

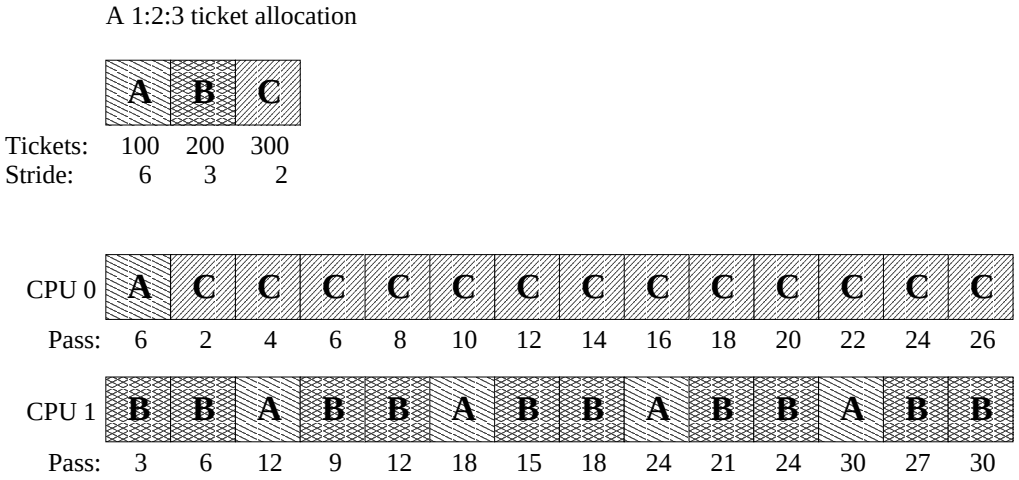

Figure 4.6: **Stride Scheduling Algorithm on SMPs.** In this example, jobs A, B, and C are competing for CPU time on a 2-way SMP. Resource rights to the SMP's 2 CPUs are encapsulated in a pool of 600 tickets. A, B, and C have 100, 200, and 300 tickets respectively, thus they should receive a 1:2:3 allocation of the machine CPUs over time. (Note that with a k-way SMP, having the rights to more than 1/k-th the tickets is really the same as having exactly 1/k unless the job is multithreaded.) A job's stride is the interval of virtual time, measured in CPU quantums, between scheduling times for that job. (It is inversely proportional to the ticket allocation.) A job's pass is the virtual time when the job will be scheduled. It is incremented by the job's stride each time the job is scheduled. For each CPU quantum on each processor, the stride scheduler chooses the job with the lowest pass that is not already scheduled on another processor, runs it for one scheduling quantum, and increments its pass by its stride.

This algorithm is deterministic and, within resolution of the scheduling quantum, schedules jobs in exact proportions based on their ticket assignments.

The first extension to the algorithm was support for SMPs. Since all of the nodes in our prototype are SMPs, extending the basic stride scheduling algorithm to support SMPs was an essential requirement. This extension involved making two modifications to the basic algorithm. First, when k processes are running on k CPUs, the global pass (see [96] for more details) must be incremented k times as fast to account for concurrency. Second, whenever a process has more than T/k tickets on a k CPU SMP, its ticket allocation must temporarily be truncated since a single threaded process cannot use more one CPU and should not be penalized for extra resources that it cannot use. (A more general scheme would also take into account the number of threads rather than assume a single thread.) Surplus-fair scheduling [11] is another proportional-share algorithm for SMPs that would have been equally applicable here.

The second extension to the algorithm was bounding and controlling resource usage within a collection of related entities (e.g., all the threads in a process, a collection of processes working together). This extension was necessary to prevent multiprocess and multithreaded applications from multiplying their resource usage by creating new processes or threads. First, we isolate all related entities using their own currencies (see [95]). Each job in the system is scheduled on the basis of the number of base tickets it has. To handle a multithreaded or multiprocess job i with base ticket allocation t_i, we simply ensure that the aggregate base ticket assignment for all entities forked or cloned from a root process, including the root itself is t_i. We do this by maintaining a linked list of currencies, each of which has an associated linked list of processes/threads that use that currency. The default policy for ticket allocation within a multithreaded or multiprocess job is to perform fair-share across all entities. For applications that desire more control, a system call interface is provided that allows applications to manage tickets in their own application-specific currency.

4.2.5 Resources

Resources in the time-shared computational economy are proportional-share CPU allocations on a subset of UC Berkeley Millennium cluster, a 300+ CPU cluster of symmetric multiprocessors. The specific subset of the cluster on which the computational economy was deployed consisted of 28 2-way Dell Poweredge 2300 multiprocessors, each running a

modified version of the Linux 2.2.5 operating system. Each node in this subset had two 500 MHz Intel Pentium III processors, 512 MB of memory, 17.4 GB of local storage, and had network connections to Myrinet [10], a switched, 1.28 Gb/s wormhole-routed network, and a switched 100 Mb/s Ethernet.

4.3 Anemone: a Market-based Batch Queue System

The second prototype we implemented is Anemone, a market-based batch queue system. Built on the infrastructure provided by REXEC, Anemone implements a market for dedicated CPU time based on a dynamic pool of batch nodes that grows and shrinks based on demand. To prioritize the scheduling of queued jobs, it uses a variant of the Vickrey auction based on rates which specify willingness to pay per node CPU hour. Like traditional batch queue systems, Anemone presents the familiar interface of client programs to submit, delete, and obtain status information on collections of batch jobs. From the user's perspective, the key difference is the specification of bids which specify valuations on dedicated blocks of CPU time. Using this extra information on CPU valuations, Anemone is capable of delivering increased value to users by focusing its resources on the jobs that deliver the most value. In this section, we provide implementation details on Anemone based on the system architecture presented in Chapter 3.

4.3.1 End Users and Applications

Anemone users tend to fall into one of two groups. The first group is users who have very large numbers of jobs to run and use the batch system out of convenience. For these users, the aggregate resource demand of their jobs usually exceeds cluster capacity. Thus, jobs necessarily must be submitted over time in order to avoid overloading the system. Using the batch system saves these users the work of monitoring cluster state and manually submitting their jobs over time. The second group of users have applications that require dedicated access and use the batch system for correctness. For these users, running their jobs on a time-shared system is infeasible since sharing would invalidate their timing results. Running through the batch queue is their only option. Workloads from the first group of users are almost always comprised of long-running, compute-intensive applications while workloads from the second group can be arbitrary (e.g., timing an I/O system). Since the Millennium cluster's primary use is computer simulation and modeling, the first class of

users easily dominates. Thus, the impact of a computational economy for CPU time should be significant.

4.3.2 Access Module

Users access the Anemone system using three client programs that present the same style of interface as traditional batch systems. In Anemone, these programs are: `eqsub` for submitting collections of batch jobs, `eqdel` for canceling collections of batch jobs, and `eqstat` for obtaining status information on collections of batch jobs. Using three programs for submission, canceling, and status information is consistent with the user interface to previous batch systems and reduces the learning curve for new users. Since the key difference with Anemone is its use of market-based scheduling, providing a familiar interface is important since it allows users to quickly ramp up on using the system and begin focusing their attention on the new features of Anemone, specifically bids on CPU time.

```
<?xml version="1.0"?>
<batch_job>
   <stdout_dir>/work/bnc/stdout</stdout_dir>
   <stderr_dir>/work/bnc/stderr</stderr_dir>
   <joblist>
       <job>
          <bid>10</bid>
          <time>0.5</time>
          <num_nodes>1</num_nodes>
          <command>/home/bnc/bin/job1</command>
       </job>
       <job>
          <bid>40</bid>
          <time>2</time>
          <num_nodes>2</num_nodes>
          <command>/home/bnc/bin/job2</command>
       </job>
   </joblist>
</batch_job>
```

Figure 4.7: **Anemone Batch File.** Anemone batch files are XML files that contain attributes associated with each batch job's execution and global attributes which specify where to save output written to `stdout` and `stderr`. Job attributes include the pathname to the program to run, the number of nodes to run the job on, the per-node block of CPU requested (in hours), and a bid. The bid, number of nodes, and CPU time requested are used to compute a rate which specifies the user's willingness to pay per node CPU hour. These rates are then used in a Vickrey auction to decide which job in the queue is scheduled next.

Users describe collections of batch jobs and submit them through `eqsub` using

XML files which specify attributes associated with each batch job's execution (Figure 4.7). For resource management, the three most important attributes are the bid, the number of nodes needed by the job, and the per-node block of dedicated CPU time requested. Together, these attributes are used to compute a rate which specifies the amount the user is willing to pay per node CPU hour. These rates are then used by the economic allocator, `eqd` ("economic queue daemon"), to determine the order in which queued jobs are scheduled on the cluster. Users should set their bids according to their true valuations for CPU time in order to maximize their utility.

In addition to the bid and CPU requirements, Anemone XML files also specify a minimal set of attributes needed to process batch jobs. This set includes the pathname to the executable to run for each job along with command line arguments as well as optional pathnames to directories where output from `stdout` and `stderr` can be saved. Since batch jobs are not associated with a terminal, specifying these pathnames can be useful for debugging or for saving program output. These three pathnames along with the bid and CPU requirements specify the complete set of attributes for each job. While this set is obviously not as extensive as feature-rich batch systems such as PBS, it is adequate for our purposes of supporting market-based scheduling of compute-intensive batch jobs.

All three programs in the access module layer assume one economic allocator which is accessible through a well-known IP address and TCP port and specified through an environment variable. Because of this, seller discovery in Anemone is not an issue as it was in the time-shared computational economy. Having a centralized `eqd` simplifies the implementation since this means `eqd` has global knowledge of batch node availability and the status of all batch jobs. Without additional mechanisms, this approach is clearly not fault tolerant. However, this is no worse than what current systems provide and thus is consistent with our goal of maintaining the status quo on fault tolerance. If fault tolerance is an issue, well-known techniques such as primary-backup could be applied to increase system availability. Scalability is also not a significant issue since batch systems do not have high processing overheads.

The access module client programs encapsulate all the details of interacting with the economic allocator and performing remote execution of batch jobs. From the users's perspective, using Anemone for batch processing compared to traditional batch systems involves only one additional step, namely specifying valuations on dedicated CPU time in the form of bids. The client programs implement the protocol to interact with the economic

allocator. Since processing is typically queued, this interaction is brief. Collections of batch jobs are submitted and added to the economic allocator's queue and, at some later point, are eventually scheduled and remotely executed on nodes in the economic allocator's pool of batch nodes.

In each of Anemone's access module client programs, strong authentication via SSL is used to verify user identities when processing requests on collections of batch jobs. As with the time-shared computational economy, this is important since users are charged for CPU usage and have finite bank accounts to purchase cluster resources. Anemone's authentication is slightly different than REXEC's since in the REXEC case, users interact directly with the `rexecd`'s. With Anemone, users authenticate themselves to `eqd`, which runs as `root`. When one of those users' jobs is eventually scheduled, `eqd` uses the results from the previous authentication step for charging and remote execution of the job on behalf of the user.

4.3.3 Economic Allocator

The economic allocator, `eqd`, is a daemon process which keeps track of all batch jobs in the queue and schedules them on batch nodes using a variant of the Vickrey auction (Figure 4.8). In Anemone, this auction is essentially a sealed-bid, second-price auction where bids are rates computed based on attributes from the user's XML file as a bid per CPU node hour. The sole difference between a pure Vickrey auction and our auction concerns head-of-line (HOL) blocking, i.e., when the most eligible job cannot be scheduled because, although some nodes are free, more nodes are still needed to run the job.

When HOL blocking occurs in the auction, free nodes may go idle until the remaining nodes become available unless perfect backfilling occurs. In general, however, backfilling is almost always imperfect and is not supported in the current version of Anemone anyway. Thus, when HOL blocking occurs, `eqd` simply waits for the remaining nodes to become available and charges the most eligible job for the CPU time wasted on the free nodes, which go unused during this time. The rationale behind these charges is that if the most eligible job was not at the head of the queue, those free nodes which are idle may actually be performing useful work.

Anemone creates the goods in its market for dedicated CPU time using the same raw materials as the time-shared computational economy described in the previous section. More concretely, Anemone manages a dynamic pool of batch nodes that comes from the

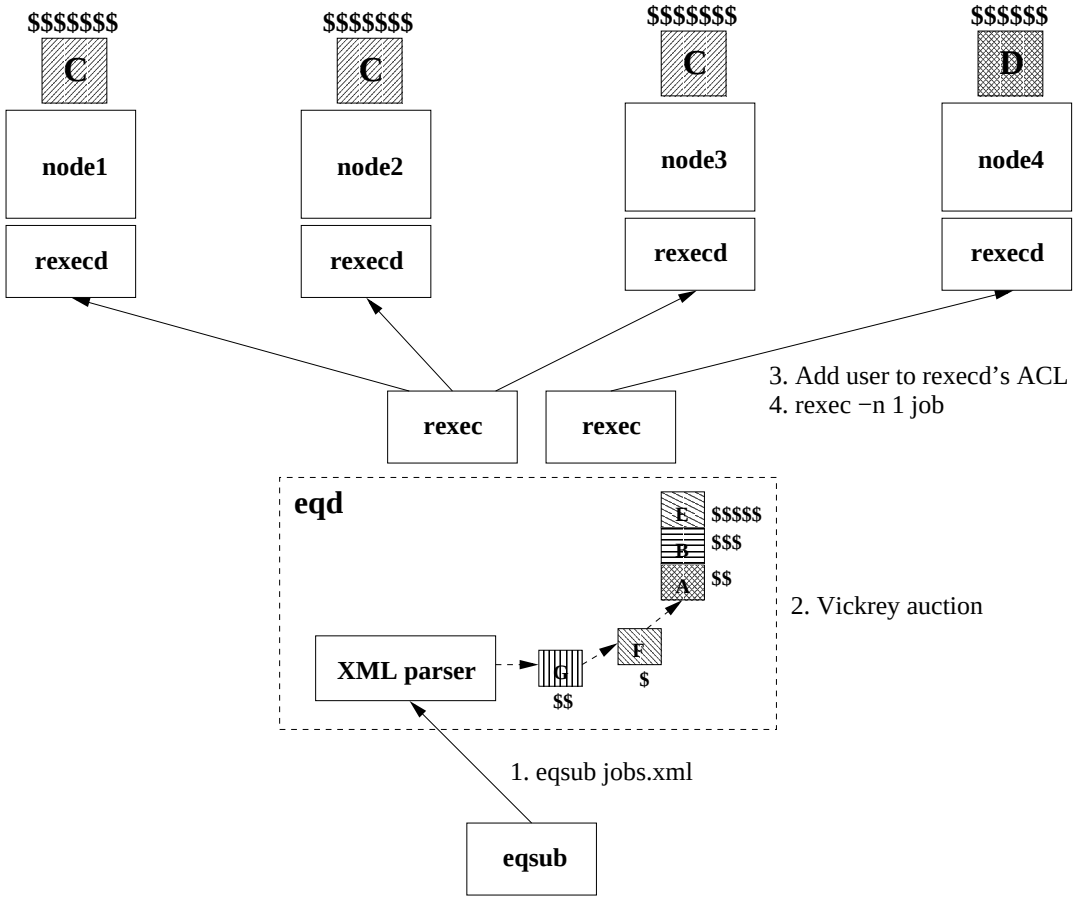

Figure 4.8: **The Anemone Market-based Batch Queue System.** The Anemone system is organized around a centralized economic allocator, `eqd`, which is responsible for running a Vickrey auction and scheduling the most eligible jobs onto batch nodes for execution. Users submit jobs by sending batch files to `eqd` using the client program `eqsub`. Upon receiving a new batch file, `eqd` parses it and extracts a list of jobs and attributes and inserts them into the batch queue. Whenever a scheduling opportunity arises (i.e., free nodes are available), `eqd` runs a Vickrey auction repeatedly and schedules as many winners as possible onto the free nodes. For each winning job, it adds the owner of the job to the appropriate node `rexecd` access control list (ACL) then uses the `rexec` client to run the job on behalf of the user. Using ACLs on batch node `rexecd`s is necessary to prevent users from bypassing the admission control in `eqd` and violating Anemone's performance guarantee of dedicated access.

same set of nodes that are accessible through the time-shared computational economy. At any given moment, each node in the cluster is in either: (i) the time-shared pool managed using the mechanisms described in the previous section or (ii) the batch pool managed by Anemone. The size of the two pools grows and shrinks depending on the demand for dedicated CPU time.

By dynamically managing the size of the time-shared and batch pools based on demand, we avoid statically partitioning the pools which causes resources to go idle in cases where there is excess demand in one pool and excess supply in another. Raw materials (nodes) to produce either time-shared or dedicated CPU goods are moved to the appropriate sellers to produce supply to address instantaneous demand. In the current implementation, growing and shrinking is initiated entirely by Anemone. That is, the default is for all nodes to belong to the time-shared pool. As demand for dedicated CPU time increases, Anemone attempts to steal nodes from the time-shared pool in order to increase its supply.

Growing and shrinking in the batch pool is done using distributed leases [42] and strong authentication based on excess demand for dedicated CPU time (Figure 4.9). By listening on the cluster IP multicast address, `eqd` maintains an approximate list of all time-shared nodes and their state, specifically whether they are idle or not. When demand for dedicated CPU time exceeds supply (i.e., when a queue builds), Anemone actively tries to steal idle nodes from the time-shared pool and add them to the batch pool by obtaining leases on those nodes. Nodes from the time-shared pool must be idle since otherwise they could not be used to produce goods that meet the service guarantees of dedicated access. When supply exceeds demand, batch nodes are returned back to the time-shared pool by canceling their leases.

Stealing and returning nodes to grow and shrink the batch pool is implemented using distributed leases with access control lists (ACLs) as an extension of the `rexecd` daemons. Distributed leases are basically resource reservations with an expiration time. We use leases as opposed to hard reservations to avoid cleanup problems associated with process failure, network partitions, and the myriad other possible failure modes of distributed systems that can lead to stale resource reservations. `rexecd` grants leases to any process that authenticates as `root` (e.g., `eqd`) and upon granting it, (i) announces its unavailability, (ii) stops announcing its availability in the time-shared pool, and (iii) begins enforcing the lease's ACL. Once leases on nodes have been obtained, Anemone adds and removes users from the ACLs associated with those leases as jobs start and end on those nodes.

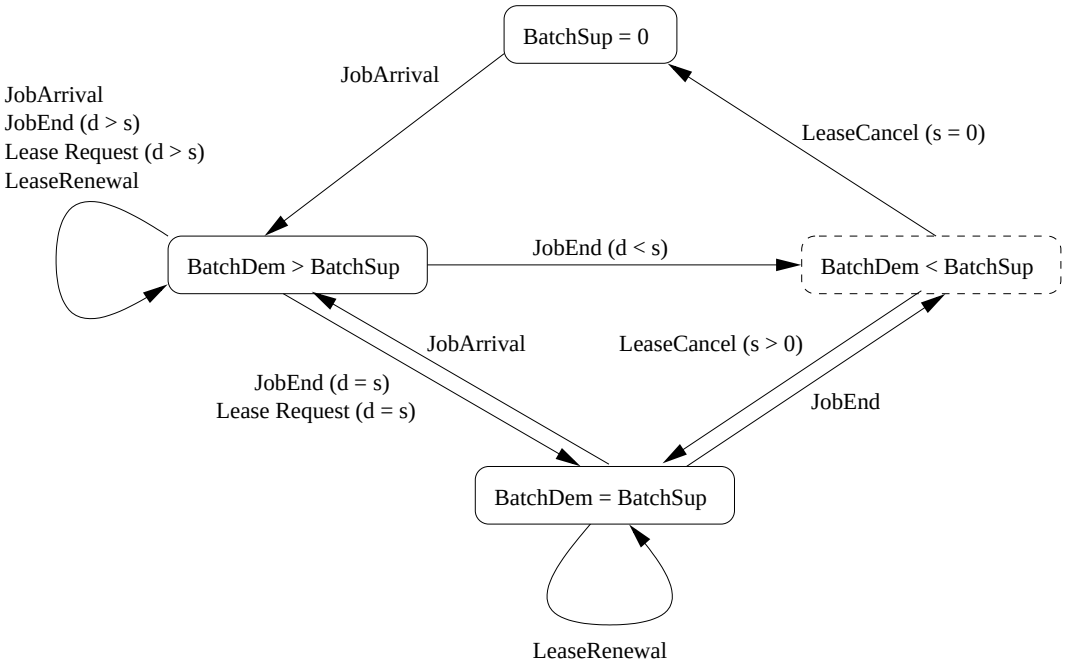

Figure 4.9: **State Diagram for Growing and Shrinking the Batch pool Based on Batch Supply and Demand, `BatchSup` and `BatchDem`.** Naturally, `JobArrival` events cause demand to increase while `JobEnd` events cause demand to decrease. In response to fluctuating demand, Anemone can steal (idle) nodes from the time-shared pool to increase supply or return nodes to the time-shared pool to decrease supply. Over time, Anemone attempts to steal just enough nodes so that `BatchDem` = `BatchSup`. Stealing and returning of nodes is done using distributed leases through `LeaseRequest` and `LeaseCancel` events. In the common case, Anemone usually is in either the `BatchDem` > `BatchSup` case state (i.e., the queue is non-empty) during periods of activity or the `BatchSup` = 0 case during periods of quiescence.

Dynamically adjusting the pool sizes addresses the practical problem of under-utilizing resources due to static partitioning. However, it is important to note that more work is still needed to implement a symmetric version of this mechanism. In the current implementation, the time-shared system cannot actively try and steal nodes back from the batch pool once they have been stolen. Furthermore, there is no mechanism, for either time-shared or batch usage, that allows either system to express an intent to steal so that no future work is placed on a node. (One possibility is for the time-shared pool to refuse lease renewals from Anemone at some point.) The design of the above mechanisms and the analysis of their interaction with simultaneous demand for both time-shared versus dedicated CPU goods is future work.

4.3.4 Resource Managers

There are no special resource managers in Anemone. To guarantee dedicated access, Anemone implements admission control to ensure that at most one user process is running on each batch node. Admission control is implemented in two places in the system. First, it is implemented in `eqd` by keeping track of whether each node is busy or free and avoiding scheduling more than one process per node. Since we assume a single `eqd` for all batch nodes in the cluster, this implementation is straightforward. Second, it is implemented in `rexecd` for nodes in the batch pool by having ACLs, managed by `eqd`, that are enforced whenever a node transitions from the time-shared pool into the batch pool. With ACLs on the batch nodes, users who attempt to bypass `eqd`'s admission control by explicitly naming batch nodes and using the `rexec` client to submit jobs will be denied access.

4.3.5 Resources

Resources in Anemone are dedicated blocks of CPU time used to run batch processes to completion. By definition, no sharing occurs with these dedicated CPU blocks. Because of this, one might be inclined to think that this deviates from our view in Chapter 3 of all computational resources being share-based allocations. However, this is not the case. Dedicated blocks of CPU time are just the special case of CPU shares that obtain the entire fraction of the processor. The key difference in Anemone is that these proportional-share allocations are guaranteed for the duration of the process using admission control. Without admission control, computational economies would be incapable of selling goods that

provide hard performance guarantees since arbitrary sharing could occur.

4.4 Experience

This section describes our experience with real users using the time-shared computational economy. Market-based systems provide benefits which necessarily depend on user participation and behavior in order to be effective. Mechanisms are provided that allow users to express valuations, but it is the users who must ultimately make effective use of these mechanisms. Similarly, while price feedback in theory can alter user behavior to avoid overcommittment of resources, whether this is true in practice can only be determined through experiments on a live system. In this section, we describe our experimental setup, our observations based on a two-month trace of user activity, and end with some discussion on the difficulty of performing meaningful empirical studies of market-based systems.

4.4.1 Experimental Setup

Our experimental setup consisted of two clusters, a priced cluster implementing the time-shared computational economy (Section 4.2) and an unpriced cluster which provided fair-share CPU allocations at no cost. Both systems consisted of 14 2-way SMPs for a total of 28 processors each. Users logged into one of two front-end machines, observed the state of both clusters, and had their choice of submitting jobs to either the priced cluster or the unpriced cluster through the REXEC remote execution system. CPU allocations on the priced cluster were proportional-share and based on the rate-based economic mechanism described earlier. In contrast, unpriced CPU allocations were fair-share regardless of how competing users valued the underlying resources. Traces recorded data on job executions (e.g., rates) and the state of the system over time (e.g., CPU load averages).

A key goal in performing this experiment was to observe how users would respond given the choice of a system managed through market-based resource management and a traditional system that lacks any machinery for users to influence their resource allocations. Naturally, our expectations were that users, while curious, would eventually revert to their usual behavior of placing jobs on the resources which are the least loaded. As contention for resource increases, we had two hypotheses for what would happen. First, as competition increases, the value of being able to express valuations increases since fair-share allocations become smaller and so users with important jobs would tend to move to the priced cluster.

Second, as priced competition became very high, we expected that users would eventually move all of their extra work onto the unpriced cluster since load does not influence cost since everything is free.

In addition to high-level observations, another goal was to collect significant amounts of data on user behavior through traces based on user job submissions and cluster state over time. We instrumented the REXEC remote execution system to collect this data, including job submission information (e.g., rates, job execution times, etc.) and the state of all cluster nodes, including statistics such as CPU load averages over time. One of our main goals in this data collection was to get an idea of how users actually value cluster resources based on their application needs and to see whether or not the expressive power of market-based systems is actually used in practice. In addition, we also planned to use the data to drive trace-based simulations of market-based systems to evaluate different market-based mechanisms and their parameterizations based on usage on a real system.

4.4.2 Observations

Perhaps the most meaningful result obtained in this data collection experiment was information on how users value cluster resources. Over a two-month window of the system's operation, we recorded rate information for all jobs submitted to the priced cluster. 3606 jobs were submitted during this time. A histogram showing the rates users assigned to their jobs is shown in Figure 4.10. We observe that a substantial number of these jobs had very low valuations assigned to them. For example, we observe that 34% of all jobs had rates less than 0.25. On the other hand, we observe spikes where valuations are substantially higher. For example, we see significant spikes at rates of 1, 3, and 5. The spike when job rate is equal to 5 corresponds to 4% (150 jobs) of the entire workload. For each of these jobs, value is at least 20 times higher than any job in the lowest 34% of the workload.

This data provides some early evidence on a real system which shows that knowing how much competing users value the resources is important for cluster resource management. As mentioned, we have cases where the ratio of valuations for competing jobs is 20:1 and sometimes even higher. In these cases, not knowing valuations and providing fair-share resource allocations is highly suboptimal in terms of optimization for user utility. By allowing users to express resource valuations, the priced cluster allows users to influence their resource allocations based on their utility functions. High value jobs are assigned higher rates rates and consequently receiver a larger fraction of the CPU over time. Conversely,

low value jobs are assigned lower rates and receiver smaller fractions of the CPU, but are charged at a lower rate.

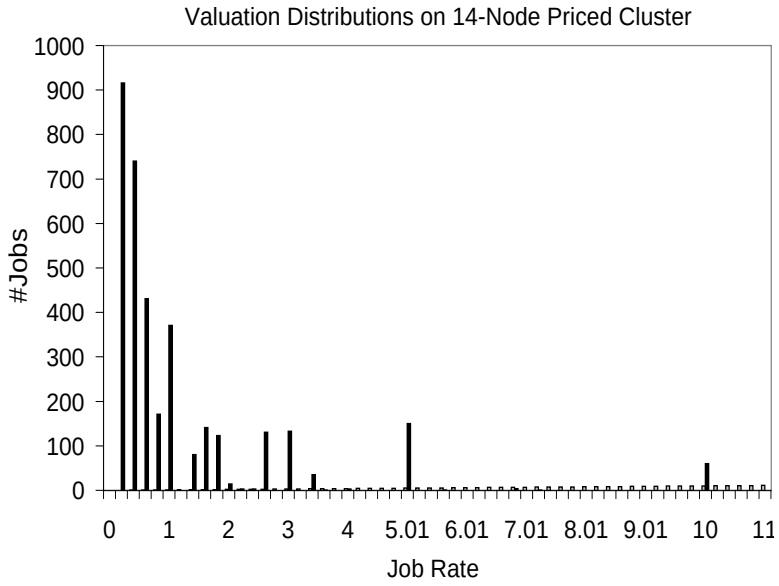

Figure 4.10: **Valuation Distributions for the Priced Cluster.** This figure shows a histogram of the rates users assigned to their jobs on a 28-processor cluster of 2-way SMPs over a two-month period in 1999. 3604 jobs were submitted to the priced cluster.

Despite this success, one of the key difficulties we had in performing these experiments was creating a meaningful workload. As computer simulation and modeling becomes increasingly important for research in computational science and engineering, high-performance computing on clusters will become a standard activity for scientists and engineers. However, we are not quite there yet, and because of this, obtaining a meaningful user workload that could drive the system to saturation was difficult, especially during the early stages of the project when this experiment was performed. Figure 4.11 shows a histogram of load on both the priced and unpriced clusters over the duration of the entire experiment. As we can see, load in almost all cases was either very low (i.e., the system was idle, background noise) or system load was slightly above 2. This shows that there was usually little competition for processor time.

As more software has become available and users have become more knowledgeable about how to the system, workloads have increased. As of December 2001, we commonly observe periods of serious contention where CPU loads have shot up to more than 20 on

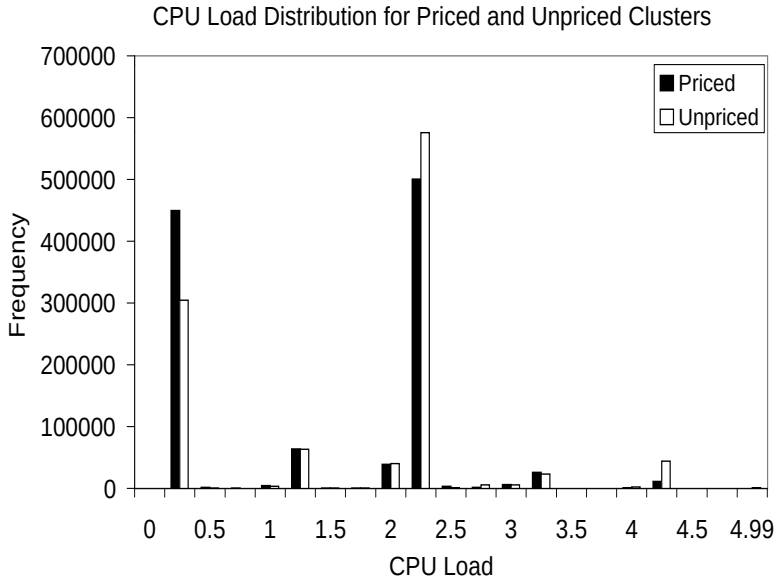

Figure 4.11: **CPU Load Distribution for Priced and Unpriced Clusters.** This figure shows a histogram of 1-minute CPU load averages for both the priced and unpriced clusters over a two-month period in 1999.

each node. The cluster also now contains over 300 processors, and thus we have seen a substantial increase in demand. Of course, at this point, the software used to obtain the data presented here has since been retired. One reason for this is maintainence costs. For example, maintaining the system software would have required consistently tweaking the stride-scheduler implementation to work with newer versions of the Linux kernel as system administrators tracked the latest kernels to fix various other system bugs. In addition, my own efforts had moved to simulation by this point and thus, there simply was not enough time to resurrect the code and redeploy it.

In summary, performing a meaningful market-based experiment of significant scale is a challenging task. Software that implements both the system and data collection needs to be implemented early and tested rigorously for both stability, correctness, and usability. As user confidence in the system grows and users become comfortable using the system, deployment of the market-based system should proceed in an incremental manner to avoid major shifts in what the user expects. Users should be provided with information on how to use the new system and what benefits it provides. Then, as more of the bugs are worked out, an eventual shift to a completely market-based system can occur. The key is to build

a consistent user community up as quickly as possible, then use incremental deployment in order to retain users as the experimental system is pushed out to all the nodes.

4.4.3 Summary

Given the lack of a sufficiently demanding workload, we were not able to answer all of the questions the study initially sought to answer. However, some meaningful data was still collected, in particular the histogram data which shows how users valued the resources. This result provides early evidence supporting the need for some of the properties of market-based systems (i.e., explicit resource valuations).

In order to answer the remaining questions we had, we turned to simulation. While it is definitely worthwhile to perform empirical studies, simulations are also valuable since they allow systematic exploration of performance under a wide variety of workloads. Towards this end, the next two chapters of this dissertation describe the simulator we built and the results from an extensive series of simulations.

Chapter 5

Stingray: a Market-based Simulator

In this chapter, we describe the design and implementation of Stingray, a simulator for market-based cluster resource management. We describe how Stingray enables a wide range of cluster resource management systems under various workloads to be studied. Of particular interest in this dissertation are simulations that investigate variations of the prototype implementations described in Chapter 4. Results of such simulations are important since implementations of the underlying base systems have already been shown to be feasible on real systems. We begin the discussion of Stingray by first presenting the system model that is assumed in all of the simulations. Next, we describe implementation details of how we simulate this model. We describe the simulation engine, the simulation modules, and how simulation runs are specified.

5.1 System Model

Stingray is based on a system model that essentially mirrors the system architecture described in Chapter 3. Again, we have systems consisting of five layers: users and applications, access modules (AMs), economic allocators (EAs), resource managers, and resources. System layers in Stingray are represented using four types of simulation modules: user modules for users and applications, AM modules for access modules, EA modules for economic allocators, and node modules for resources and their associated resource managers (Figure 5.1). Feedback between different modules is modeled using logical bulletin boards

which allow reading and writing of snapshots of module state information. Simulations consist of collections of modules and bulletin boards parameterized and interconnected in specific ways. In this section, we describe the main interactions between connected modules and present two examples of how the system model can be used to model real systems for simulation.

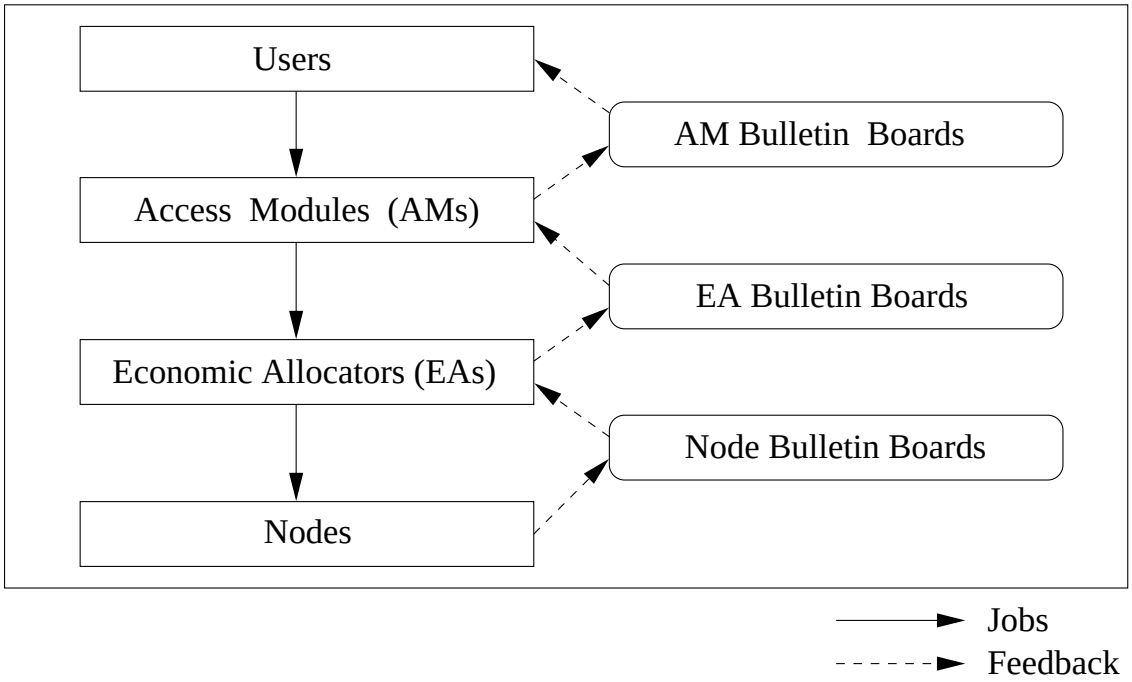

Figure 5.1: **Stingray System Model.** Users generate workloads which consist of sequences of jobs over time. Using feedback information from AM bulletin boards, users submit jobs through access modules based on configurable policies. Access modules, using feedback information from EA bulletin boards, route these jobs to the "best" economic allocators using configurable policies. Economic allocators, using local information, feedback information from Node bulletin boards, and specific economic mechanisms submit jobs for execution on nodes that they manage. Nodes, based on resource supply and demand (i.e., node resources and competing jobs), finally execute those jobs to completion.

5.1.1 Module Interactions

Modules interact with each other primarily through two types of events flowing over module-to-module interconnections: job events and feedback events. Job events model the flow of jobs from the users who generate them down to the nodes that execute them. This flow of events follows from the discussion in Chapter 3. Users generate jobs which are

submitted through the system using access modules. Access modules, implementing the protocols to interact with economic allocators, route jobs to economic allocators. Economic allocators, managing disjoint sets of nodes, then route jobs to nodes for execution based on whatever resource management policies they implement. Policies can be arbitrary. In our simulations, though, they are based on either economic mechanisms or traditional resource management algorithms (for comparison).

Feedback events model the flow of state information from the nodes providing the resources to the users wanting to make of those resources. These events model the feedback signals used in both market-based and traditional resource management systems. Feedback occurs at multiple levels. Between different levels in the module hierarchy, feedback can be viewed as being periodically written to logical bulletin boards which provide snapshots of feedback information which can then be read by modules interested in that information. In our model, feedback information is only used between immediate layers of the module hierarchy. Use of feedback from neighboring modules in the module hierarchy is consistent with typical implementations.

Feedback information flows up the module hierarchy while providing increasingly comprehensive summaries of system state. At the lowest level, individual nodes provide feedback information (e.g., CPU load) by posting information on node bulletin boards. At the next level, economic allocators provide feedback information (e.g., lowest CPU price) by posting summaries of their state (and their nodes) on EA bulletin boards. At the next level, access modules providing feedback information (e.g., lowest CPU price for all EA auctions monitored by that access module) by posting information on AM bulletin boards. Finally, at the highest level, users use feedback information from AM bulletin boards when deciding which AM to submit their jobs to.

Feedback provides modules at different levels of the hierarchy with information needed to optimize their behavior. At the user level, AM feedback provides information on the current state of the entire system which is used to decide whether jobs should be submitted now or later (e.g., if prices are very high) and how to assign bids (for market-based systems). At the access module layer, EA feedback provides information on how to balance load across different disjoint sets of resources according to configuration-specific policies. Finally, at the economic allocator layer, node feedback provides information which is used by configuration-specific resource management policies to decide how to assign jobs to nodes and how to control their resource allocations.

5.1.2 Examples

By interconnecting modules in various configurations, realistic cluster resource management systems can easily be simulated. A module configuration for the time-shared computational economy (Section 4.2), for example, is shown in Figure 5.2. Here, the mapping from implementation to Stingray modules is straightforward. Users are mapped to user modules; `rexec` is mapped to an AM module; `rexecd` daemons are mapped to EA modules; and nodes with stride CPU schedulers are mapped to node modules. Bulletin board mappings are a little less obvious. `millstat` is mapped to an AM BB since it provides state information on resources accessible through the AM. `vexecd` is mapped to an EA BB since it maintains state information from all EAs received on the cluster multicast channel. Finally, the Linux `/proc` filesystem is mapped onto node BBs since it provides node state information, specifically kernel information made accessible through a filesystem interface.

A module configuration for the Anemone market-based batch queue system (Section 4.3) is equally straightforward to create (Figure 5.3). Users are once again mapped to user modules; `eqsub`, used for user submission of batch jobs, is mapped to an AM module; `eqd`, which manages all batch nodes, is mapped to an EA module; and nodes are are mapped to node modules. Bulletin board mappings exist for access modules and economic allocators only. `eqstat` is mapped to an AM BB since it provides state information on the AM. `eqd`, the sole EA, is mapped to the EA BB since modules query it directly for state information. No node bulletin boards exist since no node feedback information is needed. In Anemone, the only state information that `eqd` is concerned with is whether a node is busy or idle. Thus, relevant node state information can be ascertained by simply keeping track of all jobs that are remotely executed on batch nodes.

5.2 Implementation

Stingray implements the system model using event-driven simulation based on a logical interconnection of simulation modules and bulletin boards. It provides a wide variety of parameterizable building blocks which can be used to construct different types of modules which are snapped together to simulate different types of cluster resource management systems, both market-based and traditional. Simulations are controlled by a simulation engine which continuously processes the earliest event in virtual time across all modules

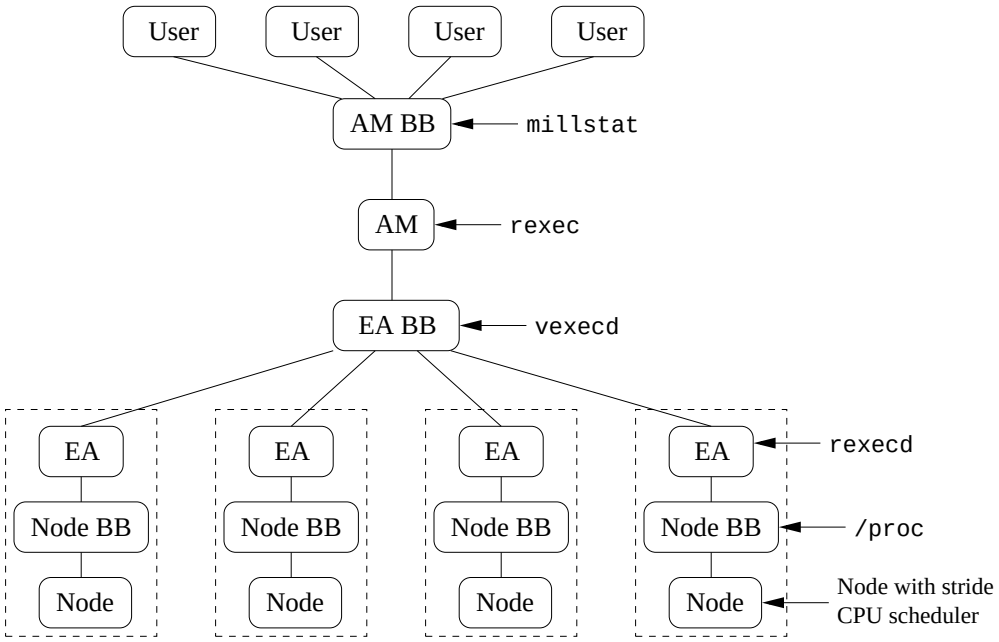

Figure 5.2: **Stingray Module Configuration for the Time-shared Computational Economy.** For modules, the mapping are straightforward. Users are mapped directly onto user modules; `rexec` is mapped to an AM module; `rexecd` daemons are mapped to EA modules; and nodes with stride CPU schedulers are mapped to node modules. Bulletin board mappings are a little less obvious. `millstat` is mapped to an AM BB since it provides state information resources accessible through the AM. `vexecd` is mapped to an EA BB since it maintains state information from all EAs received on the cluster multicast channel. Finally, the Linux `/proc` filesystem is mapped onto node BBs since it provides node state information.

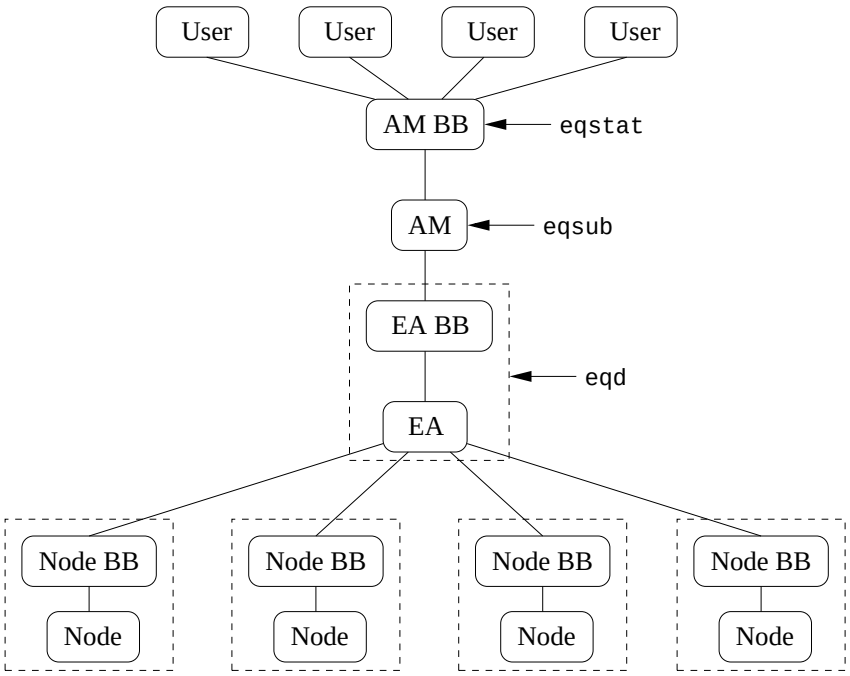

Figure 5.3: **Stingray Module Configuration for the Anemone Market-based Batch Queue system.** Users are mapped to user modules; `eqsub`, used for user submission of batch jobs, is mapped to an AM module; `eqd`, which manages all batch nodes, is mapped to an EA module; and nodes are are mapped to node modules. `eqstat` is mapped to an AM BB since it provides state information on the AM. `eqd`, the sole EA, is mapped to the EA BB since modules query it directly for state information. No node bulletin boards exist since no node feedback information is needed. In Anemone, the only state information that `eqd` is concerned with is whether a node is busy or idle. Thus, relevant node state information can be ascertained by simply keeping track of all jobs that are remotely executed on batch nodes.

until a stopping condition is reached. Simulations are driven by simulation modules which generate new events and provide module-specific event handling code. Simulations are specified using XML configuration files that specify logical interconnections of modules and bulletin boards and their parameterizations. The simulator is implemented in Java and consists of approximately 14,000 lines of code.

5.2.1 Simulation Engine

The simulation engine controls simulations by continuously processing events in virtual time order from a collection of interconnected modules. From the point of view of the engine, modules (users, access modules, economic allocators, nodes) are simply black boxes that contain event queues and have associated event handling code (Figure 5.4). At each step of the simulation, the task of the simulation engine is to dequeue the event with the lowest virtual time from some module's event queue, invoke module-specific event handler code for that event (which may create and enqueue new events on other module event queues), and optionally route that event to a destination module and enqueue it on its event queue (e.g., routing a `JOB_ARRIVAL` event from a user to an access module). The simulation ends when a stopping condition is reached. In our simulations, this occurs when all users have had their entire workloads executed to completion.

5.2.2 Modules

Modules drive simulations by generating new events and providing module-specific event handling code. They use encapsulation to hide details of their implementations. State information is exposed solely through the posting of relevant state information on associated bulletin boards. External access to module functionality is similarly limited. The simulation engine essentially treats all modules as black boxes with event queues and associated event handling code. Instances of each module type (users, AMs, EAs, nodes) have expectations on what types of events they should be able to handle. The details of how they handle those events, though, is entirely up to the specifics of those particular instances. Similarly, details on when and which events to generate are also up to the specifics of module implementations.

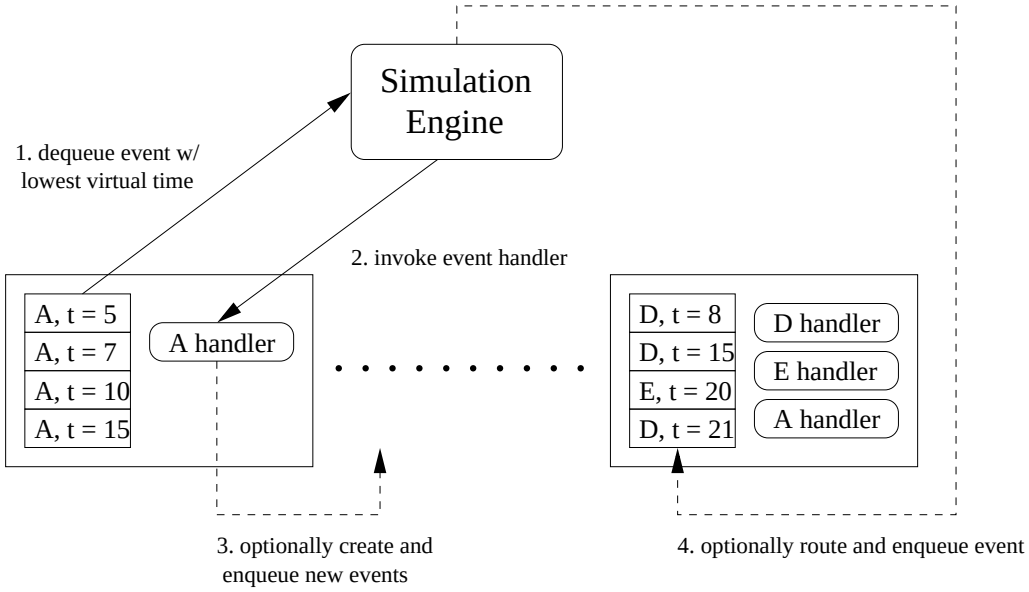

Figure 5.4: **Event-driven Simulation in Stingray.** The simulation engine controls simulations by continuously processing events in virtual time order from a collection of logically interconnected user, access module, economic allocator, and node modules. At each step of the simulation, the simulation engine dequeues the event with the lowest virtual time from some module's event queue, invokes module-specific event handler code for that event (which may create and enqueue new events on other module event queues), and optionally routes that event to a destination module and enqueues it on its event queue. The simulation ends when all user workloads have been executed to completion.

User Modules

User modules simulate users and applications and are modeled using five variables: (1) *workloads*, which specify when users submit jobs and the resource requirements associated with those jobs, (2) *valuations*, which for each job, specify the time-dependent value delivered to the user, as a function of execution time, (3) *funding*, which specifies policies on how users earn money to spend on resources to run their jobs, (4) *bidding strategies*, which specify how users map valuations to "bids" using economic allocator-specific economic mechanisms and (5) *response to feedback*, which specifies how users alter their behavior in response to various types of feedback (e.g., price feedback, performance feedback).

Workloads specify a user's job arrival process and the parameters associated with each arriving job. The arrival process models when a users uses the system and, when the user does submit jobs, how those job submissions are distributed over time. For example, one arrival process might model a user that uses the system infrequently but submits a large number of batch jobs when jobs are submitted. Job parameters describe computational resource requirements of each job, such as job length in CPU hours (at some reference CPU's speed), parallel degree, and perhaps other resource demands as well (e.g., physical memory footprint). These parameters, in conjunction with node resources and simulated execution, are used to compute simulated job execution times.

Valuations specify the time-dependent value delivered to the user as a function of job execution time. Use of valuations as part of the model is one of the main points of departure for market-based systems as compared to traditional systems. Whereas traditional systems optimize for system-centric performance metrics, market-based systems optimize for total valued delivered to users. In order to quantify the benefits of optimizing for value, we need a model of how user value depends on execution time. Valuations in our simulations are used for this purpose. Valuations can be used to express both the importance of the job, expressed by the magnitude of the valuation, and the immediacy of the job, expressed by how fast value decays over time.

Funding policies specify how users earn money over time in order to pay for the resources to execute their jobs. A funding policy specifies parameters such as the rate at which a user's account is credited with money and optionally an upper bound on the total amount of money a user can accumulate [82]. Since this work is focused on using market-based resource management in a single administrative domain and we do not focus on cases where buyers can also be sellers, buyers have no means to earn their own money. Therefore,

user funding and consequently buying power is determined solely by system funding policies, which presumably would be set by a system administrator.

Bidding strategies specify how users with specific valuations on resources assign bids on those resources when submitting their jobs to the system. In this dissertation, we examine systems that use both incentive-compatible economic mechanisms and non-incentive-compatible economic mechanisms. Given this, we examine two classes of users, one class where users assign bids which are consistent with their valuations and another class where users assign bids which might only be approximations of their valuations or based on entirely different information. The latter case is intended to model strategic users facing non-incentive-compatible economic mechanisms. We use this class of users later to examine the impact of strategic users in systems that use non-incentive-compatible mechanisms for resource allocation.

Response to feedback models how users alter their behavior based on the state of the system. Response to feedback is modeled specifically for simulated systems where incentives exist for users to observe and respond to feedback or users are strategic. As previously discussed, effective feedback signals are ones which provide incentives for users to both observe and respond to them. Traditional systems do not provide effective feedback signals because there are no incentives for users to respond to them. In contrast, market-based systems provide price incentives which can be used to alter user behavior (e.g., to encourage deferring of new jobs during periods of high load). Response to feedback is also relevant for strategic users. In systems with non-incentive-compatible mechanisms, strategic users make observations of system state and use this in combination with a bidding strategy to decide how to alter their behavior.

AM Modules

AM modules simulate access modules and are modeled using configurable routing policies which specify how access modules select the best economic allocator when deciding where to route incoming jobs. Routing policies are implemented by tracking a set of economic allocators which (i) post the same type of state to a common bulletin board and (ii) accept the same type of bids for their economic mechanisms. Given the state of all economic allocators in a bulletin board, access modules implement policies which select the best economic allocator based on this state. For example, for batch queue economic allocators that use Vickrey auctions, an access module selection policy might choose the

economic allocator that posted the lowest last winning price in its last Vickrey auction.

AM modules are assumed to operate with no overhead and use delayed state information on economic allocators when making routing decisions. For workloads comprised of long-running, compute-intensive applications, assuming no access module overhead has negligible effects on overall results since overheads would be small anyway compared to application execution times. Delayed information, however, must be modeled since we cannot assume that access modules always have up-to-date information on all economic allocators. Such an assumption would require that access modules instantaneously obtain current state information on all economic allocators each time they route an incoming job. In a large system, the overheads of this approach would likely outweigh the benefits obtained. Instead, we generally assume that AM modules simply read from EA bulletin boards and economic allocators decide how frequently to post updates.

EA Modules

EA modules simulate economic allocators and are modeled using two variables: (1) *resource management policies*, which specify how incoming jobs are scheduled onto nodes and what resource allocations they receive and (2) *bulletin board update policies*, which specify how frequently economic allocators post state updates to associated bulletin boards. Using EA modules, both centralized and decentralized resource management systems can be simulated (e.g., see Section 5.1.2 for examples). Like AM modules, EA modules are assumed to incur no additional overhead. Again, this assumption is realistic given our assumption of long-running, compute-intensive applications and the specific resource management algorithms we use.

Resource management policies specify how incoming jobs are scheduled onto nodes and what resource allocations they receive. Policies can be based on either economic mechanisms or traditional resource management algorithms (for comparison in the evaluation). In either case, the task of the EAs is still the same: economic allocators implement resource management policies using combinations of job parameters, admission control, node resource managers, and node state information. For example, when simulating the time-shared computational economy, EA modules schedule incoming jobs onto nodes using (i) job CPU requirements, parallel degrees, and bids, (ii) ticket assignments for node proportional-share CPU schedulers, and (iii) current node prices on all nodes managed by the economic allocator.

Bulletin board update policies specify how frequently economic allocators post state updates to associated bulletin boards. In most cases, bulletin board updates are simply periodic updates and so policies are simply a period. In more specialized cases, we consider other policies such as economic allocators sending bulletin board updates both periodically and whenever a significant state change occurs. These cases are used mainly to investigate the impact of algorithms that attempt to mitigate the effects of delayed information. `vexecd` daemons in the time-shared computational economy, for example, implement an update policy that sends both periodic updates and updates whenever a job starts or ends.

Node Modules

Node modules simulate cluster nodes and are modeled using three variables: (1) *resource supply*, which specifies node supply of scarce, performance-critical resources, (2) *behavior under contention*, which models how sharing of scarce, performance-critical resources affects application execution time and (3) *bulletin board update policies*, which specify how frequently nodes post state updates to associated bulletin boards. Node modules are implemented using a highly simplified model of execution that leverages our earlier assumptions on cluster workloads. There, we assumed that workloads consist primarily of long-running, compute-intensive applications. Node modules leverage this assumption by focusing primarily on competition for CPU time when simulating job execution.

Resource supply specifies the quantity of scarce, performance-critical resources possessed by each node. Which resources are scarce and performance-critical depends both on supply (node resources), demand (workload), and application characteristics. Based on our workload assumption, CPU is the only scarce, performance-critical resource and thus this is what node modules model. CPU time, however, is not well-defined since different CPUs run at different speeds and have different instructions sets. Rather than consider heterogenous CPUs, we make the simplifying assumption that all node modules consist of a single, identical CPU. Assuming all nodes are homogeneous simplifies the presentation and analysis of our simulation results. Future studies could relax this assumption if doing so would enable interesting follow-on simulations.

Behavior under contention models how sharing of scarce, performance-critical resources affects application execution time. Again, we leverage our assumption on workloads when modeling this behavior. Since CPU allocations are the main determining factor for

compute-intensive jobs, node modules model execution time as simply being k times longer when jobs are allocated $1/k$th of the CPU. With large enough scheduling quantums (e.g., 50-100 ms scheduling quantums for most OS schedulers), all second order effects such as context switches can basically be ignored and so this approximation is realistic.

Bulletin board update policies specify how frequently nodes post state updates to associated bulletin boards. In all cases we consider, bulletin board updates are simply periodic updates and so policies are simply a period.

5.2.3 Simulation Configuration Files

Stingray simulations are specified using XML configuration files which describe logical interconnections of simulation modules and bulletin boards and their parameterizations. An XML DTD is used to specify all possible options for module interconnections and parameterizations. Each configuration file must conform to this DTD and consists of a `globals` element which specifies "global" parameters, arbitrary numbers of `users`, `am`, `ea`, and `nodes` elements for user, AM, EA, and node modules, and arbitrary numbers of `am_bb`, `ea_bb`, `node_bb` elements for AM, EA, and node bulletin boards. Using various interconnections of parameterized modules and bulletin boards, a wide range of cluster resource management systems under various workloads can be simulated. Figure 5.5 shows a sample interconnection of modules and bulletin boards. Figure 5.6 shows the XML from the corresponding configuration file.

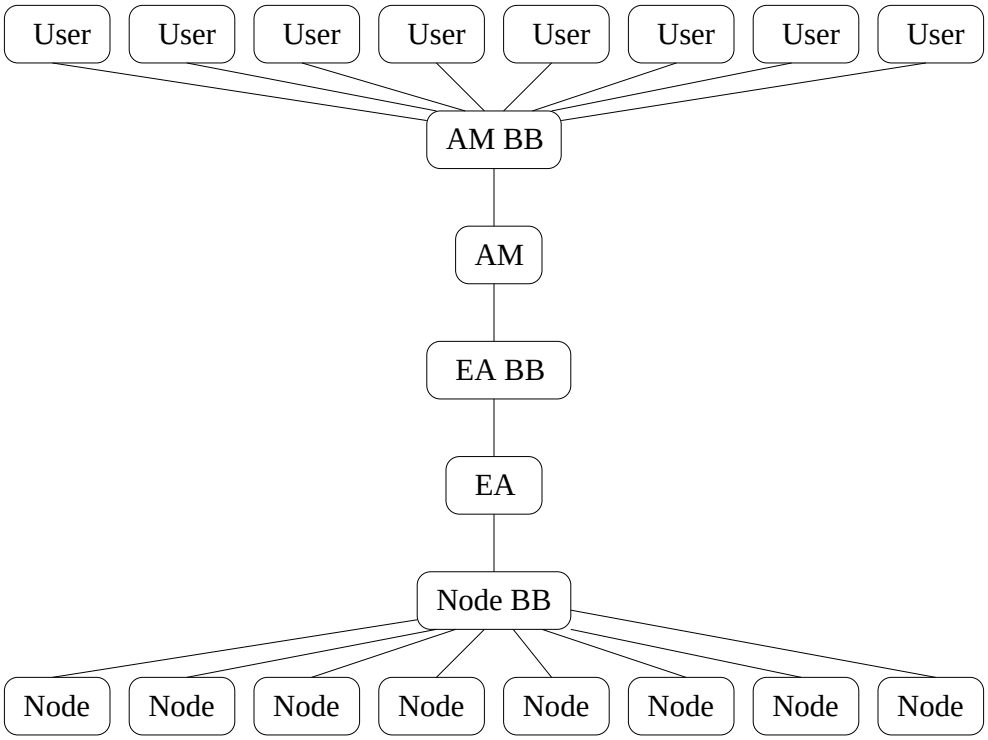

Figure 5.5: **Sample Interconnection of Simulation Modules and Bulletin Boards.** In this configuration, eight users generate workloads which are submitted to a centralized resource management system that manages eight cluster nodes. Such a configuration might be used to simulate resource management on a market-based batch queue system for example.

```
<users num=8" id="users1" type="stingray.user.BatchCpuUser"
       am_bb_id="am_bb1"
       ag_base_seed="11" rrg_base_seed="22"
       bg_base_seed="33" ufg_base_seed="33"/>
   <job_gen num_jobs="1000">
      <exponential_ag start_time="0" mean="250"/>
      <batch_cpu_rrg>
         <uniform_pd min="1" max="4"/>
         <exponential_pd mean="125" min="1"/>
      </batch_cpu_rrg>
      <scaled_util_bg><constant_pd constant="1"/></scaled_util_bg>
      <pwl_decay_ufg>
         <uniform_pd min="1" max="100">
         <constant_pd constant="0">
         <constant_pd constant="1">
         <constant_pd constant="1">
      </pwl_decay_ufg>
   </job_gen>
   <bank_account init_balance="1000" periodic_credit="1000"/>
</users>

<am_bb id="am_bb1">

<am id="am1" type="stingray.am.BatchCpuAM"
    bb_update_period="60"
    am_bb_id="am_bb1" ea_bb_id="ea_bb1">
   <batch_cpu_am_params algorithm="MIN_LAST_WINNING_BID"/>
</am>

<ea_bb id="ea_bb1"/>

<ea id="ea1" type="stingray.ea.FirstPriceBatchCpuEA"
    sched_period="30" bb_update_period="60"
    ea_bb_id="ea_bb1" node_bb_id="node_bb1"
    nodes_id="nodes1">
   <first_price_batch_cpu_ea_params backfill="true"/>
</ea>

<node_bg id="node_bb1">

<nodes num="8" id="nodes1" type="stingray.node.BatchCpuNode"
       bb_update_period="60" node_bb_id="node_bb1"/>
```

Figure 5.6: **Snippet from Example Configuration File.** This snippet creates modules (`users`, `am`, `ea`, `nodes`) and bulletin boards (`am_bb`, `ea_bb`, `node_bb`) in the interconnection pattern shown in Figure 5.5. Based on its parameterization, it corresponds to a centralized batch queue system that uses an auction for resource allocation on eight nodes. Eight users generate workloads which consist of exponential job arrival processes (mean = 250) generating parallel batch jobs with mean CPU length 125. Valuations functions are piecewise-linear and decay over time. Funding is periodic and credits each user's account 1000 units of funding. Bidding strategies are to bid based on true value. No response to feedback is simulated in this example. Access modules route to a centralized economic allocator. The economic allocator implements a first price auction and implements a backfill algorithm. All bulletin board updates for the access module, economic allocator, and nodes are periodic and occur every 60 time units.

Chapter 6

Simulations

In this chapter, we present a performance evaluation of market-based cluster resource management. We use Stingray to simulate market-based and traditional systems in both time-shared and batch mode and to quantify their performance in terms of aggregate utility. In all simulations, systems configurations were generally fixed. Workloads and users, on the other hand, were systematically varied across their key dimensions: job arrival times, job resource requirements, workload burstiness, utilization, user resource valuations, and user sensitivity to delay. Through an extensive series of simulations, we answer the two questions posed earlier in this dissertation, namely whether (i) explicit resource valuations and (ii) having price feedback can lead to significantly higher value delivered to users. In doing so, we show under what types of workloads and in which regimes of operation market-based systems are most effective compared to traditional systems.

6.1 Evaluating Performance

In all simulations, we use aggregate utility as the performance metric for comparing market-based systems with traditional systems. We use aggregate utility as the metric because traditional performance metrics are not accurate measures of total value delivered to users. To compute aggregate utility, we assume all users have valuations which are piecewise-linear and decay linearly over time as a function of slowdown (Figure 6.1). For workloads consisting largely of scientific applications, such valuations are realistic since users are mainly concerned with completion time and expectations on completion time are related to job CPU requirements. We compute aggregate utility by taking the value delivered for

each job, as computed by the corresponding user's valuation, and summing over all jobs.

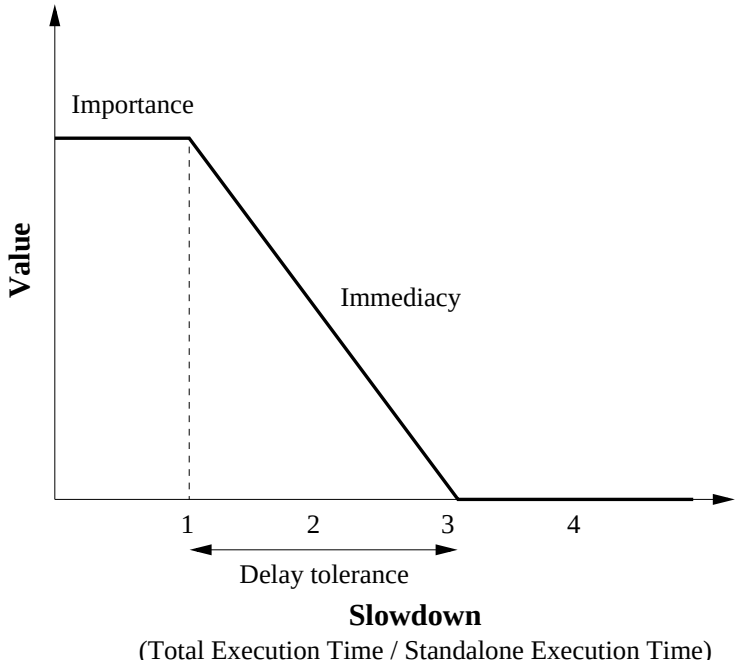

Figure 6.1: **Piecewise-linear Valuation.** This figure shows the form for all valuations in our simulations. Valuations are piecewise-linear functions which decay linearly over time as a function of slowdown. Value is assumed to be constant up to a slowdown of 1 since users cannot expect a job to finish sooner than it would with dedicated access. Value then decays linearly until slowdown reaches some critical value, at which point value delivered is 0 regardless of completion time. We assume users have some tolerance for delays due to sharing which is modeled by the slope of linear decay. We call the length of time when value begins to decay to the time when value decays to 0 the user's *delay tolerance* and measure it as a multiple of job length. A delay tolerance of two, for example, as shown above means that a user can tolerate delays of up to two extra job lengths before the job delivers no value.

We use the sum of the valuations to compute aggregate utility for both traditional and market-based systems. In both cases, using the sum of the valuations is appropriate since the bottom line for delivering value to users is by definition the sum of the valuations. Using the sum of the valuations also allows us to make meaningful comparisons between the two types of systems. To support these types of comparisons, periodic funding (positive utility) and costs associated with resource consumption (negative utility) are not factored into performance when evaluating market-based systems. Instead, we assume that cost only plays a role in determining user behavior. For example, when auctions are used to allocate

resources, users take cost into account when assigning their bids and do not assign high bids to low valued jobs.

In this dissertation, the use of funds in market-based systems is solely a means to ultimately deliver more value to users. Funds have no intrinsic value and are only useful in exchange for computational resources on a single cluster. Because of this, it makes no sense to factor periodic funding or costs into performance for market-based systems. If these variables were factored into performance, then it would be possible that a market-based system could "perform" better or worse than a traditional system even when processing the same workload in an identical fashion. In addition, resource management policies that artificially delay execution, while obtaining more periodic funding, would also appear to be delivering better performance when in fact they are actually delivering less value to users. To avoid these anomalous results, we use cost solely for modeling user behavior.

6.2 Time-shared Simulations

In this section, we analyze the performance of the time-shared computational economy in terms of aggregate utility. We use sensitivity experiments to measure performance sensitivity to variations in valuation distributions and a variety of workload parameters. Using a series of these experiments, we test the two main hypotheses in this dissertation, namely whether or not (i) resource valuation information and (ii) price feedback in market-based systems have significant impact on total valued delivered to users. Results from these experiments form the basis in supporting our thesis. With these results in place, we then examine other aspects of market-based systems. More specifically, we analyze the impact of delayed information on system performance and make some headway into understanding the performance impact of strategic users when the underlying economic mechanism is not incentive-compatible.

In all experiments, we compare two resource management systems. The first system is a market-based system and is essentially identical to the time-shared computational economy described earlier. We refer to this system as PROPRATE. The second system is a traditional cluster resource management system. It uses centralized load balancing and local, fair-share CPU schedulers on each node [1]. We refer to this system as FAIRSHARE. For all simulations, the system configuration consisted of 32 users, centralized AM and

[1] For compute-intensive applications, priority CPU schedulers are essentially fair-share schedulers over long intervals of time.

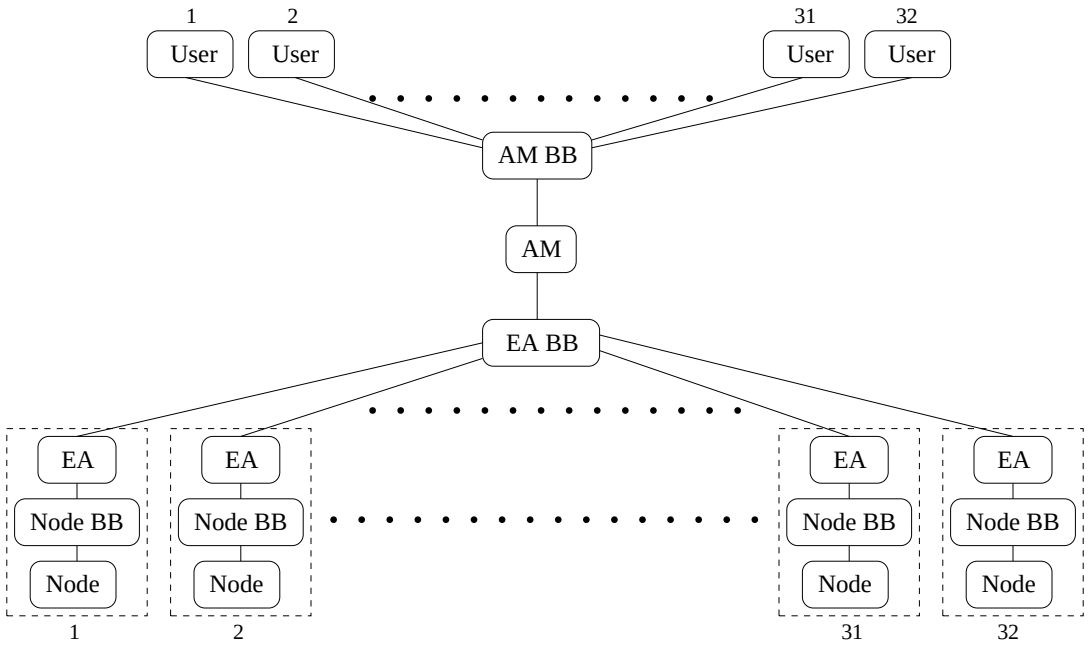

Figure 6.2: **System Configuration for Time-shared Simulations.** The system has 32 users generating workloads which are parameterized based on the specific experiment being performed. It uses centralized AM and EA bulletin boards and a single access module. The cluster consists of 32 nodes with each node being independently managed by its own economic allocator. The configuration is consistent with the organization of the time-shared computational economy. It is equally applicable for modeling traditional resource management systems such as GLUnix [37]. With GLUnix, we simply view the economic allocators as having no function other than relaying load information back to the EA BB which is used by the AM for centralized load balancing.

EA bulletin boards, and 32 economic allocators each managing one of 32 cluster nodes (Figure 6.2). Each user had a workload consisting of 1000 jobs which were parameterized depending on the specific experiment being performed. Each simulation ran all user workloads to completion and measured aggregate utility delivered.

6.2.1 Assessing the Benefits of Valuation Information

In these experiments, we test the hypothesis that resource management based on explicit resource valuations leads to significantly higher value delivered to users. We do this by performing a series of performance sensitivity experiments. First, we examine performance sensitivity to variations in resources valuations with realistic valuation distributions. Second, we investigate how user tolerance to delay impacts system performance. Third, we examine performance sensitivity to burstiness in the workload when users submit bursts of jobs as opposed to individual jobs. Finally, we examine performance sensitivity to system utilization for a realistic cluster workload. In all experiments, the focus is on quantifying the gains in aggregate utility that can be achieved by simply having information on user resource valuations. No price feedback was used. Users submitted jobs immediately regardless of the prices they were facing.

Variation in Resource Valuations

The first experiment examines performance sensitivity to variations in user resource valuations. We examine variation in valuations because the impact of having resource valuation information is directly related to variation in that information. If there were no variation in valuations, then FairShare would perform optimally since all jobs would be equally valuable and providing differential resource allocations would not matter. For this experiment, we assume that valuation distributions are bimodal and that x% of the valuations come from a low normal distribution and 100 - x% of the valuations come from a high normal distribution. Modeling valuations as bimodal is a reasonable assumption and is consistent with the data we have collected on actual usage with the time-shared computational economy.

In Figure 6.3, we present results for two bimodal distributions. The first case has low valuations centered around a normal distribution with mean $\mu_{low} = 50$ and high valuations centered around a normal distribution with mean $\mu_{high} = 500$. The second case has low valuations centered around a normal distribution with $\mu_{low} = 50$ and high valuations

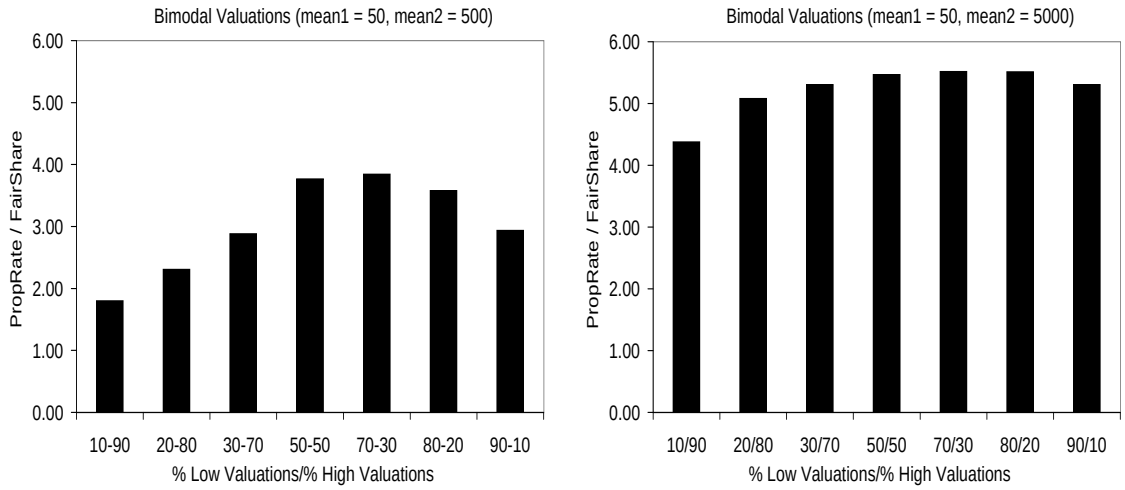

Figure 6.3: **Time-shared Performance Sensitivity to Variation in Resource Valuations.** These figures plot the ratio of aggregate utility for PROPRATE versus FAIRSHARE as a function of resource valuation distributions. For each set of results, resource valuations were bimodal with x% of the valuations coming from a normal distribution with mean 50 (low valuations) and 100 - x% of the valuations coming from a normal distribution with means of 500 and 5000 (high valuations). The system was fully utilized. Workloads consisted of 32 users submitting sequential jobs with exponential interarrivals and constant CPU lengths. All users had a delay tolerance of 4.

centered around a normal distribution with $\mu_{high} = 5000$. (Variance in both cases was 0.2μ.) Valuations were piecewise-linear (Figure 6.1) and all users had a delay tolerance of 4. The system was fully utilized and workloads consisted of 32 users submitting sequential jobs with exponential interarrivals (mean = 3600) and constant CPU lengths (3600). We used constant length jobs to isolate the effects of variation in valuations and avoid additional variations in the rates used by PROPRATE, since rates are value divided by CPU time.

Figure 6.3 plots the results as we vary the mix of low and high valued jobs. We observe that PROPRATE outperforms FAIRSHARE by wide margins for all mixes simulated with the benefit of knowing valuations increasing with larger differences in valuations. In most cases, we see that PROPRATE at least doubles or triples the performance of FAIRSHARE. The differences are especially large when the ability to identify high value jobs amongst a mix of low valued jobs becomes important. For bimodal distributions with a 10x difference in low/high value means, we observe these types of gains whenever there is a substantial number of low valued jobs competing with high valued jobs. For the 100x difference case, benefits apply in all cases since even when almost all jobs are valuable (10/90 case), the low valued 10% that contends with the high valued jobs still have a significant impact since the difference in valuations is so large.

As further demonstration that variation in valuations leads to higher aggregate utility when valuation information is used, Figure 6.4 shows results for the same experiment as Figure 6.3 except with realistic distributions on job CPU requirements. The first set of simulations shows results when jobs have CPU lengths taken from a normal distribution with mean 3600. The second set of simulations shows results when jobs have CPU lengths taken from an exponential distribution with mean 3600. In both cases, we see that PROPRATE still substantially outperforms FAIRSHARE. The gaps in performance are particularly large for exponential CPU length distributions. This difference can be attributed to exponential distributions having larger numbers of smaller jobs which in turn leads to especially high rates since high valuations are being applied to small amounts of CPU time.

User Tolerance to Delay

The second experiment examines performance sensitivity to variations in tolerance to delay. In the first experiment, we assumed delay tolerances were fixed at 4 in all simulations. Thus assumption, however, clearly does not hold for all workloads. In particular, if users are especially tolerant of delays in their job execution times, it seems plausible that

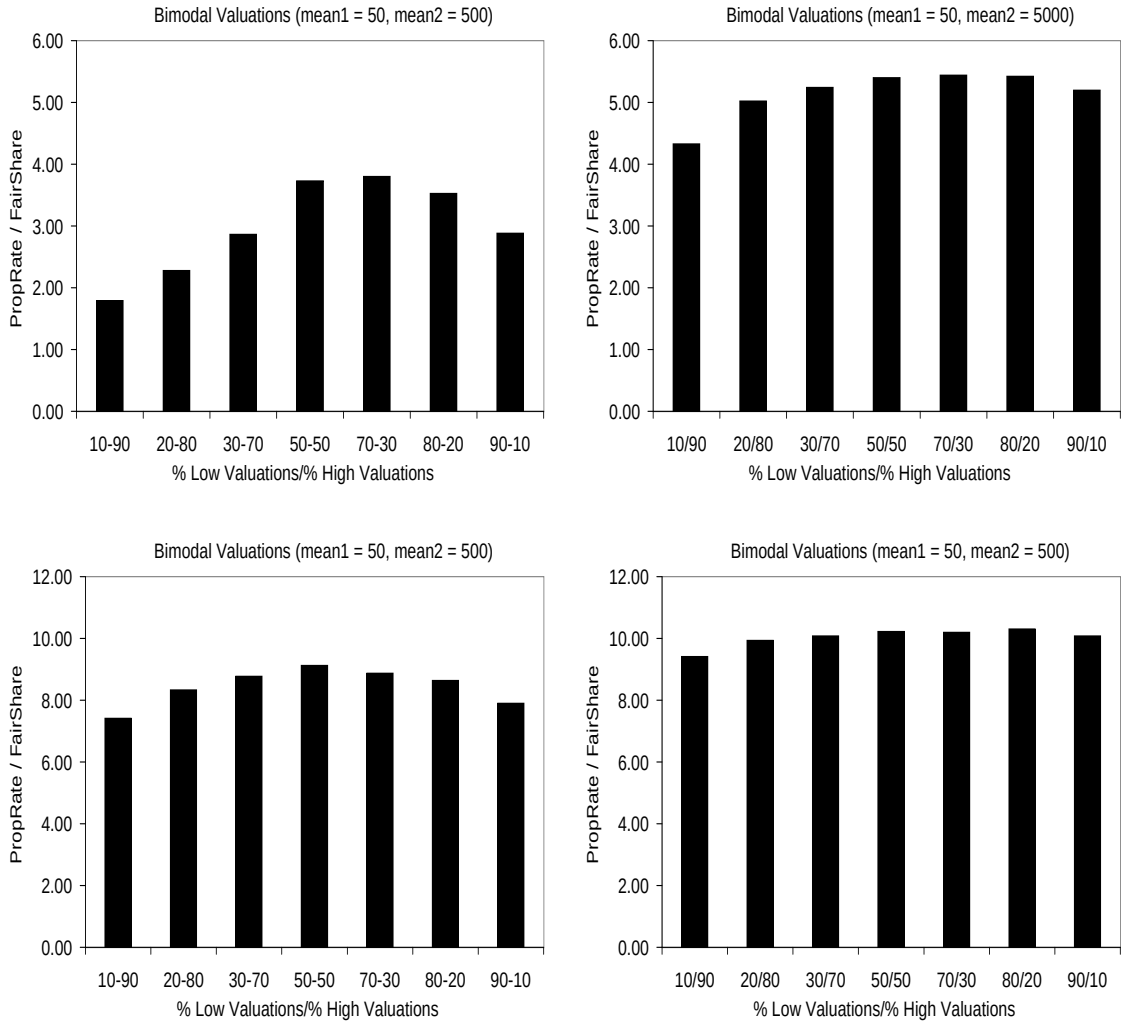

Figure 6.4: **Time-shared Performance Sensitivity to Variation in Resource Valuations (Normal and Exponential Length Jobs).** These figures plot the ratio of aggregate utility for PROPRATE versus FAIRSHARE as a function of resource valuation distributions. For each set of results, resource valuations were bimodal with x% of the valuations coming from a normal distribution with mean 50 (low valuations) and 100 - x% of the valuations coming from a normal distribution with means of 500 and 5000 (high valuations). The system was fully utilized. Workloads consisted of 32 users submitting sequential jobs with exponential interarrivals and normal (top) or exponentially distributed (bottom) CPU lengths. All users had a delay tolerance of 4.

FAIRSHARE may actually perform well. In Figure 6.5, we test this hypothesis by quantifying the impact that user tolerance to delay has on aggregate utility. As before, we use a fully utilized system and workloads consisted of 32 users submitting sequential jobs with exponential interarrivals and normal CPU lengths. Valuations were taken from bimodal valuation distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) where 80% of the jobs are low value and 20% of the jobs are high value. We refer to these types of distributions as 80/20 bimodal distributions.

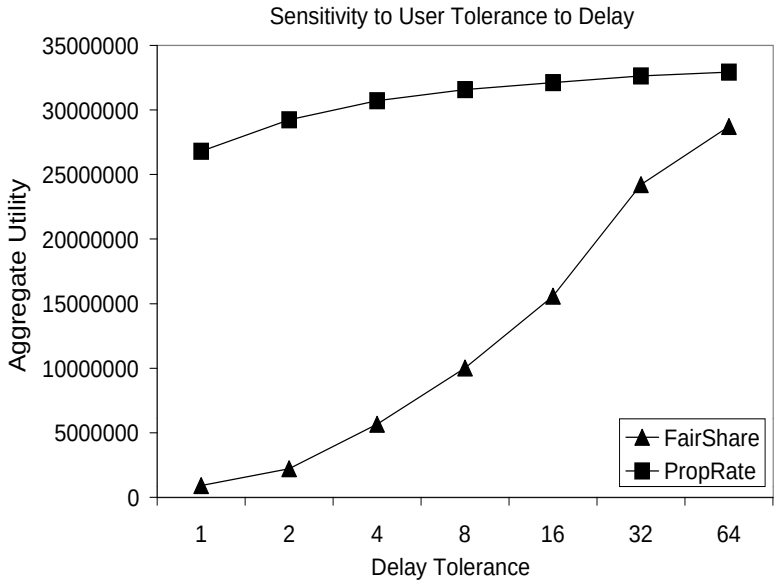

Figure 6.5: **Time-shared Performance Sensitivity to User Tolerance to Delay.** This figure plots aggregate utility as a function of user tolerance to delay. The system was fully utilized. Workloads consisted of 32 users submitting sequential jobs with exponential interarrivals and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4.

From Figure 6.5, we observe that even when users are moderately insensitive to delay that market-based systems still deliver significantly more value. We observe that PROPRATE delivers significantly more utility compared to FAIRSHARE over a wide range of user tolerances to delay. For example, when users are willing to tolerate slowdowns up to 16 times their jobs' standalone execution times, we still observe a twofold increase in performance using a market-based system. Again, this difference in performance can be directly attributed to the extra information that PROPRATE is able to use whenever

contention occurs. When contention occurs with FAIRSHARE, the system can do no better than to assign equal resource allocations for both high and low valued jobs since it cannot distinguish between the two.

Workload Burstiness

The third experiment examines performance sensitivity to burstiness in the workload. In many systems, users often submit jobs in bursts as opposed to submitting individual jobs separated by shorter periods of idle time. In this experiment, we quantify the performance impact of this by examining performance sensitivity to burstiness as function of job burst sizes. Larger burst sizes lead to more contention and subsequently more opportunity for PROPRATE to take advantage of its extra information. In these experiments, system utilization was fixed at 50% in order to highlight the performance behavior of FAIRSHARE, which otherwise plummets rapidly to 0 for high utilizations. Workloads consisted of 32 users submitting bursts (fixed #jobs per burst) of sequential jobs with normal interarrival times between bursts, and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4. Figure 6.6 plots aggregate utility versus burst size for three different delay tolerances: 4, 8, and 16.

The results show that as workloads become burstier the benefits of knowing valuations increases, with the performance of PROPRATE and FAIRSHARE diverging with larger burst sizes. We see that when users have moderate tolerance to delay (delay tolerance = 4), FAIRSHARE's performance drops off rapidly as burst size increases and drops to nearly 0 when the burst size is at least 5. As users are more tolerant to delay, the performance descent of FAIRSHARE becomes more gradual. With a delay tolerance of 16, for example, aggregate utility drops by about 50% with a burst size of 7. In contrast, PROPRATE demonstrates significantly better degradation in performance with larger burst sizes. As burst sizes increase, though, even PROPRATE suffers some performance degradation due to cases where bursts include multiple high valued jobs which compete directly against each other since they have comparable resource allocations.

Utilization

The fourth experiment examines performance sensitivity to utilization. As utilization increases, so does the probability of contention and potential opportunities for delivering more value with market-based systems. In this experiment, we quantify these

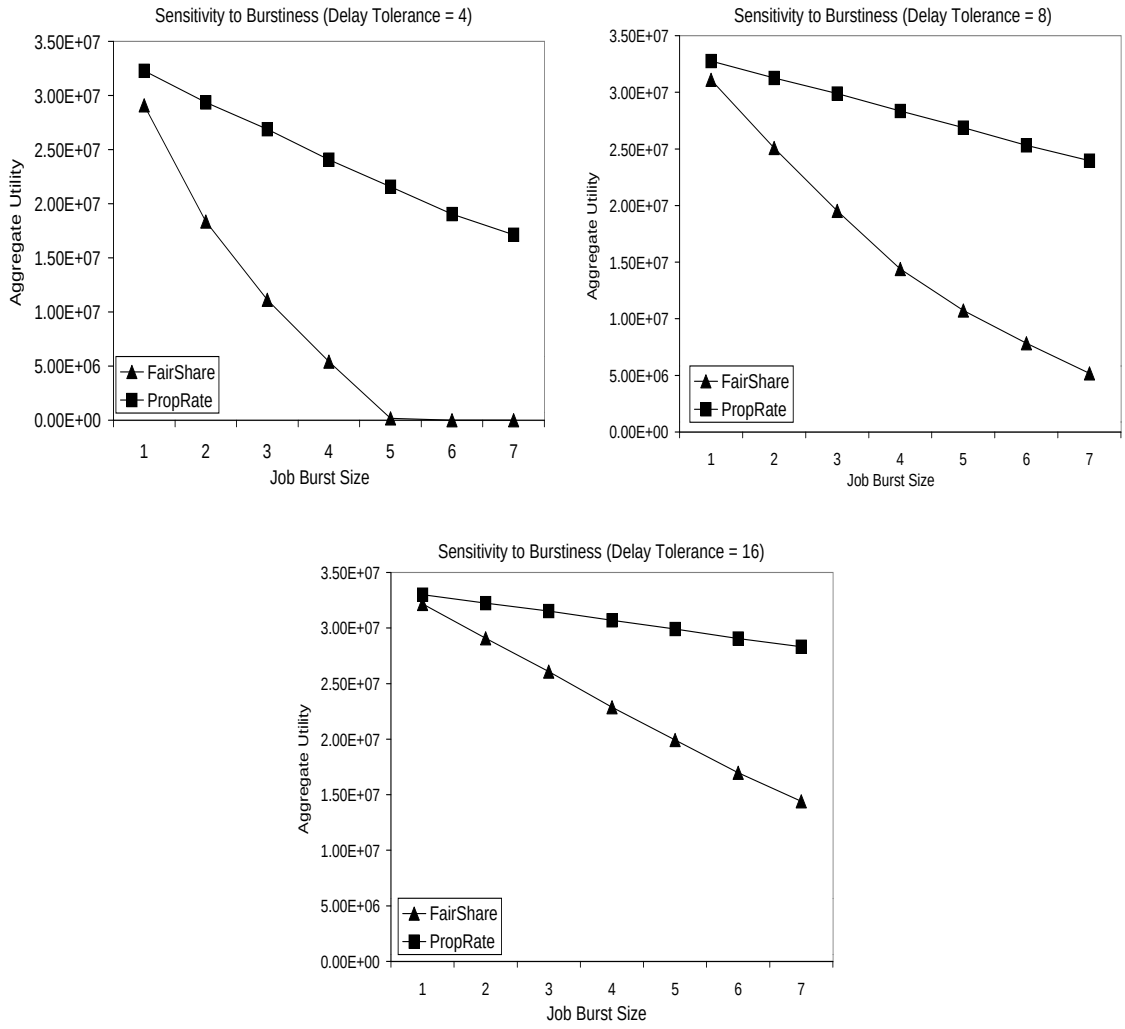

Figure 6.6: **Time-shared Performance Sensitivity to Burstiness in the Workload.** These figures plot aggregate utility as a function of job burst size. The system was 50% utilized in order to highlight the performance behavior of FairShare, which otherwise plummets rapidly to 0 for high utilizations. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts, and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a user delay tolerance of 4.

benefits using a realistic cluster workload with moderate amounts of burstiness. In this experiment, workloads consisted of 32 users submitting sequential jobs with normal job burst interarrivals, exponentially distributed job burst sizes (mean = 4), and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4. Figure 6.7 shows the results.

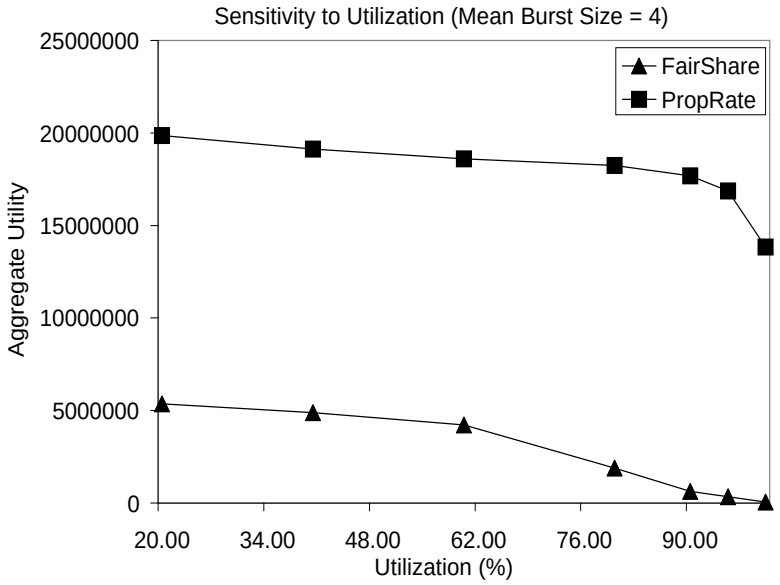

Figure 6.7: **Time-shared Performance Sensitivity to Utilization.** This figure plots aggregate utility as a function of utilization with a typical cluster workload. In this experiment, workloads consisted of 32 users submitting bursts of sequential jobs with normal job burst interarrivals, exponentially distributed job burst sizes (mean = 4), and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4.

We observe that when workloads are even moderately bursty that the value of knowing valuations is significant even at low utilization due to the increased degree of contention that competing processes face. We see that over a wide range of utilizations, PROPRATE consistently outperforms FAIRSHARE by at least a factor of 4 with performance improvements increasing with utilization due to increased likelihood of contention. As utilization exceeds 60%, we see that FAIRSHARE's performance begins to rapidly degrade and delivers almost no value by the time utilization reaches 90%. In contrast, PROPRATE shows much better performance degradation, delivering only 30% less value as utilization increases from 20% to 100%.

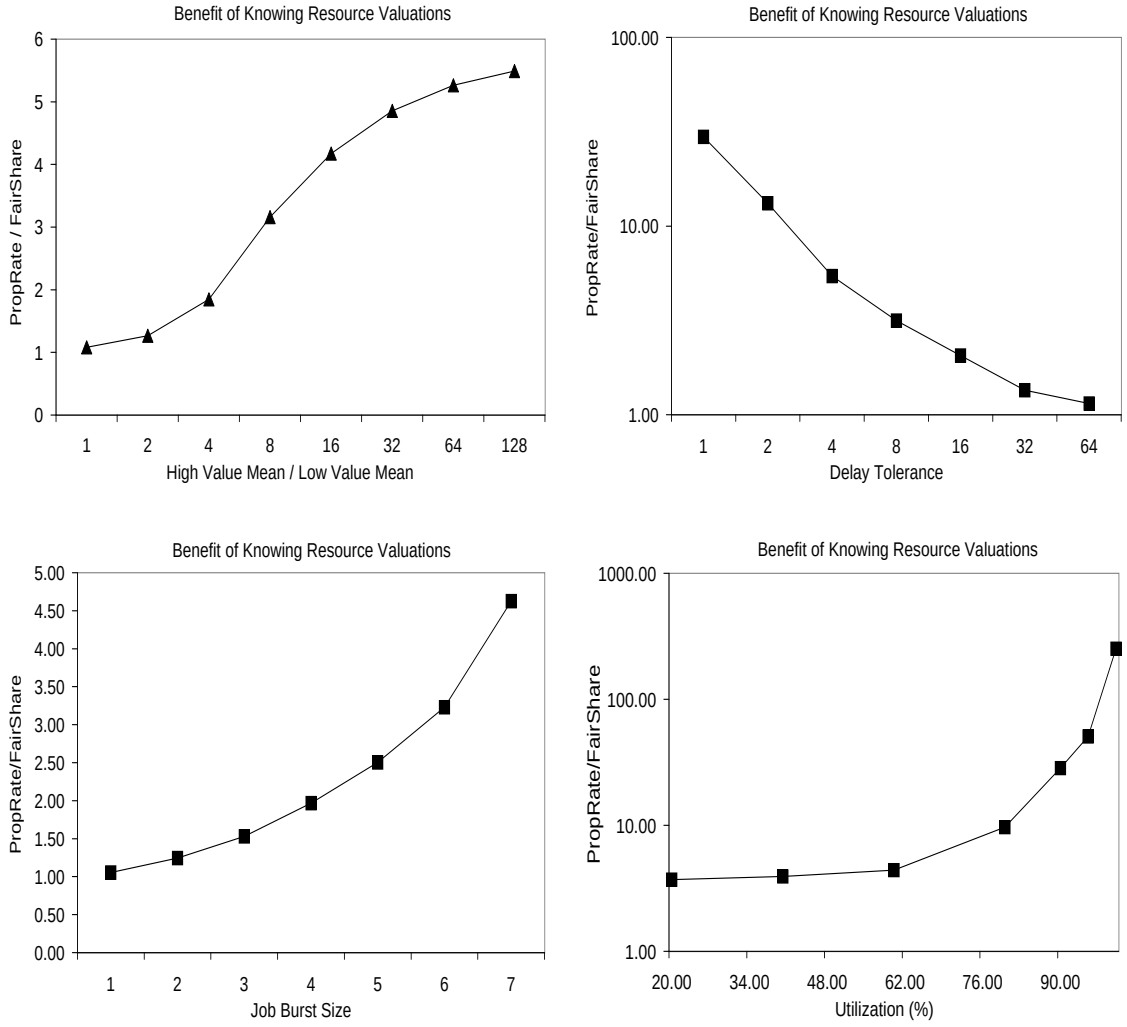

Figure 6.8: **Benefit of Knowing Resource Valuations in Time-shared Environments.** In summary, we observe that by providing users a way to express resource valuations and optimize for it, significantly higher value can be delivered compared to traditional approaches for time-shared environments. We see that the benefits of having valuation information are most pronounced when workloads are bursty and there is a wide range of valuations on the underlying computational resources. We also observe that while valuation information is most beneficial when workloads are highly demanding, it is also useful in less demanding environments. Specifically, even when users are fairly tolerant of delays and at low levels of utilization the benefits of having valuation information are substantial with even a moderately bursty workload. These benefits become dramatic as users become more sensitive to delay and utilization increases.

Summary

In summary, we observe that by providing users a way to express resource valuations and optimize for it, significantly higher value can be delivered compared to traditional approaches for time-shared environments (Figure 6.8). Through a series of experiments, we have quantified this by examining performance sensitivity to variation in valuations, user tolerance to delay, workload burstiness, and utilization for systems with and without valuation information. We see that the benefits of having valuation information are most pronounced when workloads are bursty and there is a wide range of valuations on the underlying computational resources. We also observe that while valuation information is most beneficial when workloads are highly demanding, it is also useful in less demanding environments. Specifically, even when users are fairly tolerant of delay and at low levels of utilization the benefits of having valuation information are substantial with even a moderately bursty workload. These benefits become dramatic as users become more sensitive to delay and utilization increases.

6.2.2 Assessing the Benefit of Price Incentives

In these experiments, we quantify the impact of price incentives with users who observe and respond to feedback. We assume responsive users respond to feedback by deferring job submissions when faced with high prices, with the hope that resubmitting at a later time will result in larger CPU allocations and lower overall cost. We model this responsiveness using three parameters: (i) *#retries*, the maximum number of times a user will defer, (ii) *rate threshold*, the threshold where users decide whether to submit or defer, and (iii) *retry time*, the amount of time users wait before reevaluating job submissions. We examine performance sensitivity to these three parameters and compare the performance of PropRate with unadaptive users to PropRate with responsive users. In all cases, prices were determined solely by the rates of competing jobs. In addition, jobs that were deferred were assigned rates based on their valuation at the time of actual submission.

Maximum Number of Retries

The first experiment measures performance sensitivity to the maximum number of retries for users who are responsive to price feedback. We use a maximum number of retries because we anticipate that users only have so much patience before they submit jobs

regardless of current prices. In this experiment, workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4. The system was fully utilized and burst sizes were exponentially distributed with means of 4 and 8. For responsive users, the rate threshold was set to twice the rate that would be assigned if the job were submitted now (we refer to this hereafter as the job rate) and time between retries was equal to job length.

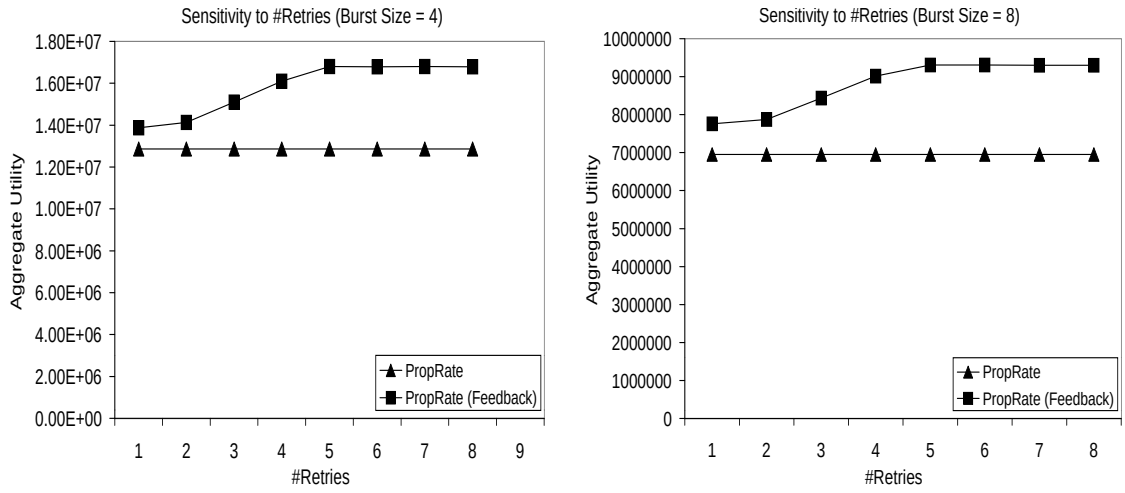

Figure 6.9: **Time-shared Performance Sensitivity to Maximum Number of Retries.** These figures plot aggregate utility as a function of the maximum number of times a user will defer. In both cases, workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$), and all users had a delay tolerance of 4. The system was fully utilized and burst sizes were exponentially distributed with mean 4 (left) and mean 8 (right). For responsive users, the rate threshold was set to twice the job rate and time between retries was equal to 1 job length.

Figure 6.9 plots the results for workloads with different degrees of burstiness. We observe that in both cases, retrying is beneficial when users are willing to defer a few times but benefits level off as users become more willing to defer. With responsive users who retry at least four times, we observe approximately 30% higher aggregate utility delivered. We also observe that after 4-5 retries, performance improvements level off. We speculate that this point corresponds to when deferring does not matter anymore since jobs have been deferred so many times that they have no value. Results also suggest that jobs that

defer are frequently being deferred the maximum number of times. This in turn suggests that during periods of contention, contention is persistent enough that users do not find it worthwhile to submit their jobs with non-zero rates.

Retry Rate Threshold

The second experiment measures performance sensitivity to retry rate thresholds and quantifies the benefits of deferring based on how sensitive users are to price feedback. We assume that retry rate thresholds are based on the rate that users would assign to a job if it were submitted at the current time. For example, if a user has a job which would be assigned a rate r, that user might defer the submission of that job if current rates were currently some multiple k of r since that would imply that submitting a job now would result in a $r/(k+1)r$ CPU allocation while being charged r units per minute. In these experiments, we use the same workload as the experiments in Section 6.2.2. For responsiveness, we set the maximum number of retries to 4 and the defer time to one job length.

Figure 6.10 plots aggregate utility as a function of retry rate thresholds for both PropRate with unadaptive users and PropRate for users who are responsive to feedback with burst sizes of 4 and 8. In both cases, we observe that performance is relatively insensitive to rate thresholds as long as the rate threshold is non-zero. (A rate threshold of 0 means we defer regardless of current rates, which is unrealistic.) This essentially means that as long as users are sensitive at all to price feedback that benefits can be obtained. Given the workload, this data suggests that load during bursts of activity is persistent enough that nearly all jobs that defer do so for the maximum length of time. The specific threshold a job uses to defer is irrelevant in this case since all jobs that defer end up deferring the maximum length of time and have 0 valuations at the end.

Retry Time

The third experiment measures performance sensitivity to retry times. Results from this experiment quantify the benefits of price feedback based on how frequently users retry their requests when they defer. For this experiment, we assume retry times are a multiple of job length since it seems reasonable that the time a user waits to retry should be somewhat related to the CPU requirements of the job being submitted. Again, we use the same workload as the experiments in Section 6.2.2. For responsiveness, we set the

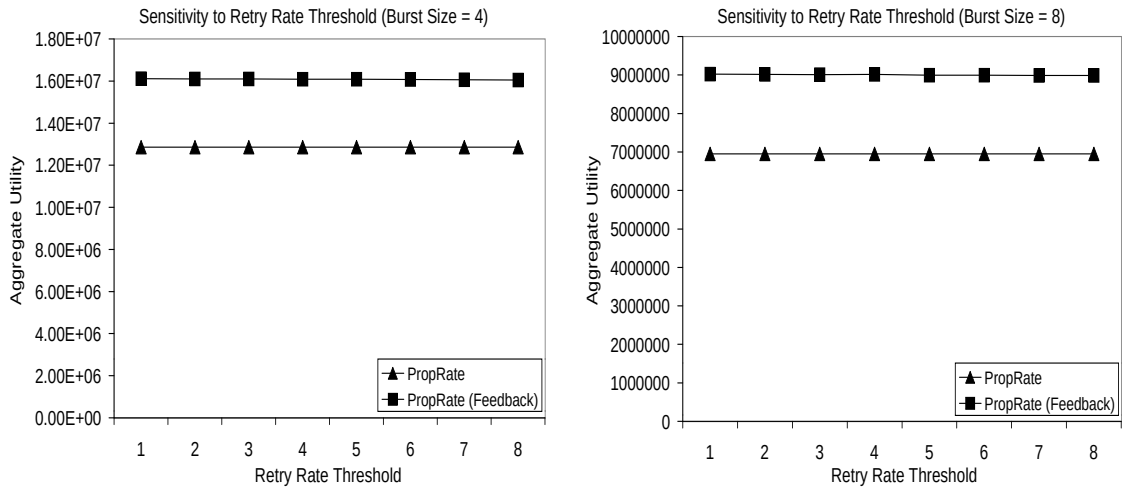

Figure 6.10: **Time-shared Performance Sensitivity to Retry Rate Threshold.** These figures plot performance as a function of retry rate thresholds. Again, workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts, and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4. The system was fully utilized with burst sizes exponentially distributed with mean 4 (left) and mean 8 (right). For responsive users, the maximum number of retries was 4 and time between retries was 1 job length.

maximum number of retries to 4 and the retry rate threshold to one job rate. Figure 6.11 plots the results.

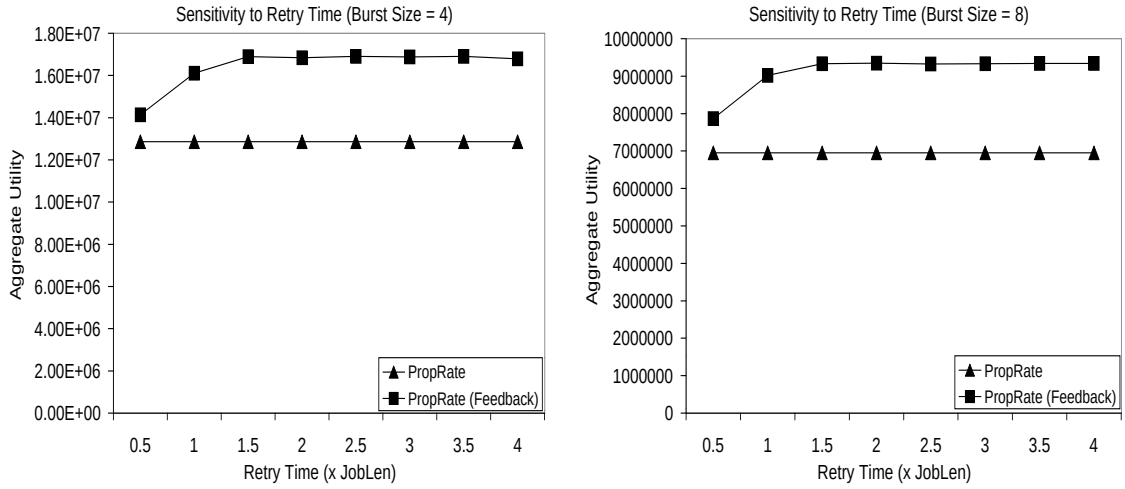

Figure 6.11: **Time-shared Performance Sensitivity to Retry Times.** These figures plot performance as a function of retry times. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts, and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4. The system was fully utilized with burst sizes exponentially distributed with mean 4 (left) and mean 8 (right). For responsive users, the maximum number of retries was 4 and the retry rate threshold was 1 job rate.

We observe that performance is relatively insensitive to retry times except at retry times less than 1.5 job lengths. This behavior is consistent with the previous two experiments which suggested that during periods of contention, contention is persistent enough that newly arriving jobs almost always defer and are never assigned a non-zero rate. With performance beginning to level off with a retry time of 1.0, we can see this since 4 maximum retries combined with a retry time of 1.0 corresponds to 4 job lengths and very low value since delay tolerance was also 4. The data also suggests, though, that contention is not so persistent that all jobs are deferred until they have 0 value. The small gap in performance between retry times of 1 and 1.5 shows that some jobs that defer do deliver non-zero utility.

Utilization

The fourth experiment measures performance sensitivity to system utilization when users defer. Results from the first three experiments suggested that during periods of contention, contention is persistent enough that deferring is a binary decision. Jobs either defer until they have 0 value, thereby letting currently running jobs deliver higher utility, or they are submitted immediately during periods of less contention. In this experiment, we investigate how the performance of a system with responsive users would fare is less loaded systems where the contention is less persistent. We assume the same workload as the first three experiments, the only difference being larger interarrival times between bursts. We use a maximum number of retries of 4, a retry rate threshold of 1 job rate, and a retry time of 1.5 job lengths. Figure 6.12 plots the results.

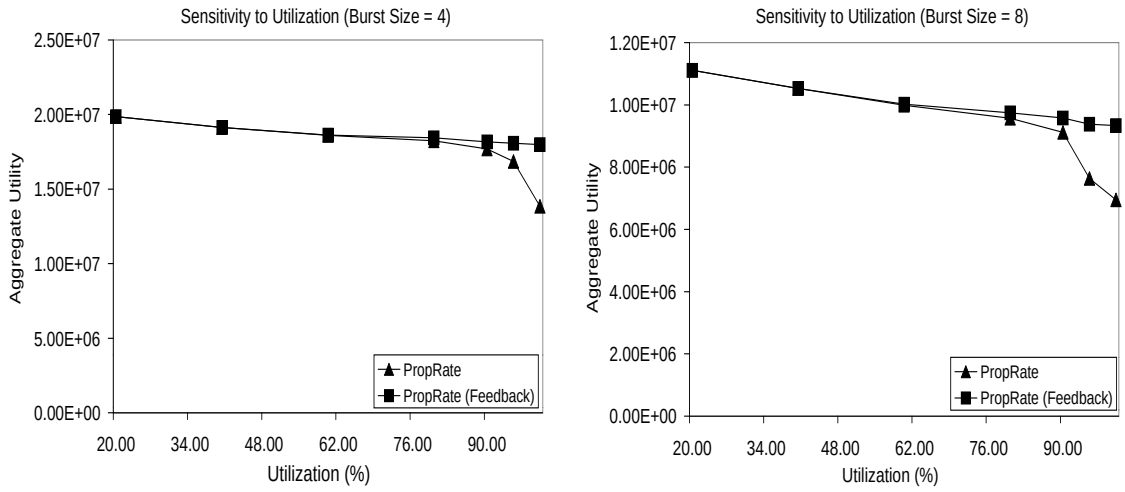

Figure 6.12: **Time-shared Performance Sensitivity to Utilization.** These figures graph aggregate utility as a function of utilization on a system with users who are responsive to price feedback. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4. The system was fully utilized with burst sizes exponentially distributed with mean 4 (left) and mean 8 (right). For responsive users, the maximum number of retries was 4, retry rate threshold was 1 job rate, and time between retries was 1.5 job lengths.

As expected, we observe that deferring is only beneficial when utilization is high. At high utilization, we see that with users who are sensitive to price feedback, up to 30% more value is delivered to users. At low utilization, we see no performance gains. The main

reason price feedback is irrelevant at low utilization is low probability that high valued jobs compete against each other. When contention occurs between multiple high valued jobs, all high valued jobs obtain large resource allocations and significantly impact each other's execution. When enough high value jobs compete, these slowdowns eventually cause all high value jobs to deliver very low value. Price feedback mitigates this by smoothing bursty workloads out over time, thereby reducing the probability that a large number of high valued jobs compete simultaneously. When rates are high, high (and low) valued jobs defer since executing now would yield a small resource allocation while being charged a high rate.

Summary

In summary, we observe that price feedback is only effective when system utilization is high. At low utilization, there is little reason to defer since the degree of competition is low. As we increase system utilization, competition for resources rises due to the increased rate of low and high valued job arrivals. Low valued job arrivals have marginal impact as we increase utilization since (i) allocating based on value causes their resource allocations to be small relatively to high value jobs and (ii) we assume slowdown is only a function of CPU sharing. On the other hand, high value job arrivals have a direct impact on the progress of other high valued jobs since they have comparable resource allocations. When burstiness causes enough high valued jobs to compete, this is where price feedback is beneficial since price feedback causes these jobs to be spread out over time.

In our simulations, we observe that price feedback results in up to 30% more value delivered to users given the workloads we simulated. It is important to note, however, that our simulations assumed that slowdowns were due solely to sharing of the CPU. It is not hard to imagine other scenarios where high degrees of multiprogramming cause other slowdowns which could factored into price feedback signals and where price feedback would be even more effective. For example, when systems are very heavily utilized, the possibility that the aggregate memory demands of competing applications cause the system to thrash become increasingly likely. When systems begin to thrash, slowdowns are dramatic since execution times are constrained by the speed of disk, which is orders of magnitude slower than physical memory. In these cases, having price feedback that takes memory into account could make a big difference in influencing user behavior to keep the system in a comfortable regime of operation.

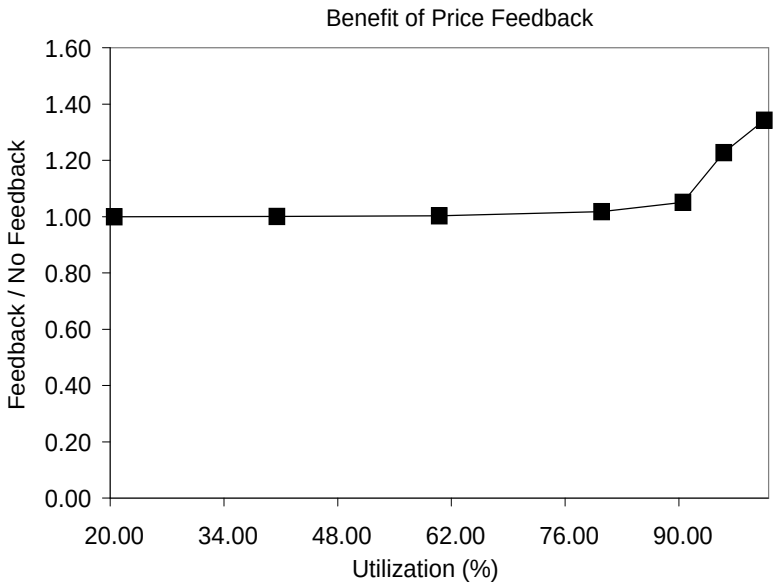

Figure 6.13: **Time-shared Benefit of Price Feedback.** This figure summarizes our results for evaluating price feedback in the time-shared computational economy. We see that price feedback makes no difference at low utilization since valuation information already provides the system with a way to focus on high valued jobs and since the probability that multiple high valued jobs compete is low. At high utilization, we observe up to a 30% improvement in performance. This improvement can be directly attributed to price feedback smoothing bursty workloads out over time and reducing the probability that multiple high valued jobs compete and slow each other down.

6.2.3 Delayed, Imperfect Information

In these experiments, we quantify the performance impact of delayed and imperfect information in the time-shared computational economy. We measure performance sensitivity to state update intervals when economic allocators use a periodic state update policy as has been the case in all experiments thus far. (We refer to a periodic update policy that sends updates every N time units as PERIODIC(N).) Next, we compare the performance of periodic state updates with two other schemes: access modules using Mitzenmacher's CHOOSE2 algorithm [60] and a state update policy that sends updates whenever prices change (i.e., whenever a job starts or ends), which we denote as PRICECHANGES. We study delayed and imperfect information because we hypothesize that timeliness of price information can have a huge impact on the effectiveness of load balancing policies based on lowest price and consequently overall value delivered to users. The results from this set of experiments underscores the importance of end-to-end system design, as opposed to exclusive focus on economic mechanisms.

State Update Times

The first experiment quantifies performance sensitivity to state update intervals when economic allocators use periodic state updates. In this experiment, each economic allocator sends periodic state updates to a bulletin board. Access modules, accessing this bulletin board, then use this information to route incoming jobs to the economic allocators offering the lowest prices. In the context of a practical implementation, the update periods in this experiment can be viewed as the time between the sending of multicast state update packets in a computational economy based on the REXEC remote execution system. This experiment quantifies one dimension of the trade-off between small update intervals for timely updates and the concomitant increase in network bandwidth required to provide those updates.

In this experiment, system utilization was 50% on average and workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrivals between bursts and normal CPU lengths. Valuations were taken from 80/20 bimodal valuation distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4. Figure 6.14 plots aggregate utility versus burst size when PERIODIC(N) price updates are used for update periods of 600, 300, 120, 60, 30, and 10.

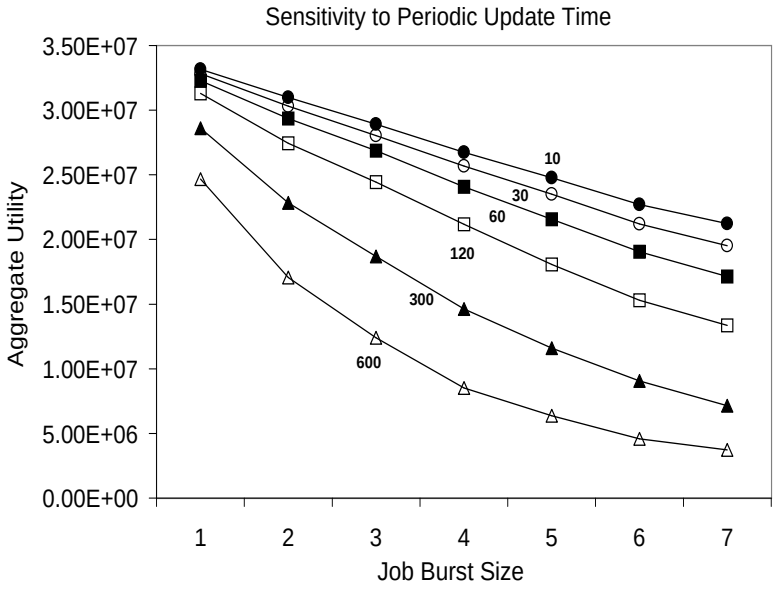

Figure 6.14: **Time-shared Performance Sensitivity to Periodic Update Time.** This figure shows sensitivity to state update times when economic allocators send state updates at intervals of 600, 300, 120, 60, 30, and 10. The system was 50% utilized. The workload consisted of 32 users submitting bursts of sequential jobs with normal interarrivals between bursts and normal CPU lengths. Valuations were taken from 80/20 bimodal valuation distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4.

We observe that for bursty workloads, infrequent updates cause significant performance degradation as burst size increases. This decrease is essentially due to bursts of jobs being routed to the same nodes due to prices not being updated quickly enough. As expected, we also observe that as update times become very frequent, performance drop-off improves considerably. It is important to note, however, that by decreasing the update period, we also increase network traffic due to the increased number of state update packets. Thus, there is a performance trade-off between having timely price information and the bandwidth required to provide fresh information using period updates.

Comparing State Update Algorithms

The second experiment compares the performance of three different state update policies for economic allocators. The first policy, PERIODIC(60), uses periodic updates with a periodic update interval of 60 as previously described. The second policy is based on Mitzenmacher's CHOOSE2 algorithm. In this policy, no state updates are sent. Instead, choosing the best node is done by randomly selecting two economic allocators and choosing the one with the lower rate. Over a wide range of system parameters, Mitzenmacher shows that CHOOSE2 results in substantially better performance compared to load balancing based on periodic load updates. The third policy, PRICECHANGES, is based on sending updates whenever prices change. Since prices only change when rates change and rates are fixed once jobs are started, this essentially means that state updates are sent whenever a job starts or ends. (In the actual implementation, state updates are also sent periodically to allow economic allocators to be initially discovered.)

Figure 6.15 graphs the results comparing the three policies for different system utilization levels. Again, we use the same workload as the first experiment. We observe that at all utilization levels, CHOOSE2 and PRICECHANGES easily outperform PERIODIC(60) whenever workloads are bursty. At utilizations greater then 80%, we observe that PRICECHANGES at least doubles the performance of PERIODIC(60). Again, this data is consistent with intuition. As burstiness increases, so does the value of having timely price information to route incoming jobs. The difference between CHOOSE2 and PRICECHANGES is small and consistent across all utilization levels. Since state change updates provide perfect information, this suggests that CHOOSE2 results in near optimal performance when the metric of performance is aggregate utility. Furthermore, since CHOOSE2 obviates the need for state change updates since state information is obtained by querying the nodes,

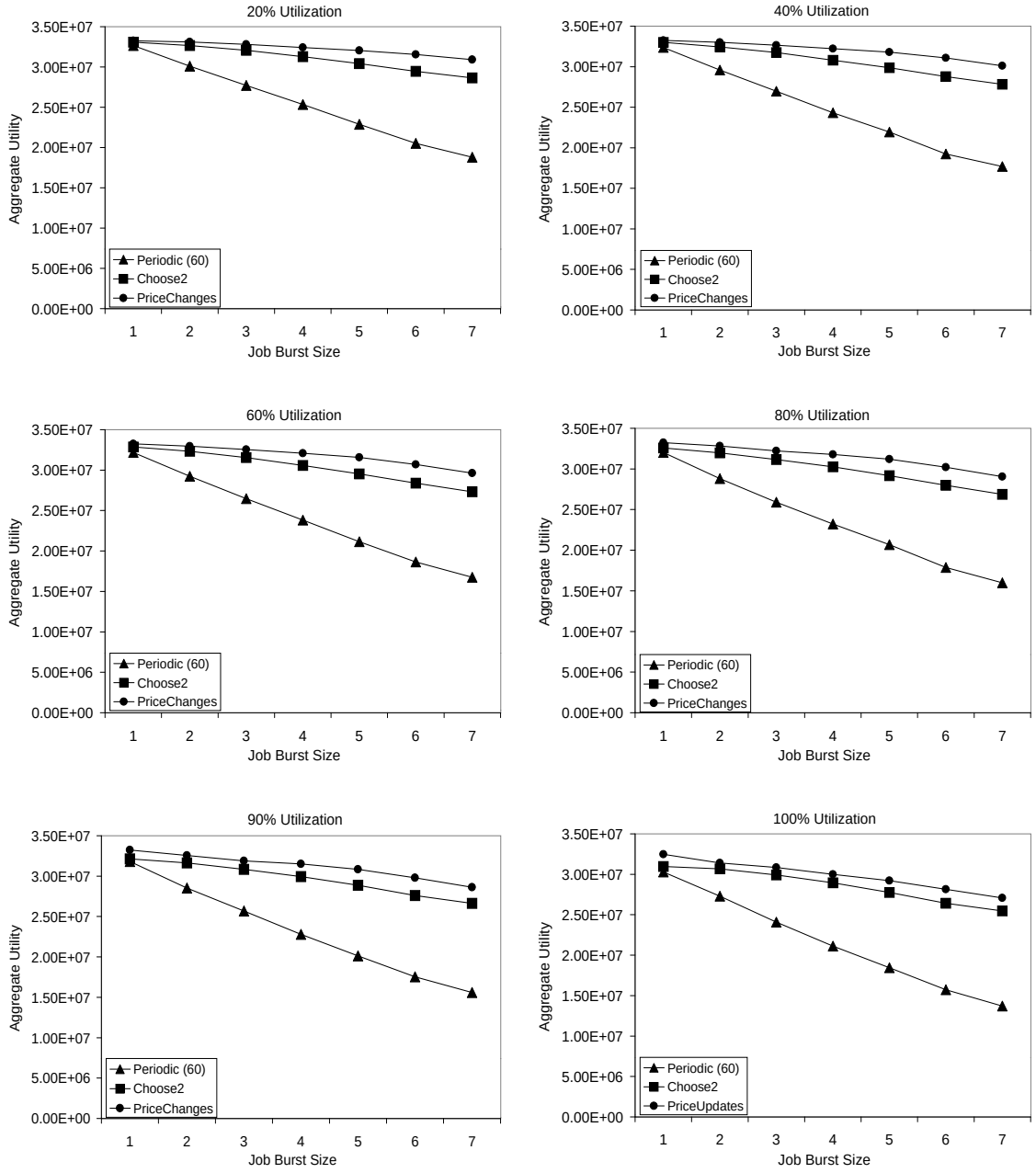

Figure 6.15: **Comparing State Update Algorithms.** These figures compare the performance of three state update algorithms: PERIODIC(60), which uses periodic state updates every 60 time units; CHOOSE2, which balancing load based on choosing two random nodes and choosing the one with the lower rate; and PRICECHANGES, which sends price changes as soon as they occur (i.e., whenever a job starts or end). We use the same workloads as Figure 6.14 and vary system utilization from 20% to 100%.

CHOOSE2 may preferable to PRICECHANGES in some systems.

In our time-shared economy, we use the PRICECHANGES policy along with longer periodic state updates rather than CHOOSE2 for two reasons. First, since prices only change when jobs start or end with our economic mechanism, the bandwidth savings of CHOOSE2 are small compared to price change updates. Second, since CHOOSE2 requires knowing the addresses of economic allocators in order to route jobs, we would still need a membership protocol if we used CHOOSE2 and thus additional bandwidth would be consumed. In the time-shared computational economy, state updates are piggybacked on membership packets. Use of CHOOSE2 would require a separate membership protocol and this protocol would obviously consume bandwidth as well.

Summary

In summary, we observe that timely price information is important to balance load effectively in market-based systems. As price information becomes increasingly stale, the value of that information decreases since it becomes less likely that the information is an accurate reflection of the current state of the system. As the interarrival times between incoming jobs decreases, the impact that this old information has on overall value delivered becomes increasingly pronounced. The impact of this is especially visible when cluster workloads are bursty with users submitting jobs in bursts as opposed to spreading them out over time. As workloads become burstier, total value delivered suffers because old information is being used to route incoming jobs that do not offer the best prices. Through a series of experiments, we have quantified this phenomena and showed that when stale information is used substantial performance losses are incurred. We also showed that by using relatively simple policies, CHOOSE2 or PRICECHANGES, that most of these losses can be avoided.

6.2.4 Strategic Users

In this experiment, we quantify the impact of strategic users when the underlying economic mechanism is not incentive-compatible. More specifically, we analyze the performance of the time-shared computational economy by varying the fraction of the user population that is strategic. As mentioned earlier, the time-shared computational economy uses a rate-based economic mechanism for CPU allocation that is simple and intuitive for users to reason about, but is not incentive-compatible. In this experiment, we quantify the

performance impact of this with strategic users that assign rates based on both personal information and the expected state of the system. The results show that aggregate utility is significantly impacted given the specific type of strategic users we consider. However, as we argue later, the likelihood of users actually behaving this way in a real system is unlikely for a number of reasons.

Balancing Value Delivered and Cost

In market-based systems, cost is an important factor in determining how users assign rates to their computations. Users must take cost into account when assigning rates since the funds in their accounts are finite and future resource demands are not completely known. Users realize that larger resource allocations are obtained with higher rates. On the other hand, they also realize that higher rates lead to higher costs, and thus they will balance their desire for larger resource allocations with the expected cost of resource consumption. This behavior of balancing resource allocations with cost can be modeled as a maximization problem where utility is defined as value delivered, based on some assigned rate, minus cost. The objective then is to find the rate that maximizes expected utility.

For this experiment, we use a highly simplified model of strategic user behavior that attempts to balance value delivered with cost. In this model, users assign rates based on personal valuations of the resources and expectations of the competition they will face. Using expectations on competition makes intuitive sense, since if users knew there would be no competition, they would simply assign a rate of ϵ regardless of their valuations. In practice, users do not know future rates. Thus, our strategy estimates future competition by assuming that current competition equals future competition. In contrast to game theoretic analyses, we do not model the behavior of other users. Instead, we only consider the aggregate behavior of all other users as captured by the simplistic assumption that future rates equal current rates.

We use this highly simplified model in order to make the analysis tractable, not because it is the optimal strategy. Computation of an optimal strategy requires the solution of a complex dynamic Bayesian game, which would further require good estimates of probability distributions on valuations, workloads, and other aspects of other user behavior to make the analysis meaningful. Because of the complexity of such an analysis and the requirements that we have good estimates of probability distributions that may not be known, we use a substantially simpler model of user behavior in order to make forward progress.

The complexity of computing the optimal solution actually bodes well for our system in practice, since if the optimal strategic behavior is difficult to compute, it is unlikely users would be capable of employing it in a real system.

The strategy assigns rates based on valuation decay and future node rates, which we assume to be equal to current rates. Valuations are assumed to be piecewise-linear (Figure 6.1) with decay segments of the form $c_1 - c_2\Delta t$, where c_2 is the valuation decay rate and Δt is execution time. Future node rates are assumed to be equal to the best current node rate, R. Given valuation decay rates and estimates on future nodes rates, a strategic user assigns a rate r that maximizes utility as given by the following equation:

$$\begin{aligned} u(r) &= \text{value} - \text{cost} \\ &= v(\Delta t) - r\Delta t \\ &= v(\text{cpuTime}\frac{R+r}{r}) - r(\text{cpuTime}\frac{R+r}{r}) \\ &= v(\text{cpuTime}(1+\frac{R}{r})) - r(\text{cpuTime}(1+\frac{R}{r})) \\ &= c_1 - \text{cpuTime}(c_2 + r)(1+\frac{R}{r}) \end{aligned}$$

where $v(\Delta t)$ is the user's time-dependent valuation and cpuTime is job CPU length. Maximizing utility, we have:

$$\begin{aligned} \frac{du(r)}{dr} &= -\text{cpuTime}(-\frac{c_2 R}{r^2} + 1) \\ r^2 &= c_2 \cdot R \\ r &= \sqrt{c_2 \cdot R} \end{aligned}$$

The utility-maximizing rate for our strategy is consistent with intuition. A strategic user is optimizing against value delivered through resource allocations and the cost to deliver that value. A large c_2 means value is decreasing quickly so a user had better assign a high rate to ensure timely execution so that high value is delivered. Similarly, a large best node rate R suggests that a user should bid higher because there is more competition and thus a higher rate r is needed to complete the job quickly. Rates are computed based on both of these factors. In addition, upper and lower bounds are imposed on optimal r's to account for the fact that we optimized based on solely on the decay segment of the valuation. We impose an upper bound based on the maximum possible rate based on true valuation and a lower bound of $\epsilon = 0.001$.

Strategic User Population

In this experiment, we examine the performance impact of having a user population that includes strategic users that employ the strategy previously described. As before, the system was fully utilized. Workloads consisted of 32 users submitting sequential jobs with normal interarrivals between bursts and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4. Job burst sizes were varied from 1 to 8. For a given burst size, we examine performance at several different utilizations and vary the composition of the user population. User populations are composed of two types of users: (i) responsive users as described in Section 6.2.2 and (ii) strategic users. We simulate populations by varying the mix of responsive and strategic users. Figure 6.16 plots the results.

We observe that as user populations become increasingly strategic that value delivered to users decreases significantly. This decrease in performance can be attributed to strategic users assigning rates lower than those based on true valuations. Lower rates lead to smaller resource allocations and subsequently longer execution times and less value delivered. This, combined with the fact that bursty workloads invalidate the assumption that future rates can be estimated as the current rates, means that performance will be strictly lower when populations include users that employ our simplistic strategy. As more users become strategic, more users hoard their funds over time as opposed to using their funds to purchase resources in order to get their computations done. This result hints at the potential performance degradation caused by strategic users, in particular ones who make imperfect assumptions about future demand.

While this experiment shows a significant performance hit when users are strategic, there are several reasons why users of a real system would be unlikely to actually employ this strategy in practice. First, as mentioned previously, the simplistic assumption that future rates equal current rates is rarely valid when workloads are bursty. Thus, an effective strategy would necessarily require using more reasonable estimates of future demand. Second, the strategy does not take into account other users employing the same strategy. Because of this, the strategy cannot be optimal since it does not attempt to exploit the excessively low rates other users are assigning. Finally, because funds in our system have no intrinsic value and are only useful in exchange for computational resources on a single cluster, the saving of funds only makes sense if we eventually make use of those funds. A strategy which, in steady-state, builds a continuously increasing reserve of funds is not

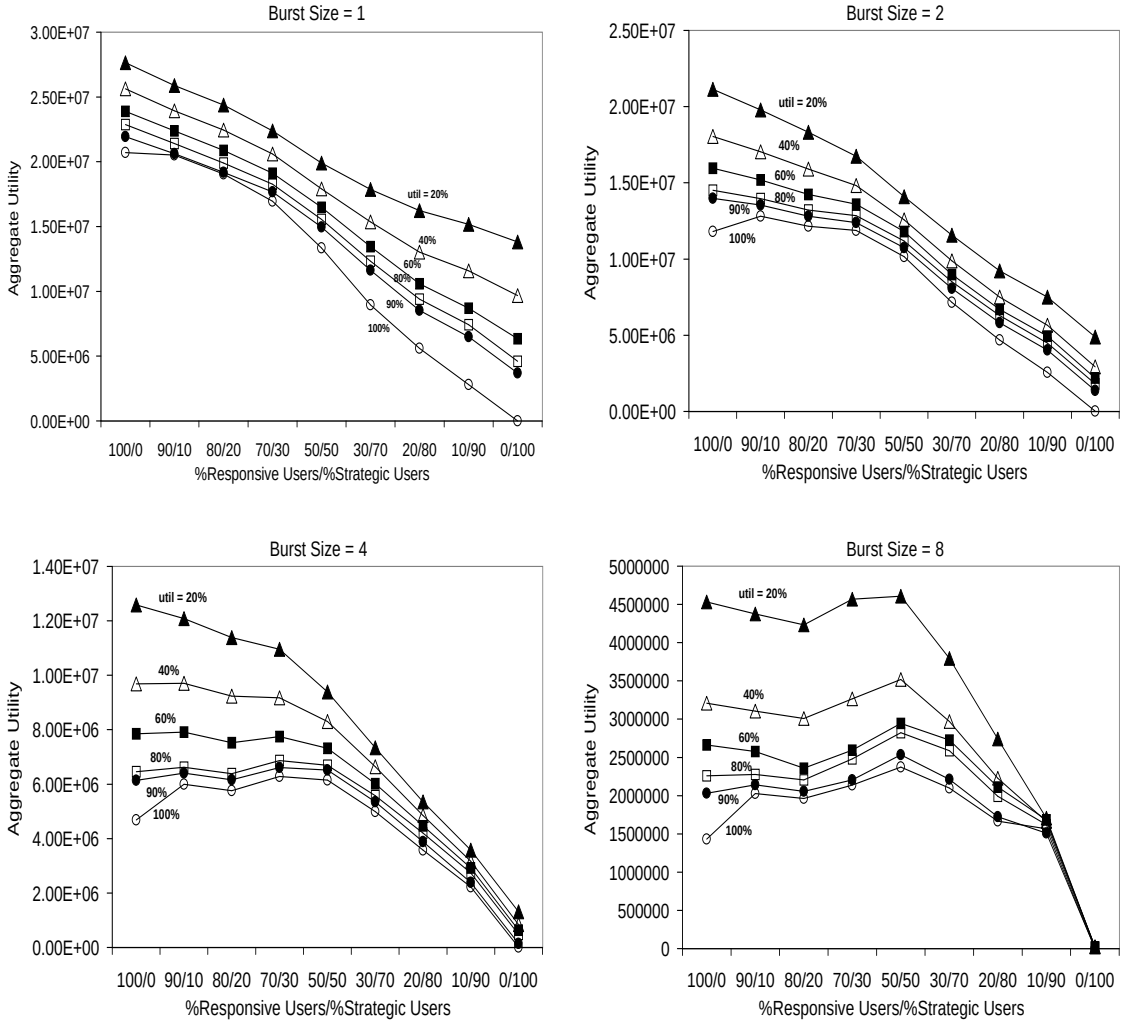

Figure 6.16: **Quantifying the Impact of Strategic Users.** These figures plot aggregate utility as a function of the composition of the user population. User populations are composed of two types of users: (i) responsive users who respond to price feedback as described earlier and (ii) strategic users as just described. Each experiment measures the aggregate utility of the system as we increase the fraction of users that are strategic. The system was fully utilized. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrivals between bursts and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 4.

realistic. Furthermore, such behavior can easily be engineered away by simply placing a bound on the maximum amount of funds a user can save as described in [82].

Summary

In summary, we observe that the performance impact of strategic users can be significant for our simplistic model of strategic user behavior. Strategic behavior necessarily implies that strategic users will assign rates lower than those based on their true valuations. The underlying motivation for such behavior is essentially lower costs and the saving of funds which might be applied to future computations. However, the strategy described in this section is not a good example of a realistic strategy that users would employ in our time-shared computational economy for several reasons. First, the simplistic assumption that future rates equal current rates is rarely valid when workloads are bursty. Second, the strategy does not take into account other users employing the same strategy. Finally, a strategy whose steady-state behavior is to continuously build an increasing reserve of funds over time is not realistic. Furthermore, by placing a bound on the maximum amount of funds a user can save, the benefits of being strategic can be diminished, thereby encouraging users to assign rates closer to their true valuations.

6.3 Batch Simulations

In this section, we analyze the performance of a market-based batch queue system in terms of aggregate utility while comparing it to three traditional algorithms. We test the hypothesis that by allowing users to express resource valuations that significantly more value can be delivered to users. We test it by performing a series of experiments which measure performance sensitivity to valuation distributions and workload parameters including valuation decay rates, variation in valuations, burstiness, utilization, and parallelism in the workload. With the base performance established, we then extend these results by examining what performance gains can be achieved through use of preemption. Since scheduling decisions are not always optimal due to lack of future information, these experiments quantify the performance gains of being able to correct short-term scheduling decisions which may no longer be optimal.

We analyze four different algorithms for these experiments. Each algorithm imposes an ordering on how queued jobs are scheduled onto the cluster. For FirstPrice,

the ordering is based on user-assigned, fine-grain, time-varying bids which specify value per unit time. For FIFO, the ordering is based on smaller arrival times. For SJF, it is based on smaller job CPU lengths. Finally, for PRIOFIFO, it is based first on priority and second on smaller arrival times. With PRIOFIFO, users have their choice of submitting jobs to three FIFO queues each with a different priority and pricing [2]. The next job to be scheduled is always the earliest job in the highest priority queue. For all four algorithms, the model we use is that the batch queue system runs the scheduling algorithm at periodic intervals. At each interval, as many jobs as possible are scheduled onto available resources.

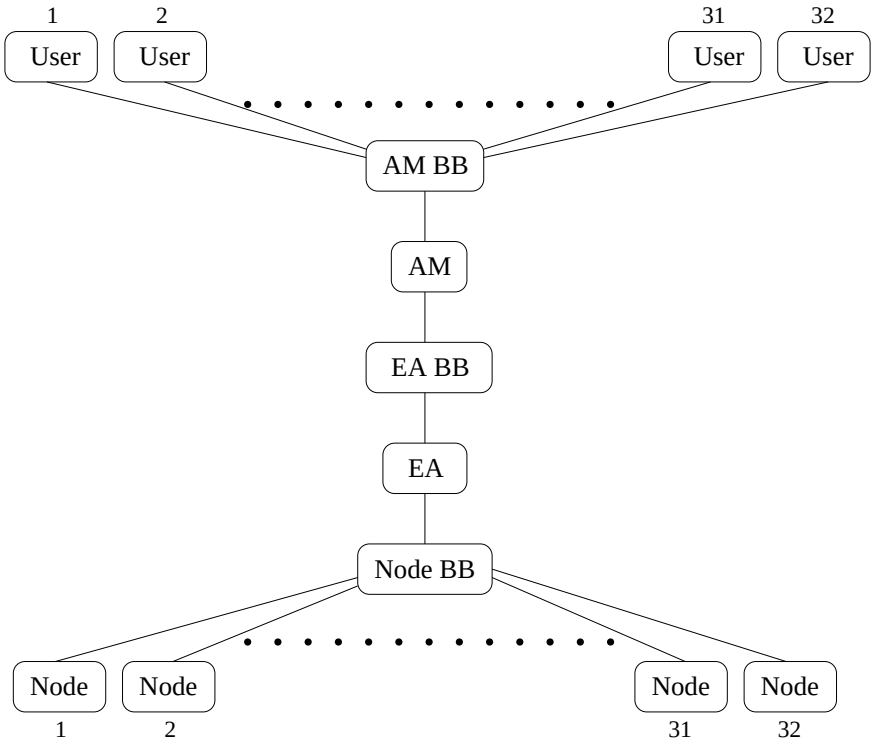

Figure 6.17: **System Configuration for Batch Simulations.** The system has 32 users generating workloads which are parameterized based on the specific experiment being performed. It uses centralized AM, EA, and node bulletin boards and a single access module. Resources are managed using a centralized economic allocator that implements the specific batch queue algorithm. The economic allocator manages resources on 32 cluster nodes. The configuration is consistent with both the Anemone batch queue system and traditional batch queue systems.

[2] We optimistically assume that prices are appropriately set so that low valued jobs map into the lowest priority queue and high valued jobs map into the highest priority queue. Assuming valuation distributions do not change much over time, such prices might be set based on empirical observations of usage of the different queues.

For all four scheduling algorithms, the same backfill algorithm is used. Let n be the number of nodes needed by the job at the head of the queue, and let n_{free} be the number of nodes currently free. Backfilling is attempted whenever $n_{free} < n$ (i.e., not enough nodes are available for the most eligible job). Given that job lengths are known, we can compute the time T before another $n - n_{free}$ nodes become free. The backfill algorithm then attempts to fill the n_{free} CPU slots of length T on idle nodes with useful computation. Logically, it does this by considering all other jobs in the queue with length less than T, then by backfilling the empty slots by first considering the second most eligible job and moving towards the back of the queue.The algorithm stops when either the idle space has been completely populated or there are no more candidate jobs to consider. Our algorithm is effective, but not overly aggressive. The most eligible job never waits any longer than it would have without backfilling.

In all experiments, we use the system configuration shown in Figure 6.17. Like the time-shared simulations, we again have 32 users along with centralized AM and EA bulletin boards. The key difference with the batch simulations is that resources are managed by a centralized economic allocator which is responsible for resource allocation on all 32 nodes of the cluster. This organization essentially mirrors that of both the Anemone batch queue system and traditional batch queue systems which use a centralized daemon for scheduling queued jobs onto cluster resources. As before, each user had a workload consisting of 1000 jobs which were parameterized based on the specific experiment being performed. Each simulation ran all user workloads to completion and measured overall system performance.

6.3.1 Assessing the Benefits of Valuation Information

In these experiments, we test the hypothesis that by providing users with a way to express resource valuations significantly more value can be delivered to users. We do this by first examining performance sensitivity to variations in resource valuations with realistic valuation distributions. We then investigate how user tolerance to delay impacts system performance. Next, we examine performance sensitivity to burstiness in the workload when job arrival processes involve bursts of jobs as opposed to individual jobs. A sensitivity experiment for system utilization with a realistic cluster workload follows. Finally, in the last experiment, we examine performance sensitivity to parallelism in the workload. Using results from these experiments, we demonstrate under what types of workloads and in which regimes of operation market-based systems make the most difference for batch environments.

Variation in Resource Valuations

The first experiment examines sensitivity to variations in user resource valuations. The intent of this experiment is identical to that of the time-shared experiment described earlier except for ordering in the queue as opposed to allocations on nodes. For this experiment, workloads consisted of 32 users submitting sequential job bursts with normal interarrivals and normal burst sizes (mean = 16). The system was fully utilized and all users had a delay tolerance of 8. As before, we assume bimodal distributions on resource valuations where x% of the valuations come from a low normal distribution and 100 - x% of the valuations come from a high normal distribution. Again, we looked at two bimodal distributions, one with $\mu_{low} = 50$ and $\mu_{high} = 500$, the other with $\mu_{low} = 50$ and $\mu_{high} = 5000$. Figure 6.18 shows aggregate utility obtained with constant CPU lengths while Figure 6.19 shows the results with normal and exponential CPU lengths.

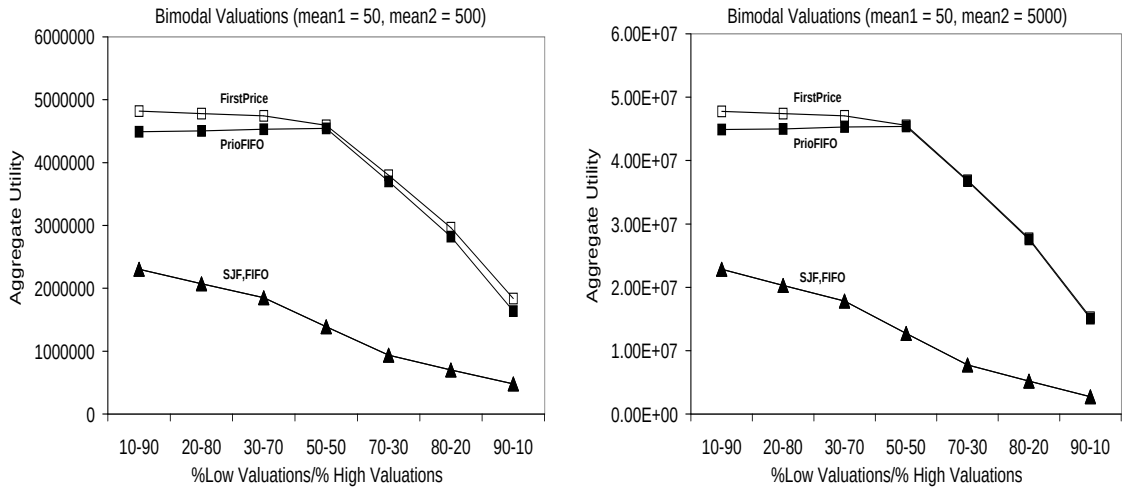

Figure 6.18: **Batch Performance Sensitivity to Variation in Resource Valuations.** These figures plot aggregate utility as a function of resource valuation distributions. For each set of results, resource valuations were bimodal with x% of the valuations coming from a normal distribution with mean 50 (low valuations) and 100 - x% of the valuations coming from a normal distribution with means of 500 and 5000 (high valuations). The system was fully utilized. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts and normal burst sizes (mean = 16). All users had a delay tolerance of 8. Constant CPU length jobs were used.

We observe that the market-based algorithm outperforms all three traditional algorithms in all cases. Again, the key to this result is valuation information. Since FIRSTPRICE

uses fine-grain, time-varying bids which reflect value to order jobs in the queue, resources are consistently being focused on the jobs that are delivering the most value to users. As queues build up due to burstiness in the workload, the value of this ordering increases since valuations are time-dependent. When queues build up, the key question is how algorithms allow high valued jobs to obtain scheduling priority in the queue. For FIRSTPRICE, this prioritization is perfectly aligned with value since ordering is done based on bids which are assumed to reflect true valuations. On the other hand, with traditional algorithms, prioritization is either non-existent (FIFO), unintentional (SJF), or imperfect (PRIOFIFO).

We can see the effect of these different types of prioritization in the queue by looking at the results in Figures 6.18 and 6.19. For constant CPU length jobs, we see that both FIRSTPRICE and PRIOFIFO perform substantially better than both FIFO and SJF. As expected, FIFO and SJF deliver lower performance since low valued work is frequently given priority over higher valued work farther back in the queue. FIRSTPRICE, on the other hand, performs significantly better since prioritization is based on fine-grain, time-varying bids which reflect value. PRIOFIFO, surprisingly, also performs well for all job mixes, including those with only small percentages of low valued jobs. This result is due to the fact that when bursts of low valued jobs queue in front of bursts of high valued jobs, all the high valued jobs suffer. Since the system was fully utilized and bursty, the aggregate result of this suffering can be significant.

For normal and exponential CPU length jobs (Figure 6.19), the results are slightly different. Again, FIRSTPRICE performs universally better than all other algorithms since it schedules jobs based on value. On the other hand, both PRIOFIFO and SJF show significantly different behavior. For exponential CPU lengths, we observe that PRIOFIFO delivers substantially lower performance compared with constant and normal CPU lengths. This result can be attributed to lowered effectiveness of high priority queues due to higher variance in CPU lengths and the accompanying, occasional long queueing delays. In addition, for both normal and exponential CPU lengths, we see that SJF performs substantially better than with constant CPU lengths. This result can be attributed to SJF unintentionally providing shorter, high valued jobs a way to bypass longer, low valued jobs in the queue. The effectiveness of this clearly decreases as more low valued jobs enter into the mix.

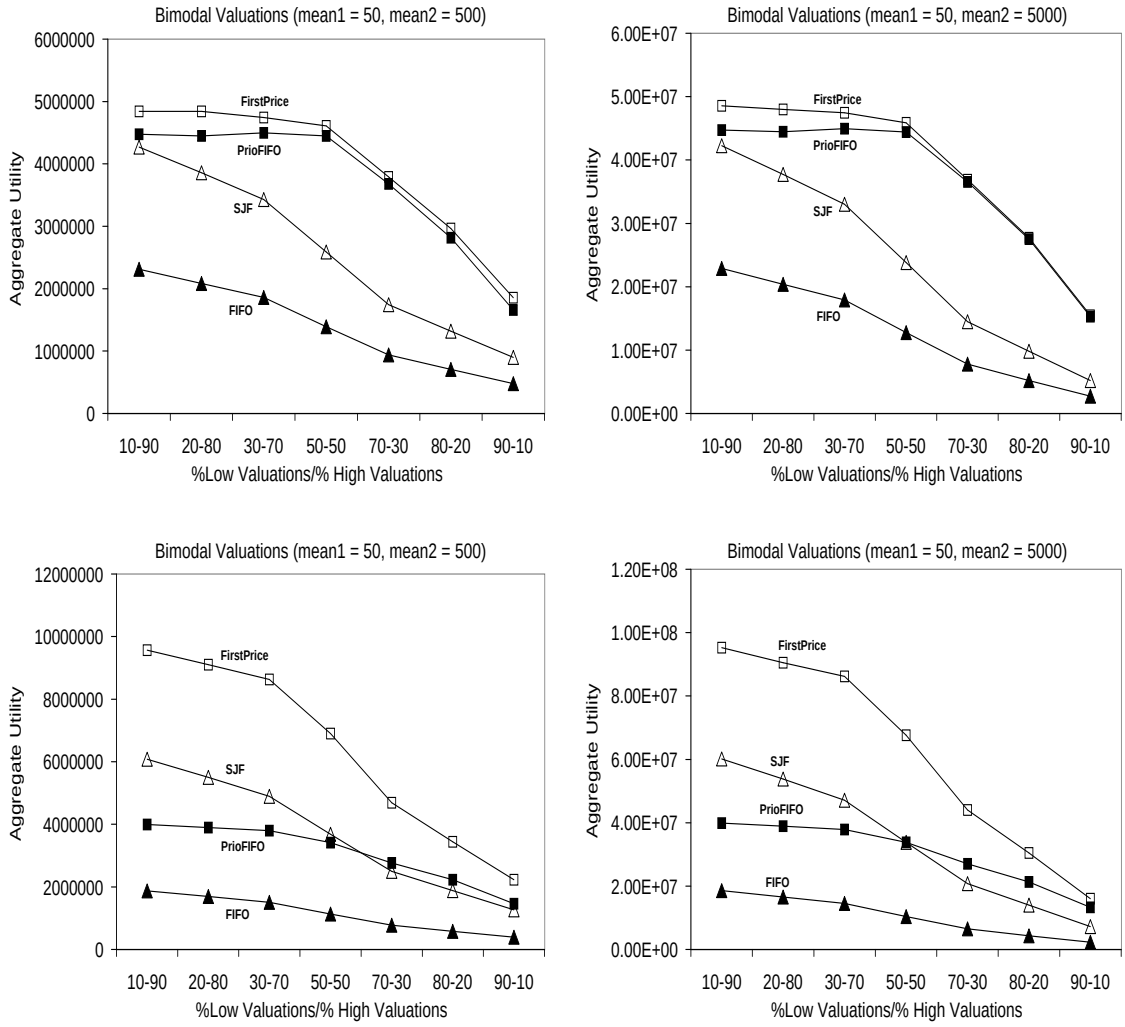

Figure 6.19: **Batch Performance Sensitivity to Variation in Resource Valuations (Normal and Exponential CPU Length Jobs)** These figures plot aggregate utility as a function of resource valuation distributions. For each set of results, resource valuations were bimodal with x% of the valuations coming from a normal distribution with mean 50 (low valuations) and 100 - x% of the valuations coming from a normal distribution with means of 500 and 5000 (high valuations). The system was fully utilized. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts and normal burst sizes (mean = 16). All users had a delay tolerance of 8. Normal (top) and exponential CPU length jobs were used.

User Tolerance to Delay

The second experiment examines sensitivity to user tolerance to delay. Again, the intent of this experiment is the same as in the time-shared case described earlier except for delays in the queue as well as delays due to execution. In this experiment, workloads consisted of 32 users submitting sequential job bursts with normal interarrivals, normal burst sizes (mean = 16), and normal CPU lengths. Valuations were bimodally distributed with 80% of the workload consisting of low valued jobs and 20% of the workload consisting of high valued jobs. As before, low valuations were centered around a normal distribution with mean $\mu_{low} = 50$ and high valuations were centered around a normal distribution with mean $\mu_{high} = 5000$. Figure 6.20 plots the results.

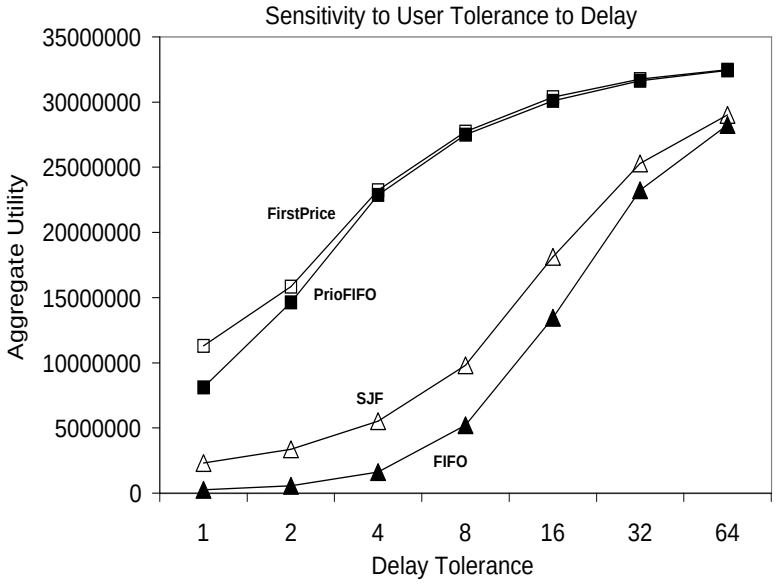

Figure 6.20: **Batch Sensitivity to User Tolerance to Delay.** This figure plots aggregate utility as a function of user tolerance to delay. The system was fully utilized. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts, normal burst sizes (mean = 16), and normal CPU lengths. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$).

We observe that the benefit of knowing valuations decreases as users become more tolerant to delay but that benefits are still significant even when users are moderately tolerant of delays. As before, we observe that both FirstPrice and PrioFifo outperform Sjf and Fifo by large margins and that FirstPrice and PrioFifo deliver comparable

performance. We observe that when users are highly sensitive to delay that PrioFifo's imperfect prioritization is still effective. This result can be attributed to, on average, four times less work in the high priority queue than the low priority queue due to the specific distribution on valuations we used. As expected, we also observe that as users become more tolerant to delay that performance of all four algorithms converges since prioritization based on value becomes less important.

Job Burst Size

The third experiment examines sensitivity to burstiness in the workload as a function of job burst size. Again, the intent of this experiment is the same as in the time-shared case described earlier except for competition in the queue as opposed to competition on nodes. In this experiment, workloads consisted of 32 users submitting sequential job bursts with normal interarrivals, normal CPU lengths, and a delay tolerance of 8. Valuations were bimodally distributed with 80% of the workload consisting of low valued jobs and 20% of the workload consisting of high valued jobs. As before, low valuations were centered around a normal distribution with mean $\mu_{low} = 50$ and high valuations were centered around a normal distribution with mean $\mu_{high} = 5000$. Figure 6.21 plots the results.

We observe that the benefit of knowing resource valuations is small when workloads are non-bursty, increases with burstiness, and finally decreases as workloads become so bursty that queue lengths are so long that all algorithms suffer. When workloads are non-bursty, the results are consistent with intuition since a single, newly arriving job with any algorithm need not wait long for a single node to become free since job CPU lengths are comparable and there are 32 nodes in the cluster. As burstiness increases, queues begin to form and thus both FirstPrice and PrioFifo begin to outperform Fifo and Sjf since high valued jobs see shorter queueing delays. Finally, as workloads become highly bursty, all algorithms suffer since queues are so long that even high valued jobs face long queueing delays.

Utilization

The fourth experiment examines sensitivity to utilization with a realistic batch workload. In this experiment, workloads consisted of 32 users submitting sequential job bursts with normal interarrivals, normal burst sizes ($\mu = 16$, $\sigma^2 = 9$), normal CPU lengths, and a delay tolerance of 8. Valuations were bimodally distributed with 80% of the workload

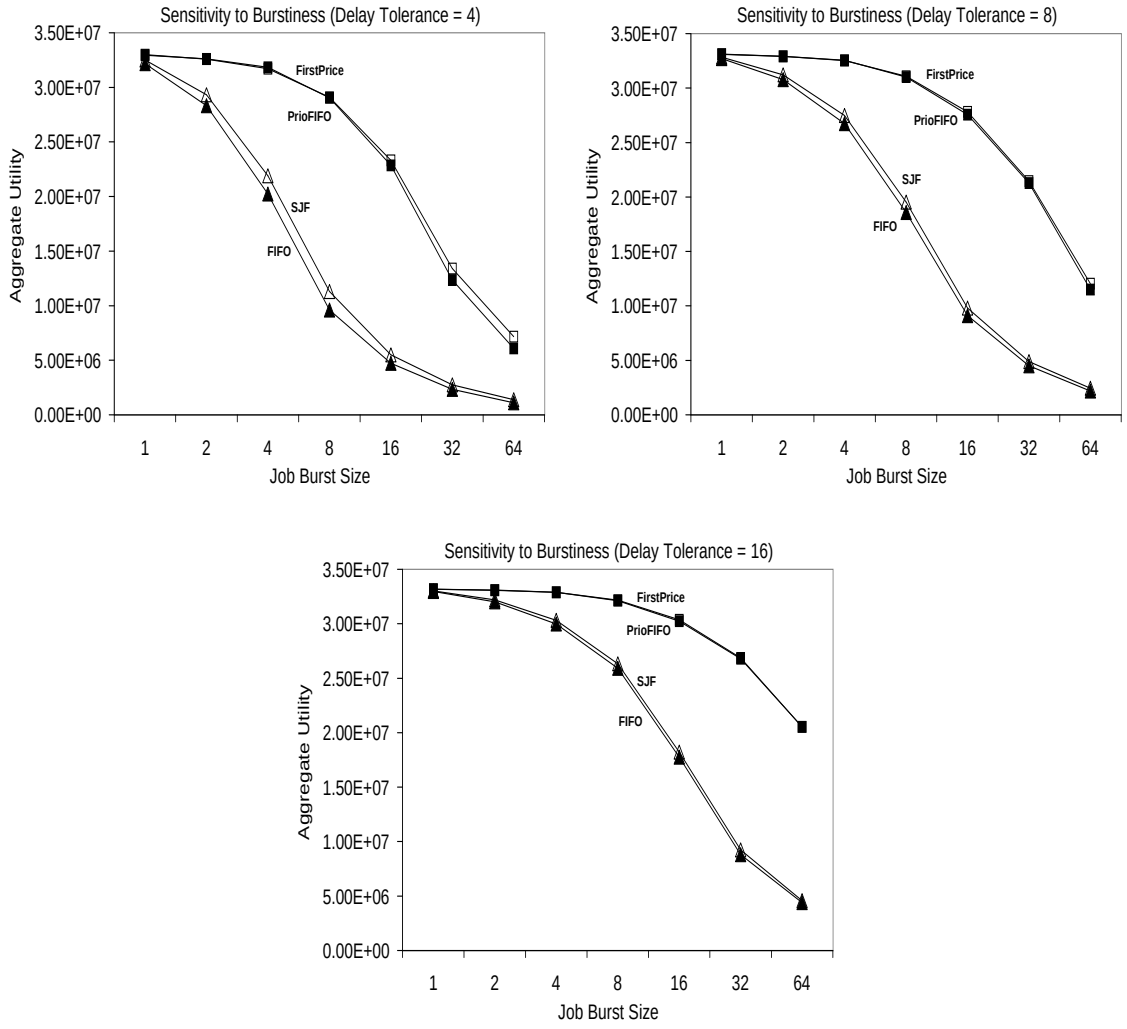

Figure 6.21: **Batch Sensitivity to Burstiness in the Workload.** These figures plot aggregate utility as a function of job burst size for several different user sensitivities to delay. The system was fully utilized. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts and normal CPU length jobs. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 8.

consisting of low valued jobs and 20% of the workload consisting of high valued jobs. As before, low valuations were centered around a normal distribution with mean $\mu_{low} = 50$ and high valuations were centered around a normal distribution with mean $\mu_{high} = 5000$. Figure 6.22 plots the results.

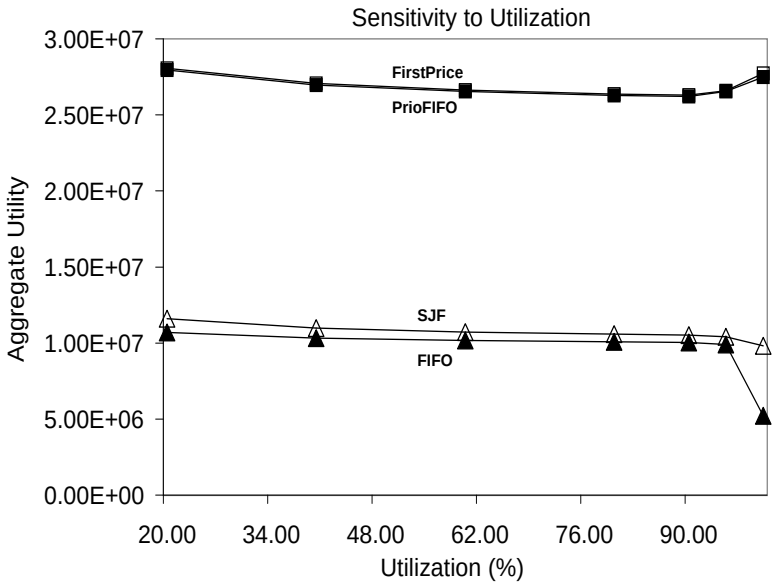

Figure 6.22: **Batch Performance Sensitivity to Utilization.** This figure plots aggregate utility as a function of batch system utilization. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts, normal burst sizes (mean = 16), and normal CPU length jobs. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a tolerance to delay of 8.

We observe that the benefit of knowing valuations is steady and significant at all utilizations simulated. We observe that both FirstPrice and PrioFifo outperform Sjf and Fifo by large margins and that FirstPrice and PrioFifo deliver comparable performance. We see approximately 150% higher value delivered at virtually all levels of utilization when valuations are known as opposed to when they are not known. Performance gaps are steady as opposed to increasing with utilization (as in the time-shared case) due to admission control. Again, we also see that for these sequential workloads, PrioFifo and its imperfect prioritization based on value is just as effective as fine-grain valuations with FirstPrice.

Parallel Degree

The fifth experiment examines sensitivity to parallelism in the workload by quantifying the effects of head-of-line blocking. Because parallelism in workloads rarely maps perfectly onto resources, there will be times when the most eligible parallel job has to wait for resources to become available since only a subset of the required nodes may be free. This waiting increases queueing delay and consequently decreases value delivered for both the blocked job and all jobs behind it in the queue that cannot be backfilled onto idle nodes. This experiment quantifies this phenomena by simulating a workload with 32 users submitting bursts of parallel jobs with normal interarrivals, normal burst sizes, normal CPU lengths, and varying levels of parallel degree. Valuations were 80/20 bimodally distributed ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 8.

Figure 6.23 plots the results when parallel degree is uniformly distributed for varying levels of utilization. (We do not show the result for 100% utilization since that simulation is unstable [3] if backfilling is imperfect, as it almost always is.) Again, we observe that FirstPrice and PrioFifo outperform Fifo and Sjf by significant margins. More interestingly, we observe that as parallelism in the workload increases, the performance of PrioFifo begins to decay more rapidly than FirstPrice. This result can be attributed to head-of-line blocking. As there is more parallelism in the workload, the probability that the most eligible job has to wait for all its required nodes to become available increases. This blocking increases queueing delay in the high priority queue and consequently reduces its effectiveness since queues are longer and processing is FIFO. With even modest amounts of parallelism, we see that FirstPrice substantially outperforms PrioFifo.

Figure 6.24 plots the results when parallel degree is taken from four different distributions: a sequential distribution (parallel degree = 1), and normal, uniform, and exponential distributions with a mean parallel degree of 8. We observe that as the variance in the parallel degree distributions increases, so does the benefit of scheduling based on time-varying bids in FirstPrice. Just as we saw with exponential CPU lengths in the sequential case and uniform parallel degree in the parallel case, larger queueing delays in the high priority queue reduces its effectiveness for scheduling based on value. The longer

[3] At 100% utilization, the simulator produces enough load in steady state so that all nodes are 100% utilized. However, for parallel workloads, 100% utilization may not always be possible since parallel jobs may need to wait for all their nodes to become available, and thus nodes will occassionally be idle. Backfilling can help mitigate this, but generally is not able to completely eliminate the idle time. Over long simulations, idle time combined with load generation for 100% utilization eventually causes queues to grow without bound.

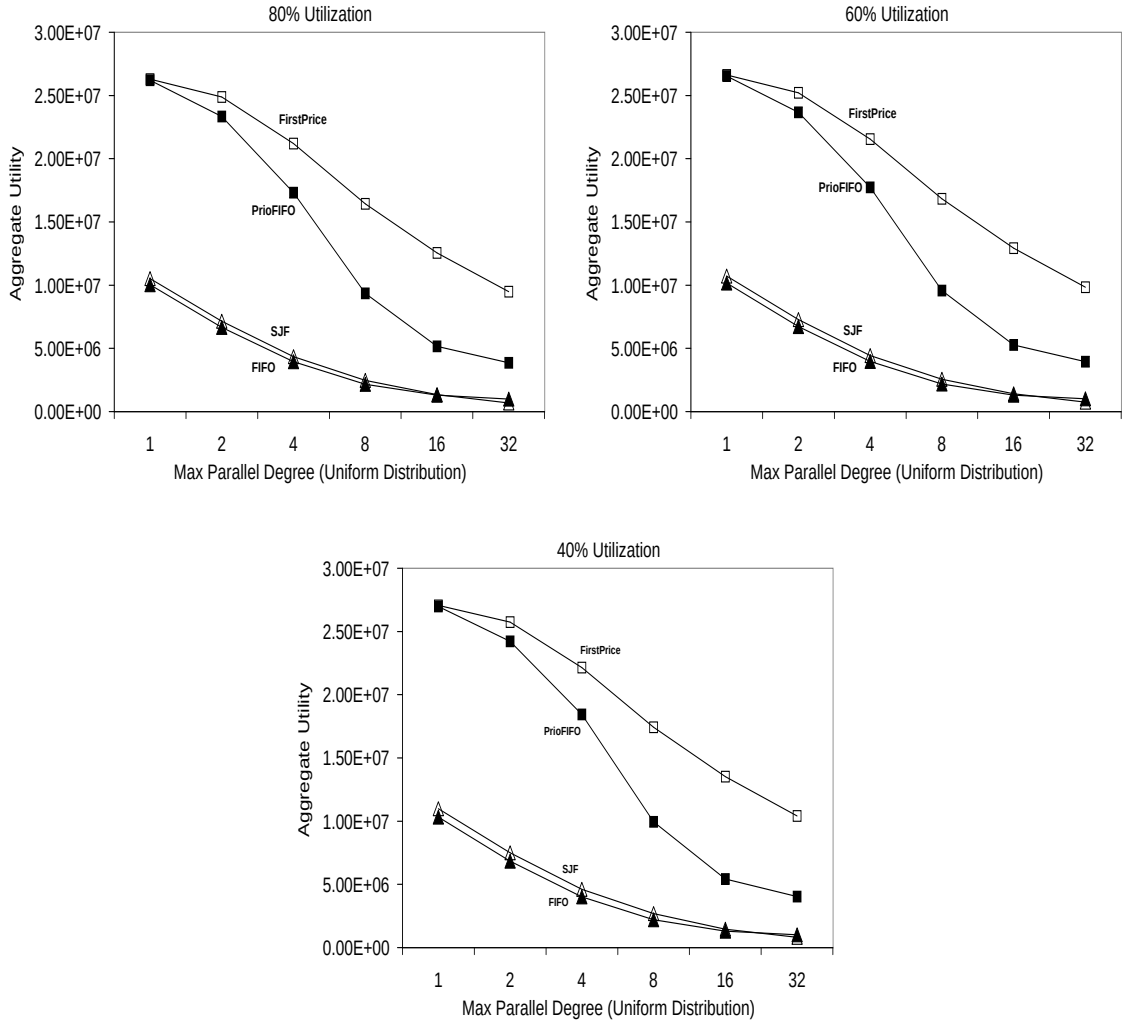

Figure 6.23: **Batch Sensitivity to Parallelism in the Workload.** These figures plot aggregate utility as a function of maximum parallel degree when parallel degree is taken from a uniform distribution. Performance is shown at three different system utilizations. Workloads consisted of 32 users submitting bursts of parallel jobs with normal interarrival times between bursts, normal burst sizes (mean = 16), and normal CPU length jobs. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 8.

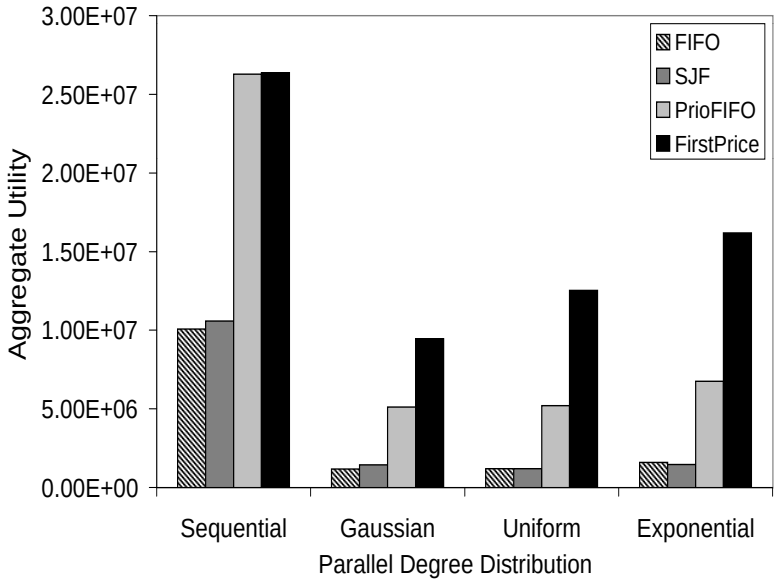

Figure 6.24: **Batch Performance Sensitivity to Parallel Degree Distribution.** This figure graphs aggregate utility when parallel degree is taken from four different distributions: a sequential distribution (parallel degree = 1), and normal, uniform, and exponential distributions with a mean parallel degree of 8. Workloads consisted of 32 users submitting bursts of parallel jobs with normal interarrival times between bursts, normal burst sizes (mean = 16), and normal CPU length jobs. Valuations were taken from 80/20 bimodal valuation distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$), and all users had a delay tolerance of 8.

the high priority queue becomes, the greater the chance that the most eligible job is not the job at the head of the queue. Since processing of individual queues in PrioFifo is FIFO, this ultimately leads to lower value delivered to users. In contrast, FirstPrice always schedules the highest value job.

Summary

In summary, we observe that knowing resource valuations leads to significantly higher performance (Figure 6.25) in batch environments. We observe that these gains in performance are especially large when workloads are bursty, which is usually the case since users submit jobs in batch by definition. We also observe that these benefits increase with variation in valuations of the underlying computational resources. We also showed that when workloads are bursty, large benefits are delivered even at low utilization and when users are fairly tolerant to delay. Finally, we also showed that as parallelism in the workload increases, so the does the benefit of scheduling based on valuations.

We observe that having fine-grain, time-varying resource valuations as opposed to coarse-grain, constant valuations results in negligible benefits for sequential workloads and significant benefits for parallel workloads (Figure 6.26). For sequential workloads, we observe negligible benefits over a wide range of workload burstiness levels, utilizations, and user tolerance to delay. The only benefits observed were for valuation distributions where all jobs have nearly the same value, in which case valuations tend to map to the same priority leading to near-FIFO performance for PrioFifo, and when users are especially sensitive to delay. On the other hand, for parallel workloads, we observed that having time-varying valuations can lead to significant gains in performance with benefits increasing with the degree of parallelism in the workload.

6.3.2 Evaluating Preemption

In this section, we evaluate preemption as a means to correct short-term scheduling decisions in light of new information. We do so because we anticipate batch systems can benefit by preempting low valued jobs to run high valued jobs which had not arrived when the low valued jobs were originally scheduled. To evaluate preemption, we assume all algorithms have some means of determining whether it is "better" to preempt a running job to run another job. We assume each algorithm has a metric which is used to evaluate how well the algorithm schedules each job (e.g., utility delivered). We also assume that algo-

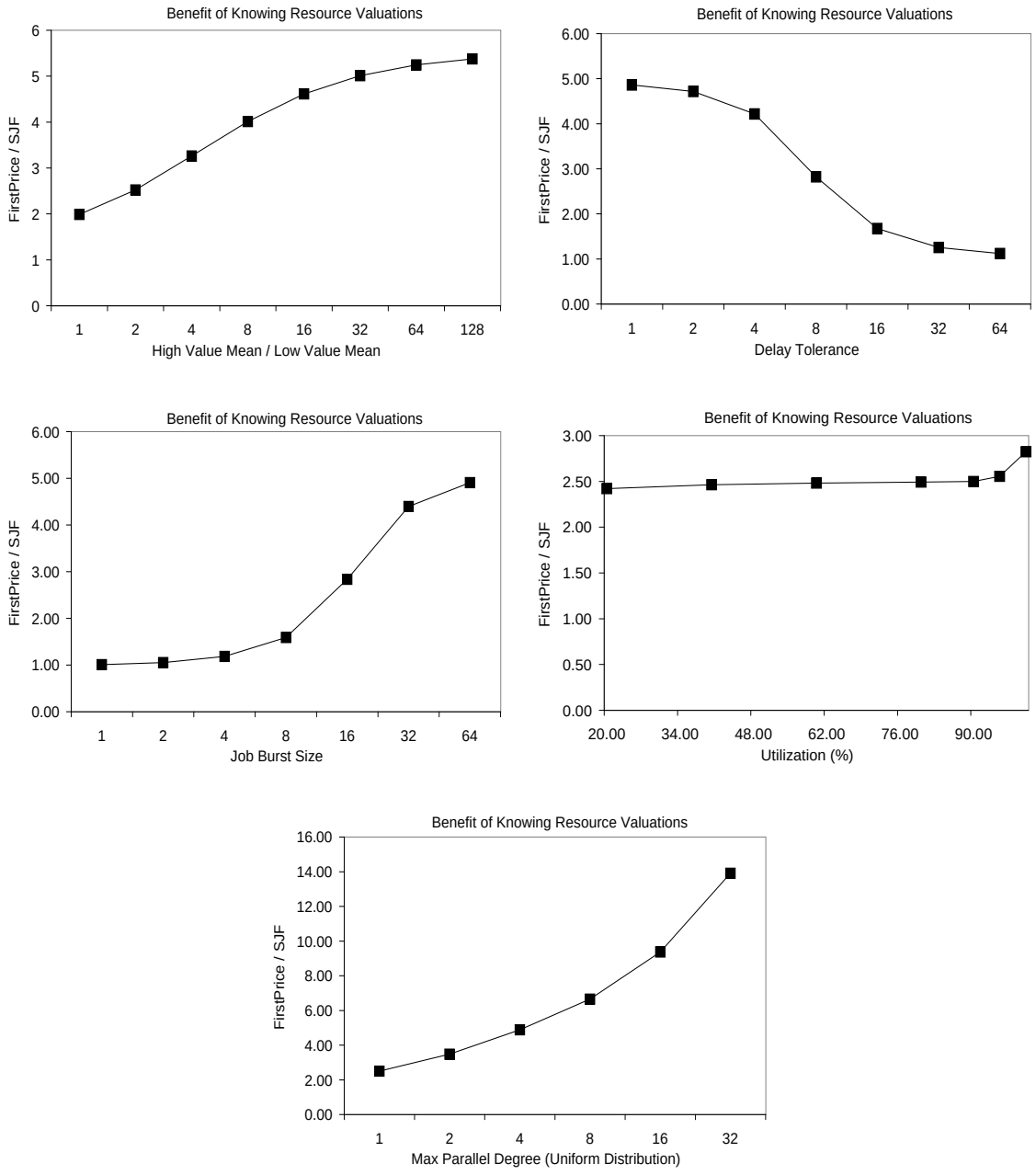

Figure 6.25: **Benefit of Knowing Resource Valuations.** These figures summarize our results which quantify the value of knowing resource valuations. We observe that the benefits of knowing valuations are most pronounced when workloads are bursty and when there is significant variation in resource valuations. In addition, we also see that as users become more sensitive to delay that the benefit of scheduling based on valuations increases. For moderately bursty workloads, significant benefits are obtained over a wide range of utilization levels due to competition in the queue. Finally, we see that as workloads have more parallelism, the benefits of knowing valuations increases significantly.

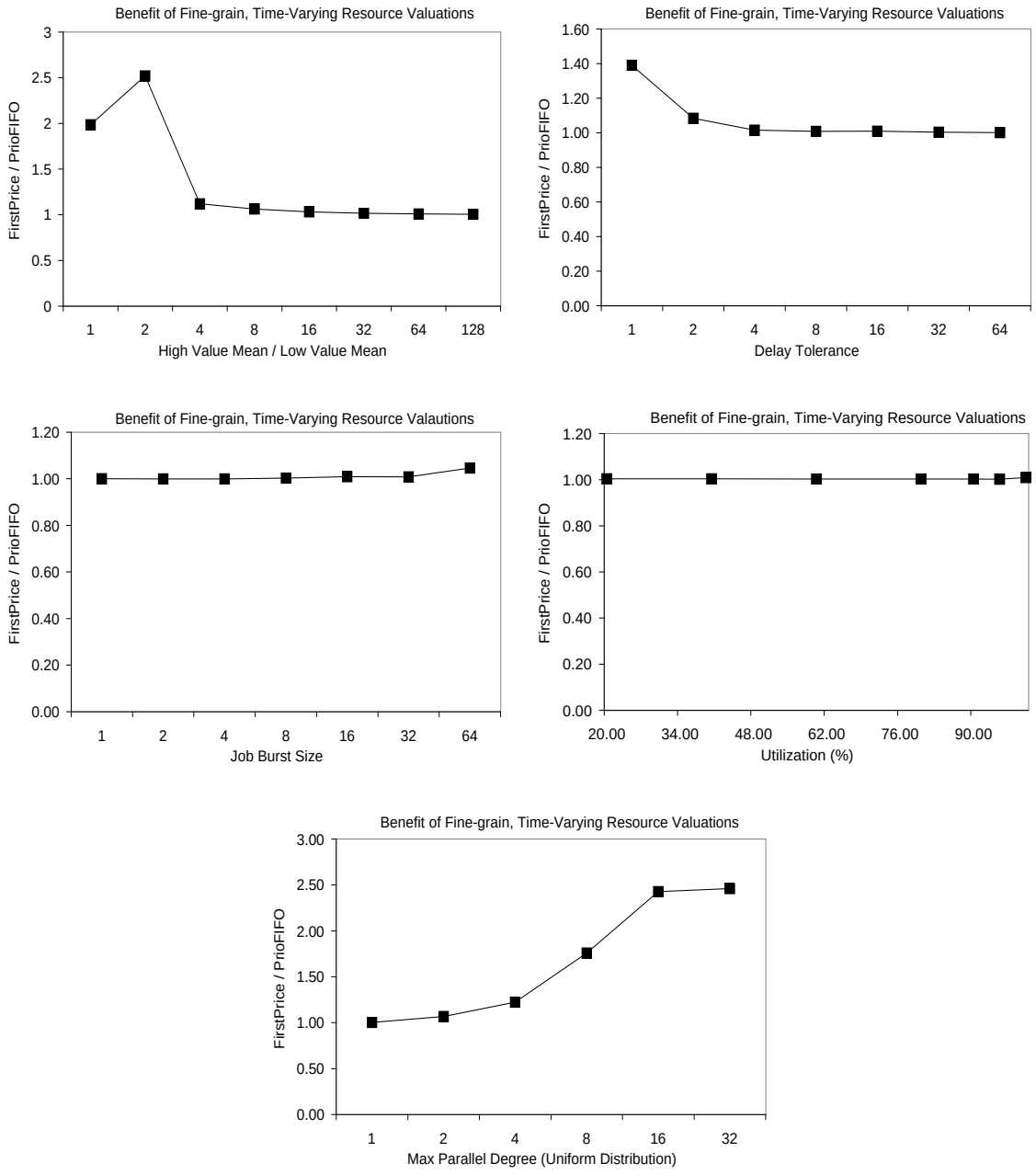

Figure 6.26: **Benefit of Fine-grain, Time-varying Resource Valuations.** These figures summarize our results which quantify the benefits of fine-grain, time-varying valuations (FirstPrice) compared with coarse-grain, constant valuations (PrioFifo). For the most part, we observe that coarse-grain, constant valuations are nearly as effective as fine-grain, time-varying valuations under most sequential workloads and regimes of operation. On the other hand, we observe significant benefit to having fine-grain, time-varying valuations both when parallelism in the workload was high and when valuations were all nearly the same in sequential workloads. Both are cases which push PrioFifo towards FIFO behavior.

rithms either maximize (e.g., aggregate utility) or minimize (e.g, mean slowdown) the sum of these metrics across the entire workload. Preemption then occurs whenever preempting a running job to run a queued job results in either a larger (for maximized metrics) or smaller (for minimized metrics) contribution to overall goodness.

We evaluate SJF, PRIOFIFO, and FIRSTPRICE for preemption. For SJF we attempt to minimize mean slowdown, so preemption occurs whenever the pairwise reordering of a running job and a queued job results in a smaller contribution to mean slowdown. For, PRIOFIFO, we generally attempt to run higher priority jobs first and so preemption occurs whenever a queued job is found with a higher priority than a running job. Any number of metrics could be used to produce this behavior given our model of minimizing or maximizing a sum of metrics. For FIRSTPRICE, we attempt to maximize aggregate utility, so preemption occurs whenever the pairwise reordering of a running job and a queued job results in a larger contribution to aggregate utility. In all cases, jobs are preempted at most once in order to help avoid starvation. We observed negligible differences in the results when preempting multiple times.

User Tolerance to Delay

The first experiment examines performance sensitivity to user tolerance to delay. We study user tolerance to delay because the impact of preemption is directly related to how long users are willing to wait in the queue. The more sensitive users are to delay, the more effective preemption should be since users would rather have a running job preempted as opposed to waiting for resources to become available. To quantify this phenomena, we simulated FIRSTPRICE, PRIOFIFO, and SJF with and without preemption and evaluated their performance. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrivals, normal burst sizes, and normal CPU lengths. Valuations were 80/20 bimodally distributed ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 8. Figure 6.27 plots the results.

We observe that preemption makes little difference over a wide range of user tolerances to delay. Only when users are highly sensitive to delay are benefits observed and even there benefits are relatively small. The main reason for this rather surprising result is expected queueing time for high value jobs. Since CPU lengths are normally distributed and the cluster consists of 32 nodes, a high valued job waiting at the head of the queue does not wait long for a node to become free. Focusing on just a single node in the cluster,

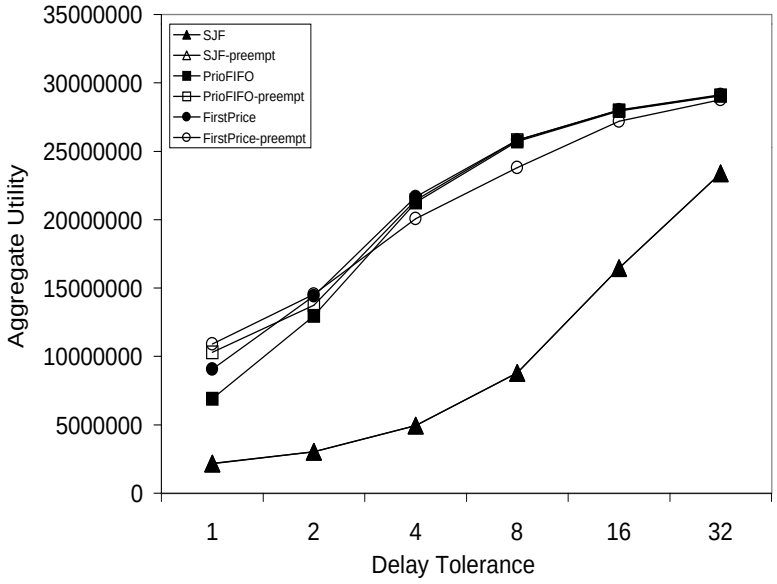

Figure 6.27: **Batch Sensitivity to User Tolerance to Delay with Preemption.** This figure plots aggregate utility as a function of user tolerance to delay when preemption is used. The system was fully utilized. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts, normal burst sizes, and normal CPU length jobs. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$, $\mu_{high} = 5000$).

the expected waiting time for that node to become free is simply the mean of the normal CPU length distribution. Given 32 instances of such nodes, it is even more likely that the minimum waiting time across all nodes is even smaller. Consequently, high valued jobs generally face small queueing delays and thus the impact of waiting is relatively small.

Variation in Resource Valuations

The second experiment examines performance sensitivity to variation in resource valuations in the workload. More specifically, we evaluate performance with and without preemption as we vary the mix of low and high valued jobs in the workload. Examining performance sensitivity to the mix of valuations is relevant since the more low valued jobs in the workload, the more opportunities there are to preempt. Again, we compare FIRST-PRICE, PRIOFIFO, and SJF with and without preemption and use workloads consisting of 32 users submitting bursts of sequential jobs with normal interarrivals, normal burst sizes, and normal CPU lengths. Valuations were bimodally distributed ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 8. Figure 6.28 plots the results as we vary the fraction of low and high valued jobs in the workload and vary user tolerance to delay.

We observe that preemption is beneficial only when users are highly sensitive to delay and the fraction of low valued jobs in the workload is high. This result makes intuitive sense given that we already saw that preemption delivers negligible improvements unless users are highly sensitive to delay. Assuming they are highly sensitive to delay and adding the additional condition that there are many low valued jobs, then it becomes clear that a highly delay sensitive job at the head of the queue facing many low valued jobs should see some improvement through preemption. For example, when users have a delay tolerance of 1 and 90% of the jobs are low valued, we see that preemption delivers about 30% more value delivered to users. In all other cases, though, improvements are either negligible or there is actually a slightly negative impact on performance. Decreases in performance can be attributed to our preemption strategy being only pairwise optimal.

Job Length

The third experiment examines performance sensitivity to job lengths. The results from this study are relevant since the benefits of preemption are directly related to the expected time a high valued job has to wait before resources become free. This in turn is related to the average length of low running jobs that could be preempted. If many low

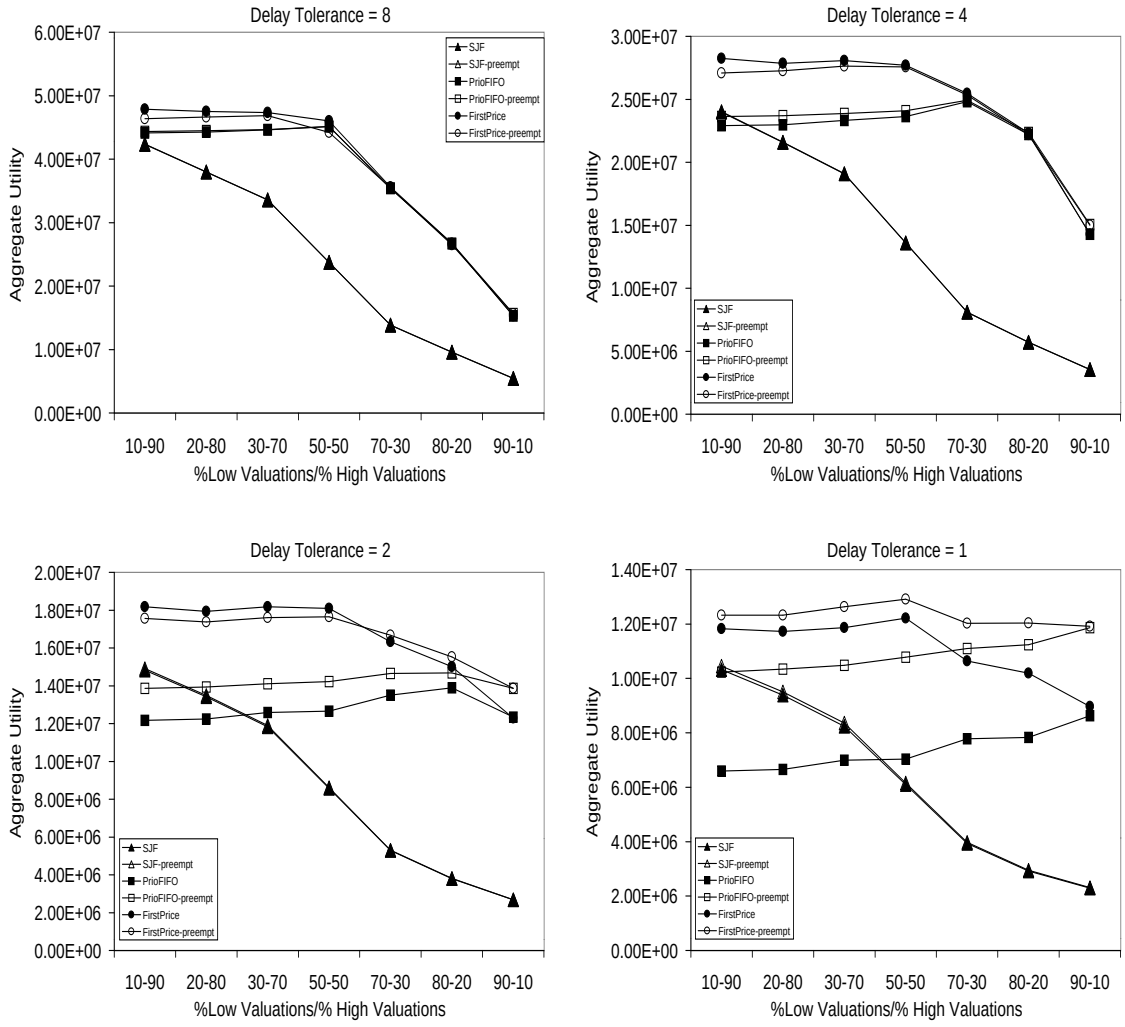

Figure 6.28: **Batch Preemption Performance Sensitivity to Mix of Low/High Value Jobs with Preemption.** These figures plot aggregate utility as we vary the fraction of low and high valued jobs in the workload when preemption is used. Results are shown for several user tolerances to delay. The system was fully utilized. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts, normal burst sizes, and normal CPU length jobs. Valuations were taken from bimodal distributions with $\mu_{low} = 50$ and $\mu_{high} = 5000$.

valued jobs are executing and they are all long-running, then the most eligible high value job faces a large queueing time which impacts performance. As low valued job lengths increase so does expected queueing delay and its impact on performance. In this experiment, we quantify this behavior by simulating the same six algorithms with a workload consisting of 32 users submitting bursts of sequential jobs with normal interarrivals, normal burst sizes, bimodally distributed valuations ($\mu_{low} = 50$, $\mu_{high} = 5000$) and a delay tolerance of 8. Figure 6.29 plots the results as we vary the ratio of low valued CPU lengths to high valued CPU lengths.

We observe that as low valued jobs run longer that the benefits of preemption increases. As low valued jobs run longer compared to high valued jobs, the expected time the most eligible high valued job has to wait for resources to free up increases. Preemption in these cases is beneficial since it prevents value from decaying for high valued jobs by preempting long-running, low valued jobs. We see that as we increase the ratio of CPU lengths that preemption results in a twofold increase in performance when delay tolerance is 8. In addition, as we decrease tolerance to delay, benefits are even higher since the effects of waiting become more significant. We observe that at high ratios, all three algorithms perform better with preemption. For FIRSTPRICE and PRIOFIFO, this is not surprising given that they use explicit valuations. For SJF, this occurs only when the ratio is greater than 2, in which case the shortest jobs are also the most valuable jobs.

Parallel Degree

The fourth experiment examines performance sensitivity to parallelism in the workload. We examine parallelism in the workload in order to evaluate whether parallelism can have as significant an impact as it did for batch environments without preemption. For this experiment, we use the same six algorithms and used workloads consisting of 32 users submitting bursts of parallel jobs with normal burst interarrivals, normal burst sizes (mean burst size = 16), and normal CPU lengths. Valuations were 80/20 bimodally distributed ($\mu_{low} = 50$, $\mu_{high} = 5000$) and all users had a delay tolerance of 8. In all experiments, jobs were preempted whenever an opportunity to improve performance arose. No backfilling occurred on unused nodes if a preempted job used more nodes than the preempting job required. Figure 6.30 plots the results when the mean parallel degree for jobs is taken from a uniform distribution.

We observe that preemption for parallel workloads results in either no performance

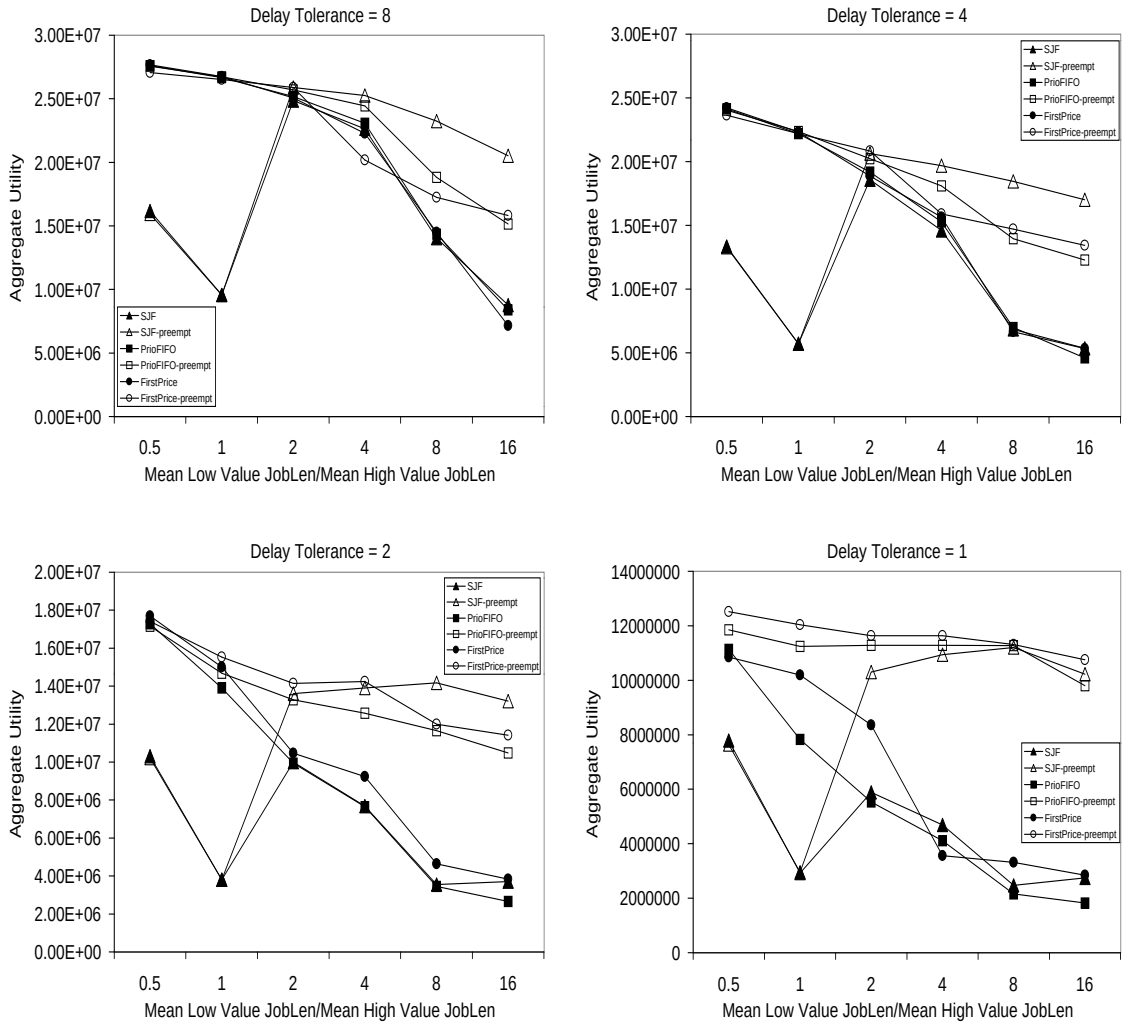

Figure 6.29: **Batch Preemption Performance Sensitivity to Job Lengths with Preemption.** These figures plot aggregate utility as a function of the ratio of low valued job CPU lengths to high valued job CPU lengths when preemption is used. The system was fully utilized. Workloads consisted of 32 users submitting bursts of sequential jobs with normal interarrival times between bursts, normal burst sizes, and normal CPU length jobs. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$ and $\mu_{high} = 5000$) and all users had a delay tolerance of 8.

Figure 6.30: **Batch Performance Sensitivity to Parallelism in the Workload with Preemption.** This figure plots aggregate utility as a function of maximum parallel degree when parallel degree is taken from a uniform distribution and preemption is used. The system was 80% utilized to keep the simulation stable since backfilling is imperfect. Workloads consisted of 32 users submitting bursts of parallel jobs with normal interarrival times between bursts, normal burst sizes, and normal CPU length jobs. Valuations were taken from 80/20 bimodal distributions ($\mu_{low} = 50$ and $\mu_{high} = 5000$) and all users had a delay tolerance of 8.

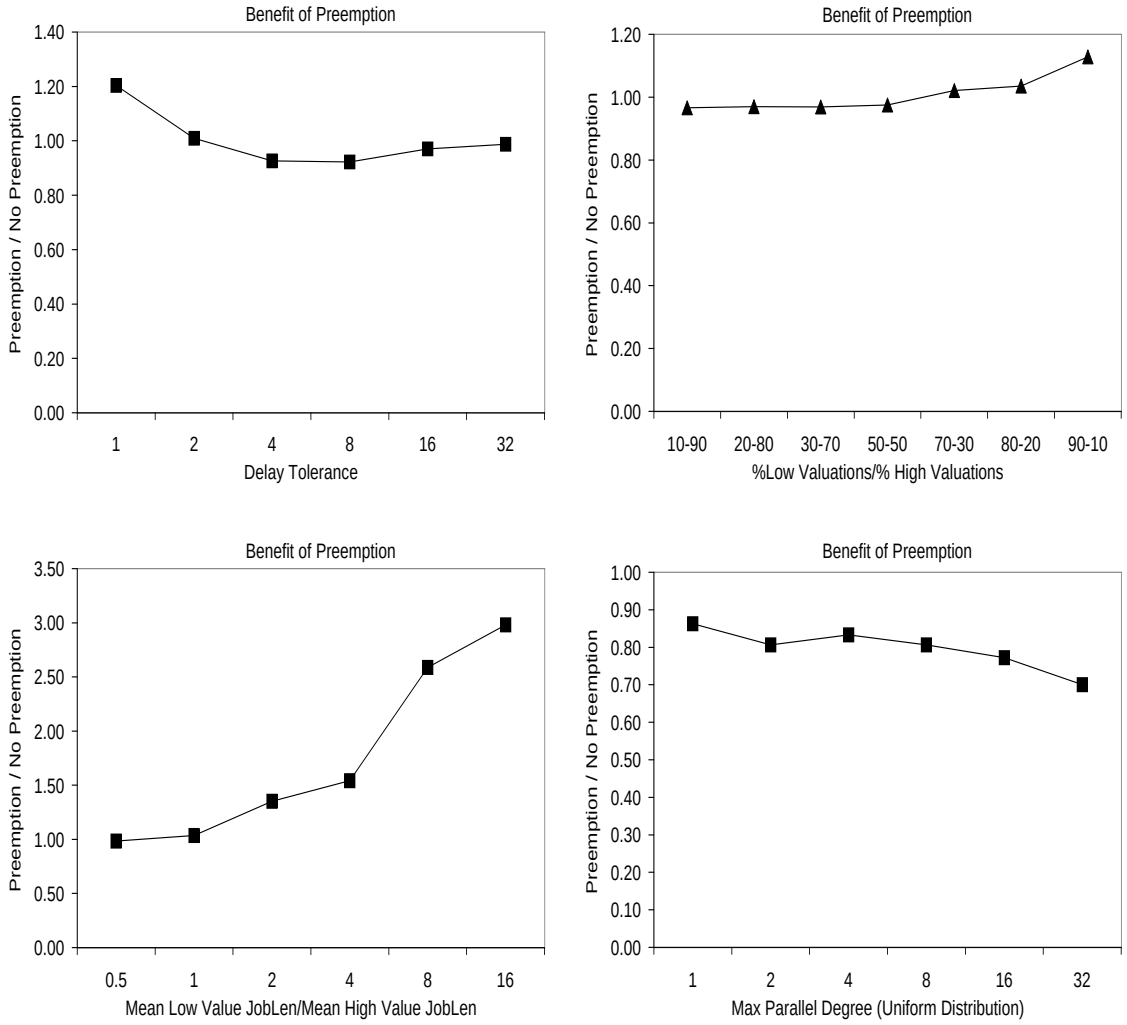

Figure 6.31: **Benefit of Preemption in Batch Environments.** These figures summarize our results which quantify the benefits of preemption in batch queue systems. We observe that preemption in batch environments delivers small benefits, or even lowers performance, in all but a few special cases. The main reason for this result is expected queueing delay for high valued jobs. Since scheduling based on resource valuations already provides high valued jobs a way to bypass low valued jobs in the queue, any benefits realized through preemption must provide high valued jobs a way to wait even less in the queue. For the workloads we examine, we observe that such opportunities do not make a significant impact on performance since the expected time for resources to free up is not very significant

improvement or lower performance for all ranges of parallelism. Our specific preemption strategy is likely the cause here. In our strategy, parallel jobs are preempted whenever a pairwise improvement in performance is detected, regardless of whether the preempting job uses all of the preempted job's nodes. In cases where the preempting job does not use all the preempted job's nodes, these resources remain idle since backfilling in our simulations was only implemented for regular scheduling and not when preempting. Consequently, when fixing the system utilization in a given experiment, preemption causes resources to go idle which in turn increases queueing delay for future jobs since resources are not being utilized to their maximum extent. We speculate that these queueing delays are the cause for the lower performance. Evaluation of other preemption strategies is future work.

Summary

In summary, we observe that preemption in batch environments delivers small benefits, or even lowers performance, in all but a few special cases (Figure 6.31). The main reason for this result is expected queueing delay for high valued jobs. Since scheduling based on resource valuations already provides high valued jobs a way to bypass low valued jobs in the queue, any benefits realized through preemption must provide high valued jobs a way to wait even less in the queue. Once at the head of the queue, there are two possibilities for a high valued job: (i) it can wait for resources to become available or (ii) it can preempt a low valued job. Our results showed that the waiting is a completely reasonable option in all but a few special cases. Prioritization based on value already decreases queueing time significantly for high valued jobs. We hypothesize that preemption on top of that is not effective in most cases since the expected time for resources to free up is not very significant.

Chapter 7

Conclusions

In this dissertation, we have shown that by applying market-based resource management to clusters, significantly more value can be delivered to users compared with traditional approaches. In market-based systems, resources are allocated through markets. Resource allocations are determined using economic mechanisms such as auctions where users place explicit valuations on the resources being contended for as expressed by mechanism-specific "bids". Explicit valuations provide systems with extra information which is used to optimize for user value, and high prices combined with charging creates feedback which causes users to back off the system when contention is high. Through prototype implementations of real systems and extensive simulation, we have shown that market-based systems are both feasible to implement and demonstrate substantially higher performance compared to traditional systems. In this chapter, we summarize our results and present future directions for this work.

7.1 Summary

To support our thesis that market-based systems can deliver substantially higher value to users compared with traditional approaches, we used a twofold approach. First, we identified the key requirements that market-based systems must meet in order to deliver more value. Then, we demonstrated that practical implementations of real systems that meet these requirements can be constructed by describing market-based systems for both time-shared and batch environments. These implementations form the framework for our the simulations which make up the second part of the approach. Using the Stingray event-

driven simulator, we conducted an extensive series of simulations to systematically explore the design space and evaluate various forms of market-based cluster resource management. Through simulation, we quantified the types of workloads and the regimes of operation in which market-based systems demonstrate the largest advantages over traditional systems. Using these results, we make recommendations on when it makes sense to apply market-based resource management to systems.

7.1.1 Implementations

Practical implementation experience with real systems is needed in order to provide the grounding and to advance the state-of-the-art in research in market-based systems. To date, most of the research in market-based resource management and related areas has been theoretical, with the bulk of the work focusing on the computation of equilibrium prices. Evaluation of economic mechanisms has mostly been done in isolation without the context of a real system or a real user community and their workloads. Consequently, while the literature on economic mechanisms continues to grow, substantially less progress has been made in figuring out how to apply these results and bring them to bear on real distributed systems. To address this problem, we assert that future work in this area should focus significantly more attention on real systems that are both feasible to implement and are based on economic mechanisms that demonstrate quantitative advantages given the expected workload.

Towards this end, we began our investigation of market-based systems by first presenting a system architecture for market-based cluster resource management. Using this architecture, a wide variety of market-based systems can be described. The architecture thus provides a vocabulary which allows us to compare and contrast different market-based designs while focusing on essential details, specifically how systems attempt to deliver more value to users. Previous work in market-based systems has produced few designs and even less implementations. Systems were presented using disparate terminology and architectures which made it difficult to focus on essential design decisions. By applying our architecture, we bring these design decisions to the forefront. Specifically, by examining access modules, economic allocators, and resource managers, it becomes clear how users express value, how value is being translated into resource allocations, and how these differential resource allocations are being enforced.

To make the ideas in the architecture concrete and to demonstrate that build-

ing real market-based systems that users can use is feasible, we constructed two prototype implementations. We presented the REXEC remote execution system as an infrastructure for implementing market-based resource management. Using REXEC as the base, we then presented two realizations of our system architecture, one for time-shared environments, the other for batch environments. We showed that through proper extensions to the REXEC system and the leveraging of work in differential resource allocations, specifically proportional-share resource allocation, that practical systems can be built which afford users significantly more control over their resource allocations while under the control of market forces. To show that such systems can be usable, we deployed these implementations on subsets of the UC Berkeley Millennium cluster, a 300+ CPU cluster of SMPs, and subjected them to use by real users.

7.1.2 Performance

We used simulations to quantify the performance advantages of market-based systems over traditional approaches. Two types of systems were simulated, time-shared systems based on the time-shared computational economy (first prototype) and batch systems based on the Anemone batch queue system (second prototype). In both cases, we examined performance sensitivity to user valuations in terms of importance and immediacy, i.e., how fast importance decays with execution time. In addition, we also examined sensitivity to a variety of workload parameters including job arrival times, job resource requirements (specifically CPU lengths and parallel degree), burstiness in the workload, and utilization among others. Using results from these sensitivity studies, we showed under what types of workloads and in which regimes of operation market-based systems are most effective and made recommendations on when market-based systems are most appropriate.

Time-shared Systems

For time-shared systems, we showed that the value of knowing resource valuations is a significant advantage while the value of having price feedback is only moderate for time-shared systems, where slowdowns are due solely to sharing of the CPU. Because time-shared systems do not use admission control, all applications immediately compete against each other for resources once submitted. Without some means to differentiate between what is important and what is not, traditional resource management systems can do no better than to provide fair-share scheduling when applications are long-running, compute-intensive

processes, as they often are with clusters used primarily for scientific computation. Given this, we demonstrate that when workloads are bimodal, consisting of low value jobs and high value jobs, that the value of knowing valuations is tremendously important for delivering high value when resources are heavily contended for.

We observed that the larger the difference in valuations, the larger the benefit of allocating resources based on valuations. This makes intuitive sense since if all jobs were equally valuable, then a fair-share scheduler should perform just as well as one that allocates based on value. For valuations which are bimodally distributed, a reasonable assumption for many workloads, we observed that market-based systems deliver up to 300% more aggregate utility when the difference in value is 10x. As the difference in value between low and high valued jobs increases, so does the benefit of being able to allocate resources based on value. As we increase the difference to 100x, we observed up to 500% more value delivered to users. In both cases, results were obtained using non-bursty workloads where individual job arrivals were separated by random amounts of times.

We observed that as workloads become burstier, the benefits of knowing valuations increases. We observed that these benefits are realized at even low overall levels of utilization. As workloads become burstier, the degree of contention that competing processes face also increases. As this degree of contention rises, it becomes increasingly important for systems to focus on the jobs delivering the highest value in order to avoid all jobs suffering slowdowns due to high degrees of multiprogramming. We showed that as burst size increases, there is a steady decrease in value delivered for traditional systems and that even at 50% utilization, even moderately bursty workloads with bursts sizes of 3-5 can result in significantly higher value delivered for market-based systems. For example, with a burst size of 4, we observed performance gains of 300%, 100%, and 30% for user delay tolerances of 4, 8, and 16.

We observed that even when users are moderately insensitive to delay that market-based systems deliver significantly more value. As long as there is significant resource contention and users are not completely insensitive to delay, there is value in knowing valuations. We observed that even with non-bursty workloads on a fully utilized system that market-based systems deliver dramatically higher value over a wide range of sensitivities to delay. This difference in value rapidly increases as user tolerance to delay approaches 1 (i.e., users are willing to wait at most one additional job length before the value of a job is 0) and approaches zero as users become infinitely tolerant to delay. We observed that even

when users are fairly tolerant to delay (valuation cutoff = 8-16) that knowing valuations still results in 200% to 300% more performance.

We observed that price feedback is moderately effective in altering user behavior and ultimately delivering higher value. In our simulations, users either submit or defer based on current rates relative to the value of their jobs. Using this strategy, we showed up to 30% improvements in value compared with a market-based system where users submit jobs regardless of current rates. When contention occurs between multiple high valued jobs, all high valued jobs obtain large resource allocations and thus all high valued jobs significantly impact each other's execution. When enough high value jobs compete, these slowdowns eventually cause all high value jobs to deliver very low value. Price feedback mitigates this by smoothing bursty workloads out over time, thereby reducing the probability that a large number of high valued jobs compete simultaneously. When rates are high, high (and low) valued jobs defer since executing now would yield a small resource allocation while being charged a high rate.

We observed that timely price dissemination from cluster nodes can have a significant impact on performance with slow price dissemination causing large chunks of performance to be lost. We showed that with periodic price updates, value delivered to users decreases with increased delays between periodic price updates. This makes intuitive sense since as updates become more infrequent, old price information is being used to assign incoming jobs to cluster nodes which may no longer offer the best rates. We showed that even with non-bursty workloads, the difference in performance between updating once a minute versus once every ten minutes is about 25%. As we increase the level of burstiness, we observed the performance gap increase from 200% with a burst size of 4 to over 400% when the burst size is 7. We also showed that algorithms such as Mitzenmacher's CHOOSE2 and REXEC's policy of multicasting price updates deliver better performance compared to periodic updates once a minute with benefits improving with workload burstiness.

We observed that user populations consisting of subpopulations of simple strategic users lead to less value delivered to users with users hoarding increasing amounts of funds over time. In these simulations, users assigned rates based on expected competing rates during their jobs' executions and their valuation decay rates as opposed to simply based on their valuations. We observed that as the fraction of strategic users in the population increases, overall value to users decreases. This result hints at the importance of providing incentive-compatible strategies. On the other hand, because the strategy played by users

was non-optimal for various reasons, including the invalid assumption that expected rates equals current rates and that other users do not play the same strategy, by no means does this result suggest that non-incentive compatible strategies should be discarded. Other reasons related to the nature of the artificial funds in the economy were discussed earlier.

Batch Systems

For batch systems, we showed that the value of knowing resource valuations is significant for both sequential and parallel workloads. Because batch systems implement admission control, benefits are not as substantial as with the time-shared case, where the impact of low valued jobs competing is significant. With batch systems, access to resources is mediated through use of a queue with sharing occurring only in space rather than in time. Because of this, the key question is how to prioritize jobs in the queue based on valuations. For market-based systems, prioritization can be highly accurate since users can assign time-varying bids which reflect their current valuations of the resources. Using these bids, high value jobs can easily bypass low value jobs in the queue which may have arrived earlier. However, as our results show, other strategies that are not based on fine-grain, time-varying valuations can also achieve similar effects under certain workloads, although not always intentionally.

We observed that for sequential jobs, the value of knowing valuations is important mainly to the extent that the system can distinguish between low and high value jobs. We observed that traditional supercomputing center policies of having multiple FIFO queues with different scheduling priorities performs almost as well as using a first price auction with time-dependent bids for many sequential workloads. This result can be attributed to the use of medium and high priority queues which provide high value jobs a means to bypass low value jobs at a coarse granularity. As long as the average amount of work in the medium and high priority queues is small, these higher priority queues are effective in expressing value and reducing queueing delays accordingly. Nevertheless, we observed that the first-price auction still delivers more value to users compared to FIFO, SJF, and the supercomputing center policy over all possible workloads.

We observed that for parallel jobs, the value of knowing valuations becomes increasingly important as parallelism in the workload increases. Because workload parallelism rarely maps perfectly onto resources, there will be times when the most eligible parallel job has to wait for resources to become available since only a subset of the required nodes may

be free. This waiting effectively increases queueing delay at all levels, including the high priority queues used in supercomputing center queueing systems. While waiting, valuations decrease since value delivered to the user decreases with increased execution time. Thus, with longer queueing delays, it becomes increasingly likely that jobs in the front of the high priority queue have lost significant amounts of value and thus more important work lies farther back in the queue. Consequently, value decreases as these queueing delays increase and the value of using time-varying valuations, as in the first price auction, increases relative to supercomputing center policies.

We observed that the ability to preempt is important only when the expected time for resources to become available for the most eligible job is high. This makes intuitive sense since if the time for resources to become free is small, it becomes less advantageous to preempt since jobs might be just as well off waiting. We observed that the benefits of preempting increase as high valued jobs are more sensitive to delay since waiting becomes more intolerable. As expected, we also observed that when high valued jobs face large queueing delays due to large amounts of long-running, low valued jobs, the value of preemption increases the longer the mean time to wait is. Finally, we see that in some cases preemption can actually hurt performance. For highly parallel and bursty workloads, we observed that preemption actually results in lower performance across the board. As discussed earlier, though, we speculate that this is due to a deficiency in the preemption algorithm we used.

Summary

In summary, we observed that market-based resource management is most effective for time-shared systems when resources are heavily contended for. We observed that when system utilization is high or when workloads are bursty, even at low utilization, market-based systems deliver substantially higher value over a range of workload parameters. We also observed that this value increases with the difference in valuations users place on the resources. Additionally, we also saw that price feedback can be effective even when the only slowdown due to multiprogramming is sharing of the CPU.

For batch systems, we observed that market-based resource management is most effective compared to the best traditional algorithm simulated (supercomputing center policies) when workloads are highly parallel or when there is substantial variation in job lengths. Both of these cases correspond to situations where supercomputing center policies suffer due to long queueing delays in high priority queues. Overall, we recommend market-based re-

source management for batch systems in general since implementations are simple and they perform universally better than traditional systems.

We observed that explicit resource valuations for both time-shared and batch systems can lead to tremendous improvements in value delivered to users, over an order of magnitude in some cases but more commonly factors of 2x-5x which are also quite good. Early evidence with real users on market-based systems showed that users have significantly different valuations of cluster resources, which further strengthens these results. Through simulation, price feedback was shown to also be effective, leading to up to a 30% improvement in value delivered in some cases. Benefits could also improve as other performance effects due to sharing are taken into effect.

7.2 Future Directions

This work opens up several interesting avenues for future work in market-based cluster resource management:

- For time-shared environments, one natural extension of this work is a study of market-based mechanisms that work well with coordinated coscheduling. To efficiently support workloads consisting of parallel applications, coordinated coscheduling [66] is often required to coordinate the scheduling of processes in parallel programs. With parallel applications, resource allocations for individual processes are not independent since processes often must be scheduled concurrently in order to make forward progress due to communication between processes. Consequently, applying market-based resource management to individual nodes may not be sufficient to deliver value to parallel programs since node-based allocation could result in imbalances in CPU allocations across processes. An interesting area for future work involves how to extend the rate-based mechanism in this dissertation to work with both explicit and implicit coscheduling [5].

- Another interesting direction is a study of market-based resource management when multiple resources are allocated using market-based mechanisms. As clusters have become more widely used, their specific uses have also become more varied. Traditionally, clusters have been used primarily for performing scientific computations. Today, they are used to support a variety of different workloads which are demanding

on CPU resources as well as memory, network, and I/O. For these emerging workloads, there is often more than one scarce resource. An interesting direction for future work would be to examine how to apply market-based resource management when there are markets for multiple resources which must be concurrently allocated (e.g., CPU time and memory). Work in this area might include the design of new mechanisms for allocation of multiple resources or the application of existing ones, such as combinatorial auctions [23] and their evaluation.

- The design of application-specific libraries and software agents which simplify the expression of value is another promising direction that would make market-based systems easier to use. Ideally, users should not have to concern themselves with the details of low-level resource allocations. Instead, the expression of value should be based on application-specific performance metrics that the user understands. Enabling this type of expression requires understanding how application-level performance metrics depend on specific allocations of the underlying computational resources. In some cases, forming these mappings can be straightforward, in particular if the relationships are based mainly on a single resource (e.g., compute-intensive applications). In other cases, though, more general techniques need to be developed [4]. Future work in this area involves the development of such techniques and their embedding in application-specific libraries and software agents.

- The design of systems that dynamically partition resources based on the demand for different qualities of service is another interesting direction with practical value. Traditionally, providing different qualities of service on clusters has been done using static resource partitioning, where some fraction of the cluster is dedicated to time-shared use while the remaining fraction of the cluster is allocated for dedicated use. This design, unfortunately, is suboptimal when the partitioning is not aligned with observed demand for both types of service and user valuations of the underlying resources. A better strategy would be to let market forces determine how best to partition cluster resources. In this dissertation, we described an asymmetric design where demand for dedicated CPU time determined how large the batch partition would be. It would be interesting to extend this work to a symmetric case where simultaneous demand for time-shared and dedicated CPU time determined the partitioning of resources.

- System evaluation based on other performance metrics besides aggregate utility would

also be interesting. One interesting avenue for future work here is to quantify the predictability of market-based systems. Explicit resource valuations are useful because they allow users to influence their resource allocations (or scheduling priority for batch systems) based on their valuations of the resources. Valuations are expressed using funds and the expectation is that overall cost should be close to how much the user actually valued the resources. A user with a large valuation is willing to pay more and thus assigns a higher valuation, but if actual costs are substantially higher (or lower) than his already high willingness to pay, this corrupts the user's mental model of how the system behaves and makes the system less predictable and less usable. Future work here would involve quantifying this phenomena by performing a study that compares utility to revenue, both at the system and individual user level.

- Finally, another interesting avenue for future work is the design of market-based systems that support server applications where value delivered is based on ongoing service as opposed to application completion time. Server applications providing web services in hosting environments such as Exodus [29] have different utility functions compared to most scientific computing environments. Value delivered is a function of how well client requests are handled over time as opposed to server completion time, which is often meaningless since most server applications are not intended to terminate. Despite this difference, server applications could still stand to benefit from market-based resource management. With market-based resource management, resources could be allocated based on observed demand as opposed to being statically provisioned based on expected demand and some tolerance for burstiness. As a result, servers would be capable of accommodating large bursts in demand while at the same time only paying for resources used.

In addition, there is also interesting work to be done in extending this work to other types of distributed systems:

- Applying market-based resource management to systems of systems where mutual trust exists is another promising area for future work. The larger the pool of aggregate resources, the greater the probability that some fraction of the pool is underutilized at any given time. Grid computing systems [34, 43] aggregate computing resources from remotely distributed systems and enable resource sharing across wide area networks. When local demand on a system exceeds local supply, Grid systems are beneficial

since they allow local demand to be farmed out to remote resources. Unfortunately, sharing of resources in these systems is usually assumed to be entirely altruistic in nature. Consequently, there are no incentives for users to make underutilized resources available as much as possible since they receive no rewards. Market-based systems could be used to provide such incentives while at the same time delivering the same benefits that were demonstrated on a single cluster.

- Another interesting study involves the application of market-based resource management to distributed systems where trust cannot be assumed. The less we assume about trust, the larger the pool of potential resources. On the Internet, for example, a large fraction of the computers connected to it are idle at any given time. These unused resources could potentially be used to perform important computations if they were made available. United Devices [88], for example, provides a distributed computing platform that currently aggregates idle resources from over 960,000 computing devices on the Internet. By highlighting a few applications with important consequences for society (e.g., curing cancer) and by providing the opportunity to win prizes depending on the amounts of computation donated, they provide minimal incentives for users to participate which so far have been enough to attract a large user community. Given the additional overhead of probabilistically verifying correctness, it would be interesting to see what benefit market-based resource management could bring to such a large system.

Bibliography

[1] Rakesh Agrawal and Ahmed K. Ezzat. Location independent remote execution in nest. *IEEE Transactions on Software Engineering*, 13(8):905–912, August 1987.

[2] David Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan, and Hari Balakrishnan. System support for bandwidth management and content adaptation in internet applications. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[3] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A case for now (networks of workstations). *IEEE Micro*, February 1995.

[4] Mohit Aron, Sitaram Iyer, and Peter Druschel. A resource management framework for predictable quality of service in web servers. 2001.

[5] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan Mainwaring. Scheduling with implicit information in distributed systems. In *Proceedings of the 1998 ACM SIGMETRICS Conference*, 1998.

[6] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for unix. In *Proceedings of the 1999 USENIX Conference*, 1999.

[7] Amnon Barak, Oren La'adan, and Amnon Smith. Scalable cluster computing with mosix for linux. In *Proceedings of Linux Expo '99*, pages 95–100, May 1999.

[8] Antonio Marinho Pilla Barcellos, Joao Frederico Lacava Schramm, Valdir Rossi Belmonte Filho, and Claudio Fernando Resin Geyer. The hetnos network operating system: a tool for writing distributed applications. *Operating Systems Review*, October 1994.

[9] Paul Barham. A fresh approach to file system quality of service. In *Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, 1997.

[10] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet—a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–38, February 1995.

[11] Abhishek Chandra, Micah Adler, Pawan Goyal, and Prashant Shenoy. Surplus fair scheduling: A proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[12] Fangzhe Chang and Vijay Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *Proceedings of the 9th International Symposium on High Performance Distributed Computing*, 2000.

[13] Andrew Chien, Scott Pakin, Mario Lauria, Matt Buchanon, Kay Hane, and Louis Giannini. High performance virtual machines (hpvm): Clusters with supercomputing apis and performance. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[14] Brent N. Chun and David E. Culler. Market-based proportional resource sharing for clusters. Technical Report CSD-1092, University of California at Berkeley, January 2000.

[15] Brent N. Chun, Alan M. Mainwaring, and David E. Culler. Virtual network transport protocols for myrinet. *IEEE Micro*, January/February 1998.

[16] Cisco Systems. Local director (http://www.cisco.com/warp/public/cc/pd/cxsr/400/index.shtml).

[17] David D. Clark. A model for cost allocation and pricing in the internet. In *Proceedings of MIT Workshop on Internet Economics*, March 1995.

[18] David Cliff. Minimal-intelligence agents for bargaining in market-based environments. Technical Report HPL-97-91, HP Labs, June 1997.

[19] Ron Cocchi, Deborah Estrin, Scott Shenker, and Lixia Zhang. A study of priority pricing in multiple service class networks. In *Proceedings of the ACM SIGCOMM '91 Conference on Communications Architectures and Protocols*, 1991.

[20] Ron Cocchi, Scott Shenker, Deborah Estrin, and Lixia Zhang. Pricing in computer networks: Motivation, formulation, and example. *IEEE Transactions on Networking*, 1993.

[21] Cray. Network queueing environment (`http://www.cray.com/products/software/nqe.html`).

[22] David Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steve 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.

[23] Sven de Vries and Rakesh Vohra. Combinatorial auctions: A survey. 2001.

[24] Alan Demers, Srinivasan Keshav, and Scott Shenker. Anaylsis and simulation of a fair queueing algorithm. In *Proceedings of the 35th IEEE Computer Society International Conference (COMPCON)*, pages 380–386, March 1990.

[25] Tony DeWitt, Thomas Gross, Bruce Lowekamp, Nancy Miller, Dean Sutherland, and Jaspal Subhlok. Remos: A resource monitoring system for network-aware applications. Technical Report CMU-CS-97-194, Carnegie Mellon School of Computer Science, 1997.

[26] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software—Practice and Experience*, 21(8), August 1991.

[27] Richard Edell and Pravin Varaiya. Providing internet access: What we learn from index. *IEEE Network*, 13(5), 1999.

[28] D. H. J Epema, Miron Livny, R. van Dantzig, X. Evers, and Jim Pruyne. A worldwide flock of condors : Load sharing among workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[29] Exodus Communications. Exodus web page (http://www.exodus.com).

[30] Donald Ferguson. Microeconomic algorithms for flow control in virtual circuit networks. In *Proceedings of IEEE Infocom*, 1989.

[31] Donald Ferguson, Christos Nikolaou, and Yechiam Yemini. An economy for managing replicated data in autonomous decentralized systems. In *Proceedings of International Symposium in Autonomous Decentralized Systems*, pages 367–375, 1993.

[32] Donald Ferguson, Yechiam Yemini, and Christos Nikolaou. Microeconomic algorithms for load balancing in distributed computer systems. In *Proceedings of 8th International Conference on Distributed Computer Systems*, 1988.

[33] Adam J. Ferrari, Stephen J. Chapin, and Andrew S. Grimshaw. Process introspection: A heterogeneous checkpoint/restart mechanism based on automatic code modification. Technical Report CS-97-05, University of Virginia, March 1997.

[34] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[35] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The ssl protocol version 3.0 (internet-draft). 1996.

[36] Andrew Geweke. A system for batch-mode economic scheduling of a cluster of workstations. Master's thesis, University of California at Berkeley, 2001.

[37] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. Glunix: a global layer unix for a network of workstations. *IEEE Transactions on Software Engineering*, 1998.

[38] Robert Gibbons. *Game Theory for Applied Economists.* Princeton University Press, 1992.

[39] Google. Google web page (`http://www.google.com`).

[40] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation.* USENIX Association, 1996.

[41] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of the*

ACM SIGCOMM '96 Conference on Communications Architectures and Protocols, pages 157–168, August 1996.

[42] Carey. G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[43] Andrew Grimshaw, Adam Ferrari, Fritz Knabe, and Marty Humphrey. Legion: An operating system for wide-area computing. Technical Report CS-99-12, University of Virginia, Department of Computer Science, 1999.

[44] Thomas Gross, Peter Steenkiste, and Jaspal Subhlok. Adaptive distributed applications on heterogeneous networks. In *Proceedings of the 8th Heterogeneous Computing Workshop*, 1999.

[45] Steven Hand. Self-paging in the nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, February 1999.

[46] Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. An implementation of parallel operating system for clustered commodity computers. In *Proceedings of Cluster Computing Conference '97 (CCC97)*, March 1997.

[47] John C. Hull. *Introduction to Options and Futures Markets (3rd Edition)*. Prentice-Hall, 1998.

[48] IBM. Load leveler (`http://www-1.ibm.com/servers/eserver/pseries/software/sp/loadleveler.html`).

[49] Inktomi. Inktomi web page (`http://www.inktomi.com`).

[50] Jiuban Ju, Gaochao Xu, and Jie Tao. Parallel computing using idle workstations. *Operating Systems Review*, July 1993.

[51] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[52] Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani. Solaris mc: A multi computer os. In *Proceedings of the 1996 USENIX Conference*, 1996.

[53] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, April 1997.

[54] Umesh Maheshwari. Charge-based proportional scheduling. Technical Report MIT/LCS/TM-529, Massachusetts Institute of Technology, April 1995.

[55] Alan M. Mainwaring and David E. Culler. Design challenges of virtual networks: Fast, general-purpose communication. In *Proceedings of the ACM SIGPLAN '99 Symposium on Principles and Practice of Parallel Programming*, May 1999.

[56] Mark S. Miller and K. Eric Drexler. *The Ecology of Computation*, chapter 4: Comparative Ecology: A Computational Perspective. Elsevier Science Publishers, October 1988.

[57] Mark S. Miller and K. Eric Drexler. *The Ecology of Computation*, chapter 10: Incentive Engineering for Computational Resource Management. Elsevier Science Publishers, October 1988.

[58] Mark S. Miller and K. Eric Drexler. *The Ecology of Computation*, chapter 7: Markets and Computation: Agoric Open Systems. Elsevier Science Publishers, October 1988.

[59] Mark S. Miller, David Krieger, Norman Hardy, Chris Hibbert, and E. Dean Tribble. *Market-Based Control: A Paradigm for Distributed Resource Allocation*, chapter 5: An Automated Auction in ATM Network Bandwidth. World Scientific Press, 1996.

[60] Michael Mitzenmacher. How useful is old information? In *Proceedings of 16th ACM Symposium on Principles of Distributed Computing*, 1997.

[61] Myricom. The gm api. 1999.

[62] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, 1987.

[63] Jason Nieh, Chris Vaill, and Hua Zhong. Virtual-time round-robin: An o(1) proportional share scheduler. In *Proceedings of the 2001 USENIX Conference*. USENIX Association, 2001.

[64] Brian D. Noble, M. Satyanarayan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[65] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus: An architecture for extensible distributed systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 58–68, 1993.

[66] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of 3rd International Conference on Distributed Computer Systems*, pages 22–30, May 1982.

[67] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *IEEE Computer*, 21(2), February 1988.

[68] PBS. Portable batch system (`http://www.openpbs.org`).

[69] James S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, University of Tennessee, Department of Computer Science, 1997.

[70] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of the 1995 USENIX Winter Conference*, 1995.

[71] Platform Computing. Load sharing facility (http://www.platform.com/products/LSF).

[72] Christ Priest. Algorithm design for agents which participate in multiple auctions. Technical Report HPL-2000-88, HP Labs, July 2000.

[73] Christ Priest. Economic dynamics of agents in multiple auctions. In *Proceedings of Autonomous Agents 2001*, 2001.

[74] Jim Pruyne and Miron Livny. Managing checkpoints for parallel programs. In *Proceedings of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, 1996.

[75] Ori Regev and Noam Nisan. The popcorn market – an online market for computational resources. In *Proceedings of the 1st International Conference on Information and Computation Economies*, 1998.

[76] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-pcs. In *Proceedings of IEEE Aerospace*, 1997.

[77] Lawrence A. Rowe and Kenneth P. Birman. A local network based on the unix operating system. *IEEE Transactions on Software Engineering*, 8(2), March 1982.

[78] Jakka Sairamesh. *Economic Paradigms for Information Systems and Networks.* PhD thesis, Columbia University, 1997.

[79] Scott Shenker, David Clark, Deborah Estrin, and Shai Herzog. Pricing in computer networks: Reshaping the research agenda. *ACM Computer Communications Review*, 26(2):19–43, 1996.

[80] Prashant Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 ACM SIGMETRICS Conference*, pages 44–55, June 1998.

[81] Ken Shirriff. Building distributed process management on an object-oriented framework. In *Proceedings of the 1997 USENIX Conference*, 1997.

[82] Ian Stoica, Hussein Abdel-Wahab, and Alex Pothen. *Lecture Notes in Computer Science, Vol. 949*, chapter A Microeconomic Scheduler for Parallel Computers, pages 200–218. Springer-Verlag, 1995.

[83] 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. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 58–67, September 1994.

[84] Michael Stumm. The design and implementation of a decentralized scheduling facility for a workstation cluster. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 12–22, March 1988.

[85] Vaidy S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[86] Todd Tannenbaum and Michael Litzkow. Checkpointing and migration of unix processes in the condor distributed processing system. *Dr. Dobbs Journal*, pages pp. 40–48, February 1995.

[87] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, 1985.

[88] United Devices. The global metaprocessor service (http://www.ud.com/products/global_mp_service.htm).

[89] University of California at Berkeley. Uc berkeley millennium project (http://www.millennium.berkeley.edu).

[90] Pravin Varaiya, Richard Edell, and Harish Chand. Index: The internet demand experiment. 1996.

[91] William Vickrey. Counterspeculation, auctions and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[92] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain, CO, December 1995.

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

[94] Carl A. Waldspurger, Tag Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–177, February 1992.

[95] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11. USENIX Association, 1994.

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

[97] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, 1983.

[98] Michael Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[99] Yechiam Yemini, Apostolos Dailianas, and Danilo Florissi. Marketnet: A market-based architecture for survivable large-scale information systems. In *Proceedings of the 4th ISSAT International Conference on Reliability and Quality in Design*, August 1998.

[100] Yechiam Yemini, Apostolos Dailianas, Danilo Florissi, and Gur Huberman. Marketnet: Protecting access to information systems through financial market controls. *Decision Support Systems Journal*, 28(1-2):205–216, 2000.

[101] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, pages 177–184, August 1999.

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

[103] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Utopia: A load sharing facility for large, heterogenous distributed computer systems. *Software—Practice and Experience*, 1992.