UNIVERSITY OF CALIFORNIA, SAN DIEGO

Practical Market-Based Resource Allocation

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

in

Computer Science

by

Alvin AuYoung

Committee in charge:

Professor Alex C. Snoeren, Chair
Professor Jeanne Ferrante
Professor David Miller
Professor Joel Sobel
Professor Amin M. Vahdat

2010

The dissertation of Alvin AuYoung is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2010

DEDICATION

I dedicate this dissertation to my family: they were there for my first day of school and have been waiting patiently for me to finish my last.

EPIGRAPH

自己永不足教人永不疲倦
—孔 夫 子

The market can stay irrational
longer than you can stay solvent.
—John Maynard Keynes

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I cannot ask for a better group of friends, colleagues and mentors than those I have at UC San Diego. Much of what I have accomplished and what I still aspire to do is largely inspired by these tremendous people. A few paragraphs of acknowledgements aren't enough to express my sincere gratitude, and I apologize in advance to anybody I unintentionally omit.

I would first like to thank my thesis committee: David Miller and Joel Sobel for patiently correcting my constant misuse and abuse of economics theory; Jeanne Ferrante for her insightful questions and prompt feedback; Amin Vahdat, who, along with Brent Chun, provided the initial inspiration for this area of study, and even though he is not my adviser, still provides a constant source of motivation, support and tremendous vision, particularly with his ability to frame smaller and less obvious research problems into a broader and more inspirational context; and finally my advisor, Alex Snoeren, who has gone above and beyond his role as a mentor and advisor, while somehow managing his life as a new father, a daily commuter from Orange Country, and annual champion of the department NCAA Tournament bracket. He has exhibited a *tremendous* amount of patience to let me explore an area of research foreign to him while still providing a constant source of constructive criticism and technical enlightenment. I have learned much from his ability to focus on the important details of any data, and to whittle down my often incoherent thoughts, and focus them into specific research problems. With his guidance, I have matured significantly as a scholar, but can only hope to one day exhibit half as much patience, passion, acumen and rigor for detail that he has demonstrated to me over the past several years.

I am also indebted to John Wilkes, Janet Wiener, Jeff Shneidman, Thomas Sandholm, David Parkes, Chaki Ng, Dave Levin, Kevin Lai, Laura Grit, Brent Chun, and Phil Buonadonna, all of whom have been my collaborators and colleagues, and provided countless hours of discussion, debate and last-minute paper efforts, without which, much of the ideas behind this dissertation would not have been possible.

Although I am excited about the future, I will sorely miss the camaraderie of my current and former colleagues: Michael Vrable, Patrick Verkaik, Andreas Pitsillidis, Justin Ma, Ryan Braud, Sriram Ramabhadran, Travis Newhouse, Eugene Hung, Diane Hu, Cynthia Bailey Lee, John McCullough, David Hutches, Yuvraj Agarwal, Kashi Vishwanath, Chris Tuttle, Barath Raghavan, Marti Motoyama, Priya Mahadevan, Diwaker

Gupta, Yu-chung Cheng, and Jeannie Albrecht. I have learned much of what I know from them, and their brilliance, empathy and levity make coming into the office every day something to look forward to and I cannot imagine having made it through graduate school without the benefit of their company.

I would also like to thank the various faculty and staff with whom I have interacted. In particular, Geoff Voelker, George Varghese, Stefan Savage, Allan Snavely, Joe Pasquale, Ranjit Jhala and Keith Marzullo have always had an open door and always managed to impart wisdom with only a few words, doing so with the utmost humility. Also a special thanks to Brian Kantor, Chris Edwards, Dave Wargo, Kathy Reed, Yuka Nakanishi, Kim Graves, Julie Conner, and CSE help, who are members of the underappreciated staff who make this department run smoothly, and always seem to do it with a smile.

Thanks to Jimmie Ye, Caitlin White, Arghavan Salles, Aron Lum, Eric Lin, Matthew Kwong, Clement Kam, Carri Chan and Ali Bashir, all of whom I have tremendous respect for, and have been significantly inspired by in some way. A special thanks to my housemates Kia Yang, Karl Ni, Brendan Morris, Ed Liao, Allen Chu, and Arun Batra for everything else: the many carpool commutes, birthday pranks, sports debates, fancy dining excursions, television marathons, intramural sports and adventures from Sorrento Valley all the way to Rancho Penasquitos — these memories will certainly last a lifetime.

To Danielle Gaeta, whom I certainly cannot say enough about, for providing the unconditional love, support and an inexplicable willingness to listen to and discuss my technical ramblings; you are the best thing that happened to me in San Diego.

And last and most of all, I would like to thank my family: Kevin and Mona for constantly setting the bar high for me ever since kindergarten, and for being the example of what hard work and dedication can achieve; and 爸爸 and 媽媽 for providing unwavering support when things get difficult and instilling the confidence to pursue my dreams. I can only hope to one day make you as proud as I am to be your son.

Finally, I must also acknowledge that portions of this dissertation are in part a reprint of published academic work: (1) Chapter 3 is in part a reprint of material that appears in the Proceedings of the OASIS Workshop, 2004, by Alvin AuYoung, Brent N. Chun, Alex C. Snoeren, and Amin Vahdat; and Chapter 23 of *Market Oriented Grid and Utility Computing* published by Wiley, 2009, written by Alvin AuY-

oung, Phil Buonadonna, Brent N. Chun, Chaki Ng, David C. Parkes, Jeff Shneidman, Alex C. Snoeren and Amin Vahdat; (2) Chapter 4 is in part a reprint of material that appears in the ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis, 2009, by Alvin AuYoung, Amin Vahdat and Alex C. Snoeren; (3) Chapter 5 is in part a reprint of material that appears in the Proceedings of the IEEE International Symposium of High Performance Distributed Computing, 2006, by Alvin AuYoung, Laura Grit, Janet Wiener and John Wilkes. The dissertation author is the primary investigator and author in these papers.

VITA

2002	B. S. in Electrical Engineering and Computer Science, *summa cum laude*, University of California, Berkeley
2006	M. S. in Computer Science, University of California, San Diego
2010	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

"Evaluating the Impact of Inaccurate Information in Utility-Based Scheduling." Alvin AuYoung, Amin Vahdat, and Alex C. Snoeren. *22nd ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis* **(SC-09)**, 2009.

"Two Auction-Based Resource Allocation Environments: Design and Experience." Alvin AuYoung, Phil Buonadonna, Brent N. Chun, Chaki Ng, David C. Parkes, Jeff Shneidman, Alex C. Snoeren, and Amin Vahdat. *In Market Oriented Grid and Utility Computing, Rajkumar Buyya and Kris Bubendorfer, editors*, 2009.

"Service contracts and aggregate utility functions." Alvin AuYoung, Laura Grit, Janet Wiener, and John Wilkes. *15th IEEE International Symposium on High Performance Distributed Computing* **(HPDC-15)**, 2006.

"Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems." Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent N. Chun. *10th USENIX Workshop on Hot Topics in Operating Systems* **(HotOS-X)**, 2005.

"Mirage: A Microeconomic Resource Allocation System for Sensornet Testbeds." Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes. Jeffrey Shneidman, Alex C. Snoeren and Amin Vahdat. *2nd IEEE Workshop on Embedded Networked Sensors* **(EmNetS-II)**, 2005.

"Resource Allocation in Federated Distributed Computing Infrastructures." Alvin AuYoung, Brent N. Chun, Alex C. Snoeren, and Amin Vahdat. *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on-demand IT InfraStructure* **(OASIS)**, 2004.

ABSTRACT OF THE DISSERTATION

Practical Market-Based Resource Allocation

by

Alvin AuYoung

Doctor of Philosophy in Computer Science

University of California, San Diego, 2010

Professor Alex C. Snoeren, Chair

The arrival of large-scale open platforms such as cloud-computing infrastructures and petascale clusters is fueling the emergence of a new class of users and applications with large and diverse resource needs. Optimizing for traditional system-centric metrics such as response time or throughput does not capture the diverse and often conflicting needs of different users and resource providers. Collaboration between economists and computer scientists over the past ten years has lead to many important results describing designs of market-based mechanisms to address exactly these issues. However, there is very little progress in building these systems due to apprehensions about their potential unfairness and fragility in a live deployment. The resulting reality is that systems continue to rely on traditional and inadequate designs despite theoretical results demonstrating the potentially significant value added to all stakeholders by designing market-based systems.

This dissertation investigates the sensitivity, applicability, and practicality of a market-based policy to increase the efficiency of resource allocations — measured using a utilitarian social welfare metric — in real, large-scale computing systems. We build upon previous work in this space by moving beyond "paper designs" and using an iterative process of implementation, deployment, measurement, modeling and simulation.

We begin by presenting Mirage, our implementation of a microeconomic resource allocation system for a sensor network, which is still in use today. In this context, we identify the compromises that deployed market-based systems must make when using traditional theoretical designs due to computational challenges and various imperfect conditions of a live deployment. Nonetheless, based upon user data, we find that these systems are still expected to be robust to typical user errors seen in traditional computing systems. We also describe extensions of existing designs to address some of these limitations. Finally, we discuss our experience with deploying systems with real users and applications and conclude by discussing the forthcoming challenges in pushing more widespread adoption of these market-based designs in real systems.

Chapter 1

Introduction

Many computer systems have reached the point where the goal of resource allocation is no longer to maximize utilization or response time; instead, when demand exceeds supply and not all needs can be met, one needs a policy to guide resource allocation decisions. One natural policy is to seek *efficient* usage, which allocates resources to the set of users who have the highest utility for the use of the resources. Researchers have frequently proposed market-based mechanisms to provide such a goal-oriented way to allocate resources among competing interests while maximizing overall "happiness" (or "aggregate utility") of the users. In general, however, market-based allocators have yet to catch on in practice or with a large body of computer science researchers.

This dissertation investigates the potential of applying economic market design principles to solve emerging resource allocation problems in distributed systems. Systems such as high-performance computing clusters, data centers and other large-scale resource infrastructures play a tremendous role in supporting large-scale scientific applications [121]. However, the arrival of large-scale open platforms such as cloud-computing infrastructures and petascale clusters is fueling the emergence of a new class of users and applications with large and diverse resource needs. A desire to schedule these applications on a consolidated resource infrastructure poses a challenge to traditional system design. For example, an application may prefer twice as much memory at the cost of an additional unit of processing power or storage, or a user may prefer to reduce power consumption at the cost of decreased throughput. Optimizing for a traditional system-centric metric such as response time or throughput does not capture the diverse, and often conflicting needs of these stakeholders. It is clear that there is significant potential

to improve the value and satisfaction delivered by these systems by introducing a more sophisticated allocation policy.

The challenge for any system to deliver on these goals is to first elicit this "utility" information from its users and then to determine how to use the corresponding information to meet its goals. Collaboration between economists and computer scientists over the past ten years has lead to many important results that describe *how to design* economic-inspired mechanisms to address exactly these issues. However, there is very little progress on *how to build* systems with these mechanisms. The resulting reality is that systems continue to rely on traditional and inadequate designs despite theoretical results demonstrating the potentially significant value[1] added to all stakeholders by designing economics-inspired systems. This dissertation focuses on bridging this widening gap between theory and practice. Through a process of deployment, measurement and modeling, we uncover limitations of proposed theoretical models, and motivate the development of refined models to better fit emerging system designs.

The contributions of the work contained herein are as follows: (1) we implement and deploy two real market-based systems and present our corresponding deployment experience, (2) we provide a simulation-based analysis of the expected performance of a market-based system in the presence of imperfect operating conditions, (3) we introduce a market-based scheduling framework that increases the utility of both a profit-seeking service provider and its utility-maximizing clients, and finally, (4) we implement a market-based scheduler in a small Hadoop cluster that can take advantage of hidden preference information in real applications to increase both system and application utility.

On a broader scale, this research intends to provide a deeper understanding of when and where such economics-inspired theory can be applied to other problems in systems and networking, and perhaps allow us to rethink traditional systems and networking design in order to allow us to build systems to meet our current and emerging needs.

1.1 Emerging Large-Scale Applications

In the past decade, computer and network-based applications have reached an unprecedented level of ubiquity in society. From support for on-line communities (Face-

[1] We use the terms value, utility and user satisfaction interchangeably unless otherwise noted.

book, Myspace, Twitter), to clients for sharing multimedia (BitTorrent, Hulu, YouTube), new applications are constantly developed that can leverage increasingly cheap and powerful computational [99] and networking capabilities [30]. The unique characteristics of these applications pose a challenge for their underlying resource infrastructures to support. We discuss several examples below.

The growth and presence of Internet applications continues to generate massive volumes of application data for Web service providers. Careful analysis of this data yields information about individual users, social networks, and other patterns of usage that providers like Google, Yahoo! Microsoft and Facebook find useful to refine their services and products. Since these data are simply too large to store and analyze with existing resource infrastructures, companies have developed sophisticated programming models and scheduling systems like MapReduce [37], Hadoop [16] and Dryad [55], to allow these large data-intensive jobs to run on existing hardware. The motivation for these new scheduling frameworks highlights the significant limitations of existing infrastructures, irrespective of resource capacity.

The emergence of diverse scientific applications also poses a challenge to high-performance computing clusters. When using metrics like system throughput or response time, the performance of traditional scientific applications scale well with increased resource capacity. However, the emergence of a *diverse* set of applications and users, with their own resource configurations (loosely coupled vs tightly coupled), distinct deadlines (e.g., seismic monitoring application wishing to process available data immediately following a major seismic event) and importance (perhaps defined by relative government-endowed grant money towards a research project) have forced researchers to design addition system support to account for these user-specific metrics [66, 93].

The increasing availability of computational and networking hardware is fueling the development of large-scale distributed testbeds as a foundation for the development of large-scale distributed applications and systems. Resource allocation in testbeds like PlanetLab [90], EmuLab [44] and GENI [77] is designed on the principle of best-effort scheduling, with the objective of scaling application performance with the underlying resource capacity. However, experience with these systems demonstrates that such a policy can fail to meet this objective: when users are responsible for both selecting resources and moderating resource consumption, this policy often leads to poor resource utilization and potentially avoidable resource contention [87]. Since these problems arise

primarily from user behavior, and users lack both an incentive and mechanism to shift or curb consumption during times of peak demand, it is not clear if these systems can in fact, scale with additional resource capacity using existing allocation policies.

Finally, non-scientific applications have also emerged as consumers of these large-scale infrastructures, and supporting these applications using traditional allocation policies has proven to be challenging. For example, Pixar Animation Studios leased access to an external data center to support the rendering of a large and complex set of computer graphics imagery for one of its films [13]. Despite having ample system capacity, the existing scheduling policies used by the data center did not support the diverse workflow constraints, task deadlines and priorities within each job, and therefore schedules often had to be hand-tuned in order to achieve the necessary job throughput [7].

As an increasing number and variety of applications find opportunities to leverage available on-demand computation, network or storage, existing resource infrastructures will become critical bottlenecks to application scalability and performance. From these anecdotes, it is clear that today's computational resource infrastructures are no longer limited to traditional applications with predictable needs and that emerging applications are fundamentally different from those in the past. On one dimension, applications vary greatly in raw quantity of resource consumption: varying in required run-time (from seconds to months), size (from a single processor to tens-of-thousands of processors), or network radius (localized versus geographically distributed network topology). On another dimension, applications, vary in less easily quantifiable measures, such as importance (e.g., a scientific application simulating the effects of a new cancer-treatment drug), or elasticity (e.g., sensitivity to time deadlines). It is becoming clear that the existing allocation policies of emerging shared resource infrastructures must adapt to support these emerging applications.

1.2 Shared Resource Infrastructures

The traditional method to mitigate resource contention is to provision excess capacity in anticipation of peak demand. When capacity exceeds supply, even the most simplistic scheduling policies can provide competitive performance with more sophisticated policies. Unfortunately, provisioning for peak demand is slowly becoming an untenable strategy due to prohibitive capital and energy costs [20]. Furthermore, as we mention in the anecdote of distributed applications, simply having excess capacity in the

system does not mean that applications can or will take advantage of it. Together, these concerns have lead to a paradigm shift in the provisioning of resource infrastructures.

Today, we see an increasing number of providers of on-demand computation and storage. This so-called *Cloud-computing* or *Service-oriented computing* model has arrived in enterprise computer systems [86], whereupon internal resources are provisioned and leased—on-demand—to consumers. This model has many benefits: clients avoid the hassles and expense of maintaining their own infrastructure and provisioning it for their peak load; clients benefit from the service provider's economies of scale, expertise, and management; and multiple clients can share a common infrastructure and (sometimes) expensive software licenses, resulting in lower costs. Companies like Amazon, Sun and Microsoft are increasingly investing in this cloud-based services model.

This shift has also manifested itself in academic and non-commercial infrastructures like PlanetLab [90], GENI [77] and the Grid [45], whereupon the system is composed of loosely-coupled resources which are federated or shared across hundreds or thousands of users. Existing policies do not lend well to considering the cost of using such infrastructures to support a variety of applications. As these infrastructures begin to play a more critical role in serving the computational and network needs of society at large, it is critical that they move beyond traditional or ad-hoc allocation policies.

1.3 Market-Based Allocation Policies

Studies show that traditional scheduling policies such as *first-come-first-served* (FCFS) or *proportional-share* (PS) are unable to support such heterogeneity in applications. In fact, more sophisticated policies dramatically outperform these approaches: researchers have claimed improvements in aggregate user utility ranging from 4–20% [67] to 100–1400% [25], with the magnitude of the improvement depending upon the details of the particular environment. To put this improvement into context, if an increase in utility leads to a similar increase in revenue in a commercial infrastructure, consider that an estimate of annual revenue from Amazon's Elastic Compute Cloud is roughly $220 million annually [29] — which means that a 4% improvement could lead to an increase in over $8 million in annual revenue.

The key idea behind this class of scheduling policies is the use of market-inspired mechanisms to infer information about the characteristics of the resource demand in order to determine socially desirable allocations, for instance, allocating scarce resources

to the users who derive the most value, or *utility*, from consumption. This "market-based" approach uses this additional information (importance, elasticity of raw resource consumption) to make an informed scheduling decision and adapt to dynamic resource demand. However, despite such lofty claims, utility-based techniques are rarely deployed in practice.

1.4 Challenges

Opponents of market-inspired techniques often dismiss these mechanisms as too burdensome for end-users, inequitable in their allocations, or too fragile to use in production environments [101]. While proponents cite the results of numerous independent simulation and theoretical studies showing significant performance improvement, each of these studies assume perfect operating conditions; however, operators argue that such assumptions rarely hold in practice, and it is unclear how market-based systems will perform in practice relative to these idealized results.

In this dissertation, we address the following challenges to help advance this long-standing debate about the viability of market-based scheduling systems:

- **Lack of implementation experience.** The majority of research in large-scale market-based allocation systems is limited to theoretical designs or simulation-based results, and very little has been done to describe how to *implement* market-based designs in real systems. Even in traditional — and often simpler — scheduling systems, there are many parameter settings to tune upon deployment, such as reservations in backfilling algorithms, or the number and relative levels of priority queues. With limited experience in either identifying or appropriately setting the corresponding market parameters such as currency policies and resource pricing mechanisms, there is a significant portion of unexplored design space for implementing a market-based framework for a real system.

- **Lack of deployment experience.** While there does not seem to be disagreement that market-based systems offer the *potential* to improve allocations in shared resource infrastructures, most of the apprehensions involve the departure of simulated results from actual performance upon deployment. Specifically, there are concerns that these systems may be too fragile, non-deterministic, or burdensome

for users in practice, and therefore, it is unclear how much improvement to expect from a market-based system in a live deployment.

- **Lack of empirical system analysis.** Another major source of apprehension for these systems is a lack of quantitative experimental data. While many concerns surrounding market-based systems require experience with a live deployment to address properly, many important analyses and simulation studies can be conducted simply using empirical data, such as user preferences, demand and other basic workload characteristics. Indeed, many advanced scheduling techniques in high-performance computing systems are developed using detailed trace-based simulation studies from workloads of real systems. Although they are not a complete substitute for a deployment, real workloads fill an essential gap between general, theoretical designs and the application of the design to a particular environment. Such studies can provide fundamentally important comparisons between traditional and existing approaches, and address such basic issues as when a particular scheduling policy is more appropriate than other and identifying the basic trade offs.

- **Lack of empirical user analysis.** Unlike existing scheduling systems, market-based systems fundamentally rely on "rational" behavior from users. Specifically, these systems assume that users will provide demand information to the market, and in turn react properly to the corresponding prices from the market. However, with a lack of empirical evidence for how users behave, it is unclear whether or not users *will* behave as expected, and if not, what impact this *imperfect behavior* will have on the performance of the system. Indeed, prior theoretical work has outlined the potential for significant cognitive burden for users in specific market settings [63, 85], and prior empirical work has demonstrated that a large set of users of computational system are limited in the information they can provide [66]. However, neither observation has been applied to these market-based designs.

- **Designing for multiple objectives: users and providers.** Finally, for commercial systems, a resource service provider desires an allocation policy that increases *profitability*. It is not clear how the potentially independent objectives of service-provider profit-maximization, and user utility-maximization interact when using a market-based allocation policy, particularly when a resource service provider

must also consider its own underlying infrastructure costs.

1.5 Contributions

Given the strong momentum towards shared resource infrastructures, and the continued emergence of diverse and large-scale applications, we believe that now is an opportune time to re-visit these sophisticated allocation techniques, in particular, market-based systems.

1.5.1 Hypothesis and Approach

We believe that the fear and apprehension towards sophisticated, market-based allocation techniques are founded, and in particular, that existing principals in systems — comprising of current applications and users — cannot be expected to interact like agents in an idealized market. However, we believe that we can still implement market-based mechanisms that will allow systems to more efficiently support the diversity in emerging applications. In this dissertation, we take the following steps to test this hypothesis:

First, we address the lack of deployment experience with these systems by implementing and deploying two real market-based resource allocation systems from which we measure the impact of the market on real users, and uncover limitations in existing market designs.

Second, we address the lack of rigorous empirical analysis of these systems by using our collected deployment data to drive a set of detailed simulation studies to quantify the benefits of a market design with respect to traditional scheduling systems in the face of imperfect operating conditions from both the system and the users.

Third, we introduce a market-based allocation policy that can be used by a commercial infrastructure to maximize profitability, and a non-commercial infrastructure to maximize user satisfaction; we argue that such a market-based design can be applied to both commercial and non-commercial infrastructures.

Finally, we demonstrate that market-inspired systems can be used effectively in a production system with minimal user overhead. Specifically, we implement a market-based scheduler in a Hadoop cluster which infers utility information, and demonstrate how this information can be used to improve the performance of real applications.

1.5.2 Results

Our contributions can be summarized as follows:

- **Case studies of implementation and deployment experience.** Our deployment of a market-based allocation system reveals challenges not typically addressed in the literature. For example, we use a heuristic algorithm instead of an idealized market-clearing algorithm to handle the worst-case complexity of market-driven resource requests in a large-scale environments. Furthermore, users of market mechanisms exhibit a wide range of behavior, including unwanted and unanticipated behavior that can affect system performance. We find that there are indifferent users, who do not participate in the market, and there are sophisticated users, some of whom will properly leverage market mechanisms, and yet others, who will further manipulate the system for their own benefit and to the detriment of others (Chapter 3).

- **Empirical analysis of the impacts of imperfections in user behavior and system settings.** We quantify the sensitivity of a market-based system based upon a model of imperfect conditions and empirical data. We find that significant user uncertainty can severely decrease the aggregate performance and per-user fairness of a heuristic scheduling policy, but projected levels of uncertainty indicate significant benefit in both respects from this same policy. In each uncertainty regime, we compare how a market-based approach fares against simpler traditional allocation policies (Chapter 4).

- **An allocation model for service providers and clients.** We investigate a portion of the design space of a profit-seeking job-execution service provider, utility-maximizing clients with sophisticated job requirements, and variability in underlying resource availability; we explore the effects of profit-aware algorithms, and study how load, aggregate utility functions, and the number and cost of resources influences the service provider's profit. We demonstrate that our profit-aware approach outperforms previous ones across a wide range of conditions (Chapter 5).

- **Implementation to reduce user burden.** We implement system support for a market mechanism in an existing scheduling system that automatically infers hidden utility information of clients. We demonstrate how to use this mechanism

improve the performance of real applications in a Hadoop cluster without requiring explicit input from a user or application (Chapter 6).

1.6 Roadmap

In the next chapter, we formalize the resource allocation problem, and our assumptions, and place our contributions in the context of related work. In Chapter 3, we present a case study of deploying market-mechanisms in two different systems, and explain how the details of a particular environment (job, users, resource partition) impact the choice of market mechanisms. Based upon these deployments, we discuss how this choice of mechanism fares in the presence of the imperfections and challenges of the environment. In Chapter 4, we take a step back and present a model for the types of challenges observed in Chapter 3, and provide a trace-driven analysis of how this market mechanism is projected to fare in general systems based upon this model. In Chapter 5, we discuss how to extend our allocation system to consider a profit-seeking service provider, and how to design the allocation policy to simultaneously maximize revenue and client satisfaction. Finally, in Chapter 6, we present an implementation of an improved market-based mechanism in a real system and show how it can improve allocation decisions while also reducing usage burden on users. We conclude with our high-level observations about the feasibility of market-mechanisms in real systems and next steps in this research agenda.

Chapter 2

Background

In this chapter, we formalize the definition and scope of our resource allocation problem, and discuss the challenges with applying existing theoretical approaches, and the limitations of existing market-based designs. We conclude with an outline our approach to this problem, and a detailed roadmap of the remaining chapters.

2.1 The Resource Allocation Problem

Abstractly, the resource allocation problem can be described as a classic assignment problem. Consider a set of jobs $J \in \mathbb{N}$ that wish to run on a system with capacity $C \in \mathbb{N}$. A job $j \in J$ has a *size* and *length* that it occupies, and each of the $N \in \mathbb{N}$ available resources $i \in [1, ...N]$ is partitioned with some integer capacity c_i, such that $\sum_{i=1}^{N} c_i = C$. An example of an assignment for a job j with size $s_j \leq N$ and length l_j is $A = \{1, 2, ...s_{j-1}, s_j\}$ such that the remaining capacity of each resource $i \in A$ is $r_i = c_i - 1$ for l_j time units (assuming no other jobs have been scheduled on those resources). The goal is to assign jobs to resources such that a particular optimization criteria (or single criterion) is maximized.

There are several theoretical results which describe algorithmic solutions to a few common optimization criteria for this problem, with each solution depending on specific assumptions made about the environment, such as job characteristics, job arrivals and server capacity. As we will see, these results rely on assumptions which depart from important practical considerations, and therefore, are not directly applicable to the environments that we are interested in.

2.2 Existing Scheduling Approaches

Traditionally, schedulers for large-scale systems have a simple goal, which is to minimize the waiting time for an individual job (*response time*), or for a set of jobs (*makespan*) [1]. Extending our model described earlier, we define $t_j \in \mathbb{R}^+$ as the absolute completion time of job $j \in J$, which is determined based upon the particular job scheduling algorithm, and define $a_j \in \mathbb{R}^+$ as its arrival time ($\forall j, t_j > a_j$). This goal can thus be formulated as assigning jobs to resources to minimize $\frac{1}{J} \cdot \sum_{i=1}^{J}(t_i - a_i)$ for optimizing over average response time, or to minimize $[max_{i \in J} t_i - min_{j \in J} a_j]$, so as to optimize for the makespan.

2.2.1 Theory vs Practice

There are several algorithms which can determine an optimal, or near-optimal job schedule based upon these constraints; in this section, we discuss how critical assumptions made by a few of these algorithms render the proposed solution impractical.

Perhaps the most straightforward solution is for a scheduler to simply create a schedule based upon a solution to the aforementioned optimization problem. However, this approach requires clairvoyance to do so; solving this particular optimization problem requires foresight of all future job arrivals, and in practice, this information is not known. An alternative approach is to use queueing theory, which provides a mathematical framework to estimate the *expected* response time or makespan when workload information — such as job arrival times — is stochastic. Unfortunately, many of the results from queuing theory are not applied in practice for both quantitative and qualitative reasons.

First, applications in our environments require scheduling *parallel* jobs, which is to say that they require simultaneous use of multiple resources such that $J, C >> 1$. Unfortunately results from queueing theory are restricted to small clusters ($C \leq 2$) or workloads with small job sizes ($s_j \leq 2$). For instance, the near-optimality of a shortest-remaining-processing-time (SRPT) policy in minimizing expected response time [114] has only been demonstrated for a single-server cases ($C = 1$).

Second, theoretical results often ignore significant practical costs. For example, even if the same SRPT policy was demonstrated to be useful in a multi-server environ-

[1] It is worth noting that in addition to these goals, some administrators for larger clusters also may have a conflicting desire to maximize system resource utilization, which we will discuss later in this section.

ment, the assumptions used to establish this result ignores the cost of job pre-emption. An SRPT policy requires jobs to be pre-emptible. Specifically, an SRPT policy schedules the job with the least amount of remaining processing time, with new job arrivals potentially pre-empting a currently-running job. There is a significant cost incurred from job pre-emption: both in time required to halt, off-load and store a pre-empted job, and in the time to restore it. In large-scale systems such as high-performance computing clusters, where there are many data-intensive tasks which require significant storage, these operations are prohibitively costly, as multiple pre-emptions would require significant intermediate storage for this data.

Finally, there are qualitative concerns that such optimization unfairly biases against larger jobs. For example, the shortest-job-first (SJF) policy has similar properties as SRPT, but without pre-emption costs. Under this policy, it is believed that larger jobs are penalized unfairly (although recent work by Wierman *et al.* [113] shows that in many cases, SRPT does not penalize larger jobs anymore than the alternative, best-effort scheduling policy), hence few schedulers use this type of policy.

We discuss the limits of job-scheduling optimization and queueing theory simply to illustrate the gap between theoretical and practical scheduling algorithms. For a thorough exposition of theoretical results, refer to Pinedo [89]. Instead, the focus of this dissertation is on mechanisms that can be used in real systems. The challenges we have described force real systems to rely on heuristic scheduling policies that are engineered for their particular workload and environment. In these systems, we typically see two classes of scheduling policies: *time-sharing* and *batch scheduling*. The choice of class depends upon the size of a typical job relative to the capacity of a single resource; jobs are either scheduled to time-share a resource by running simultaneously ($c_i \geq 1$) or sequentially on a resource as a batch ($\forall i, c_i = 1$).

2.2.2 Time-Sharing Systems

Systems like PlanetLab [90] and GENI [77] are infrastructures which have many long-lived and interactive services, and typically have jobs that are significantly smaller than the available capacity of a single machine. Therefore, time-sharing is a natural policy for these systems whereupon jobs are run simultaneously on a single machine, and a machine's resources are multiplexed across all running jobs. Perhaps the most natural resource allocation policy in these systems is *proportional share*, which provides *equal*,

simultaneous access to time-shared resources to all users. It is well known, however, that proportional-share scheduling alone does not function well in systems where over-demand for resources is common. As demand for resources increases, the multiplexed time-share received by each user decreases, with any associated scheduling overheads remaining constant (or in some cases, increase due to thrashing). Taken to scale, we observe that as the number of competing users increases, the amount of time each user receives is surpassed by the pre-emption costs of multiplexing between jobs, thereby decreasing the performance of each user (or application), and thus, decreasing the value derived from the system. Solutions to this problem are primarily ad-hoc, and involve a heuristic admission-control algorithm, which will evict jobs when resource capacity is deemed saturated. However, due to a lack of fine-grained accounting in most modern operating systems, and because many tasks have unknown dependencies, it is actually difficult to tell when resources are constrained and which jobs are responsible for consumption, which makes it difficult to establish a "good" eviction or admission-control policy.

2.2.3 Batch Scheduling Systems

High-performance computing clusters [41], supercomputers [1, 2] and emerging petascale systems [93] generally support scientific applications which are often large ($s_j \in [100, 1000]$, $l_j \in [\text{hour}, \text{days}]$), and resource-intensive (e.g., either CPU-bound, I/O-bound, or both). These jobs have a performance bottleneck on at least one of the resources of the machine (e.g., CPU, memory, I/O bandwidth), and a job is given exclusive access to the machine during its run time. *First-come first-served* (FCFS) scheduling is among the earliest and simplest policies used in these systems. In this scheduling discipline, users submit a job, along with the desired degree of parallelism (size), and the jobs are executed in order of arrival. A limitation of this technique is its poor resource utilization, which can in turn negatively impact individual job response time. This problem can occur if a job at the head of the queue requires more processors than are available, and blocks all other jobs behind it in the queue — even if there are enough processors to satisfy those job requests. This problem is referred to as head-of-line blocking. EASY backfilling [70] addresses this problem by allowing other jobs to jump ahead in the queue, provided their executions do not delay the projected start time of the job at the head of the queue. EASY requires each job to be supplied with an estimated running time. Using these estimates, the scheduler can create a reservation time for the

first waiting job, and also determine which jobs can be "backfilled". Variants of this technique, such as conservative backfilling or selective reservation have been studied, where the number of jobs with reservations varies [43, 105]. A particular variant is usually chosen on an ad-hoc basis.

2.2.4 Evaluation Metrics

The reliance of scheduling heuristics yields little room for rigorous mathematical evaluation compared to their theoretical counterparts. Instead, these systems rely on empirical measurements for evaluation. For example, early administrators of batch and parallel computing environments were concerned with maximizing system utilization [70]. Historically, FCFS schedulers exhibited a utilization between 50 and 80% [105]. A combination of improvements in algorithms, such as backfilling, and increased usage [105] have increased overall utilization on these systems, where many parallel processors typically see a utilization percentage between 90 and 98% [42]. However, utilization in these systems can still vary greatly between peak and base demand, with the average varying from between 50 and 98%, depending on the time scales [42] being observed. In other words, it is possible for jobs to experience long wait times during periods of heavy demand even though a larger snapshot of resource utilization falls below 100%.

Although most scheduling systems seek to optimize metrics to improve end-user application performance, oftentimes system administrators have an incentive to maximize a metric of their own interest, such as resource utilization. Anecdotally, the motivation for this desire is to justify increasing resource capacity (i.e., buying more machines) from available funding sources. In fact, as usage of these systems increase, it is becoming apparent that users have different requirements for their jobs: some jobs are significantly more important than others, and some jobs have different levels of urgency (i.e., deadlines) than others [66]. Regardless of the motivation, this desire for multi-objective optimization has encouraged a body of research to develop more sophisticated scheduling algorithms to satisfy these needs.

2.3 A Market-Based Scheduling Approach

It should be apparent that many existing scheduling policies are based upon simple ideas, but end up using heuristics to deal with the increasing complexity of applications. Moving forward, the fundamental shortcoming of existing scheduling approaches

is rooted in the fact that they do not explicitly consider a user's utility for scheduling a job when making an allocation decision. Job priorities are used to distinguish important jobs from others [56, 105]. However, these priority-based systems typically employ between four and six priority queues, which restricts the amount of utility information a user can express for a job. Market-based approaches extend the idea of priority by allowing a scheduler to prioritize jobs based on finer-grained information, such as per-job utility functions [20, 21, 25, 54, 67, 110].

Proponents of these policies tend to view the goal of a scheduler as one to satisfy higher-level user needs [66]. Indeed, our argument for supporting applications with diverse objectives simultaneously on a consolidated infrastructure seems to provide similar motivation for this type of scheduling approach.

The objective of a market-based scheduling policy can be formulated as the well-known *weighted knapsack* optimization problem [59]. In this formulation, a knapsack is defined by its capacity (a positive integer), which represents the capacity of a resource in the system. Items to place in a knapsack represent the jobs in the system. Each item is completely defined by its *size* and *weight*: its size is an integer that represents how much of a knapsack's capacity it consumes, whereas its weight represents the value of an assignment (this model can be extended to a multi-dimensional case which considers the job length as well [64]). The goal of the optimization problem is to choose an assignment of items to knapsacks that maximizes the sum of the weights of packed items. In a market-based setting, the weights of each job represent the value, or utility that a job derives from use of the resources, and therefore, this goal is analogous to scheduling to maximize total value delivered by a resource assignment.

In economics, this particular choice of function to maximize is often called *utilitarian* social welfare function, where the sum, or aggregate utility of a population is the measure of success, rather than the distribution of utility across the population. For the remainder of this dissertation, we will refer to this metric as *aggregate utility*, and use it as the common metric when comparing allocation policies.

It is worth noting that there may be several choices of allocation which maximize aggregate utility, and we are not explicitly concerned with which one of these allocations is chosen. Furthermore, this concept of aggregate utility or efficiency is different from the economics concept of Pareto efficiency [75]. Formally, a Pareto efficient allocation outcome in our context is an allocation in which no change in allocation can increase the

utility of a single participant without also decreasing the utility of another.

In time-sharing systems, any work-conserving allocation policy is approximately[2] Pareto efficient (though not necessarily desirable) since resources will only be unallocated in the absence of demand, and resources are always allocated in the presence of demand. On the other hand, batch-scheduling systems may leave resources underutilized due to inefficiencies of the scheduling policy: for example, a conservative backfill scheduler performs allocations which are weakly Pareto superior (at least as efficient, and usually more so) to those of a simple FCFS scheduler. While we do not explicitly seek such Pareto efficient outcomes, our goal of maximizing aggregate utility is in line with a Pareto efficient outcome, since achieving an optimal aggregate utility also results in a Pareto efficient outcome. In other words, while the allocations from a market-based system and a simpler, work-conserving proportional-share system may both be Pareto efficient, the market-based system will usually lead to an allocation with higher *overall efficiency*, or aggregate utility.

Similar to the case with existing scheduling policies, designing and implementing a market-based policy for a real system cannot be done by simply solving the associated optimization problem. Creating a *robust* market for computational resources requires addressing several design and implementation challenges. In this section, we discuss these theoretical design challenges, and the challenges of implementing a practical deployment in the subsequent section.

2.3.1 Theoretical Design Challenges

Much of the work in the area of market-based computational resource allocation depends on results in economics, such as general equilibrium theory, auction theory, game theory and consumer theory. However, we will not discuss this specific literature, but will mention and summarize results as appropriate. For a thorough exposition of these subjects, refer to Mas-Colell *et al.* [75]. In the next section, we discuss the theoretical treatment of computational markets, and the challenges in designing robust markets in this context.

[2]The statement relies upon the assumption that resources are infinitely divisible, and that the consumption functions for an application are continuous; this is not necessarily true in practice, but is an assumption often made in the literature.

Mechanism Design

Perhaps the economics theory that most completely captures the issues in designing a computational market is *mechanism design* [75]. In a functioning computational market, resources are allocated based upon some social welfare metric (which we described earlier as aggregate utility), which in turn, is based upon the preferences of individual users. Traditionally, this preference information is known only by each user. Therefore, the challenge in mechanism design is to design a system with proper participation rules to provide *incentives* for users to *truthfully* reveal this private information. Note that there is a significant difference between simply *enforcing* rules, and providing incentives to follow them. Since we assume the information held by a user is private, a mechanism can require that such a user ostensibly "play by the rules", but cannot verify the accuracy of the provided information. Therefore, designing robust incentives such that a self-interested user would be directly motivated to reveal this information is of critical importance.

In our setting, the users (and therefore, their applications) represent the participants in the game, their valuation for resources represent their private information, and the allocation system is in charge of supporting any infrastructure needed to create incentives and enforce the rules of the mechanism.

Example. Imagine a simplification of our resource allocation problem, where we have one resource, and multiple users competing for that resource. Each user has a maximum willingness to pay for the resource, which we can measure in units of currency [3] — a commonly used measure of value for a resource. An allocation policy can maximize aggregate utility by simply assigning the resource to the user with the maximum willingness to pay. However, since this information is private to each user, the allocation system will have to design rules to elicit this information. An auction is a common mechanism used in this situation; the system holds an auction for the resource and awards it to the user with the winning bid. In this case, a critical part of the mechanism design problem is to create auction rules that will motivate each user to bid their maximum willingness to pay. As we mentioned earlier, we assume that all users will pursue actions to maximize their utility. In this case, a user i submits a bid b_i to the auctioneer (i.e., the scheduler), and wants to maximize $u_i = v_i - p_i$, where v_i is his maximum willingness

[3] Note, currency is just a particular unit of measurement for user "value" or "utility". In economics, these units are abstractly referred to as *utils*.

to pay for the resource, and p_i is the price that the user pays for the resource (if won, otherwise, $p_i = 0$). The utility u_i received by a user i thus depends upon his payment p_i.

First consider a standard sealed-bid[4], first-price auction, where $b_i = p_i$. If a user i believes that other users j have a lower value for resources ($v_i > v_j$), then user i will have an incentive to bid an amount lower than v_i, but higher than v_j, in order to maximize u_i. In fact, Vickrey, demonstrated that if all bidders I are risk neutral and are commonly aware that private values v_i are drawn uniformly from a known distribution ($\forall i \in I, v_i \in [0, V]$), then *every* bidder will have an incentive to under-bid: $b_i = \frac{(|I|-1)}{|I|} \cdot v_i$ [108]. Therefore, the rules of this type of auction do not elicit the desired information from users.

Now consider a Vickrey auction (sometimes referred to as a second-price auction) in which $p_i = b_j$,[5] where i has the highest (and therefore, winning) bid, and j has the second-highest bid. In other words, the winner pays the bid amount the next-highest (first losing) bid. If we repeat the analysis from the first-price auction, we can see that users have no incentive to change their bids in order to maximize their utility, since the winning payment is a function of the other bids. For a more formal explanation, refer to the seminal paper by Vickrey [108] or a more detailed discussion by Mas-Colell *et al.* [75].

While the example of a Vickrey auction and its generalized counterpart, the Vickrey-Clarke-Groves mechanism, are still abstractions of our exact problem setting, they are important results in mechanism design because they satisfy several desirable properties in common strategic settings. We will discuss a few of these properties briefly to explain what we wish to achieve with a mechanism, and what we cannot. Beyond this explanation, we do not explicitly consider all of these issues further in this dissertation.

Typical properties of a robust mechanism are incentive-compatibility, efficiency, individual rationality and budget-balance [83]. A mechanism is *incentive-compatible* if users are motivated to be truthful. In the example of a first-price auction, we saw that users were not motivated to reveal their private information truthfully, however, in the second-price auction, they were. A mechanism is *efficient*, if it leads to an allocation outcome that maximizes the desired social welfare function. In our examples, both the

[4]Sealed-bid simply means that each user's bid is concealed from other bidders.

[5]There is also a more general Vickrey-Clarke-Groves (VCG) mechanism which is commonly referred to in the related literature. In this mechanism, the winning payment is measured by the opportunity cost by their presence in the auction. The Vickrey auction as described is an example of a VCG mechanism.

first-price and second-price auction are expected to lead to an allocation that maximizes aggregate utility — although the first-price auction requires some assumptions about how users will bid.

Individual rationality is a concept that quantifies whether or not this mechanism imposes any harm to a user; formally, a mechanism supports individual rationality if a user is never made worse off (in terms of u_i) by participating in the auction. Again, in our example, the Vickrey auction achieves this property. Finally, *budget balance* is the property that the system does not run at a loss, meaning that no cash is transferred into or out of the system to support currency payments. There is also an associated notion of *weak* budget-balance, which simply states that agents (in an auction, these would be the bidders) can make payments to the system, but we cannot assume that the system can make payments to the agents. This property is important if using real currency, or any mechanism of significant value in the system to elicit information.

In the simple setting of our example, where there is a single resource, a Vickrey auction satisfies the first three properties, as well as weak budget balance, but it is known that in more general settings, *no* mechanism can satisfy all of these properties. In fact, there is an important result called the Myerson-Satterthwaite theorem (see Parkes [83] for original citation and a discussion of this result), which states that there can be no market exchange that can achieve these four properties; an exchange merely differs from an auction in that there are multiple sellers and buyers, and where an agent can be both a seller and a buyer simultaneously.

It is not clear if our systems impose the same assumptions required by the Myerson-Satterthwaite theorem to be applicable, but fortunately, we do not require all of these properties to be satisfied. In developing an allocation system, we ultimately care about overall efficiency, but due to the many practical limits of resource scheduling, a *near-efficient* solution is not only adequate, but often times preferred. Furthermore, we care about incentive compatibility and individual rationality insofar as they allow us to achieve this goal. Budget balance also becomes an issue only if dealing with real currency or a currency of value. This relaxation of requirements may make it easier to develop theoretically sound systems, as we will discuss later in this dissertation, we will be forced to make other trade-offs to accommodate the practical limitations of algorithmic solutions.

Computational Complexity

A tangential challenge that of mechanism design is handling the computational complexity of algorithms. Even if there is a mechanism that can satisfy the desired properties in mechanism design, the solutions must be computationally tractable.

As an example, consider an extension to the single-good auction from the previous section where we now auction multiple goods. In both parallel batch scheduling systems, and large-scale testbeds, users often require multiple resources simultaneously. In economics theory, this property is referred to as complementary goods: where the value in consuming multiple goods exceeds the sum of the values derived from consuming each of the goods individually. A *combinatorial* auction is one which allows bidders to express combinatorial bids, which are bids over *sets* of goods rather than only individual items simultaneously.

The algorithm used by the auctioneer to determine winning bids in a combinatorial auction is in a class of problems defined to be NP-complete [68], which is considered computationally intractable in theoretical computer science [32]. Likewise, the users themselves may also face similar computational obstacles [17, 31, 52, 63, 85, 97] when interacting with the market. For example, a user's valuation for an allocation outcome may depend upon a large number of contingencies, which require an intractable amount of computation (e.g., the traveling salesman problem [32]) to determine or communicate. So even if we have a robust mechanism (as defined by our sample list of desiderata with mechanism design) for this problem, a practical system may not be able to implement the necessary rules. There is an entire area of research called *algorithmic mechanism design* that is dedicated to the study of creating computationally tractable solutions for mechanism design [80].

2.3.2 Implementation Challenges

Thus far, we have discussed some of the theoretical design challenges to creating a robust market-based framework such as challenges in designing appropriate information-elicitation mechanisms, and using computationally tractable algorithms. However, even if we have such a computationally tractable market-based mechanism for our resource infrastructure, there are several implementation challenges that remain. A system that supports a market-based policy must provide a mechanism by which users can communicate their preferences for resources or their utility for a particular job. Also, such a

market-based system must provide mechanisms which can ensure that users *will* provide this information truthfully. Collecting this information and enforcing this behavior can be both computationally expensive and inexact.

Currency

Market-based approaches assume the existence of a mechanism to prevent self-interested users from artificially overstating their job priority or utility. Our example of an auction assumes the existence of a universal currency, and indeed, this is the same assumption used in a market. When deploying a market-based system, we must consider how currency is created, distributed, and used.

Currency as a Mechanism. Currency plays a central role in dictating user behavior. The theory we have discussed thus far assumes that users a utility-maximizing, and thus have an incentive to spend and save their currency based upon their preferences. Indeed our auction example illustrates how this idea is used to elicit private valuation information from a user. Unfortunately, real currency isn't always an option in real systems, particularly in scientific or research infrastructures where resource access is designed to be free.

In these settings, use of a *virtual* currency is proposed as an alternative to real money [53]. Virtual currency is completely generated and accounted for by the system, and is designed to serve the same purpose as its real-world counterpart. Unfortunately, a user with virtual currency does not necessarily behave in the same way as a user with real currency. Consider a scenario with two users who each have the same budget of currency, but have different demand (i.e., the first user has more jobs to submit than the second user). Without loss of generality, we can assume that each user otherwise has the same private valuation for each job. If using virtual currency, neither user will have incentive to save money, and thus the user with fewer jobs will be able to express a higher willingness to pay relative to the other user, despite the fact that they both have the same private valuation information. This problem does not occur with real money because consumers in a fully-functioning market have an incentive to save money to spend on other goods, and therefore, a utility-maximizing user would not choose to "throw money away". In a market with virtual currency, the actions of spending and saving money is isolated, and thus actual user behavior may differ from the expected utility-maximizing behavior. Therefore, using currency as a mechanism to elicit the

desired information poses an important challenge.

Currency as a Policy. Assuming that we have a sound currency mechanism which creates the desired incentives, there is the challenge of deciding how to distribute currency among users. Ostensibly, a user with a larger endowment of currency will be able to pay more for consumption, and thus state a larger willingness to pay for resources. Assuming that the currency is still spent and saved inline with a utility-maximizing user, choice of currency endowment might not affect the aggregate utility of the system. For example, there may be several efficient allocation outcomes. In fact, the second fundamental theorem of welfare economics (see Mas-Colell *et al.* [75] for original citation) states that any of these outcomes can be attained by changing the initial endowments of users. However, as we noted earlier, the desired metric of market-based schedulers is to maximize aggregate utility, rather than Pareto efficiency, so this result does not directly apply. We mention this result merely to illustrate the potential impact choice of currency distribution can have on the allocation efficiency. In this dissertation, we will assume that chosen endowments are exogenously determined, and will not explicitly consider the problem of how to perform these assignments. In practice, these assignments are often assigned by an external entity; for example, the National Science Foundation (NSF) assigns service units (credits) to research projects that wish to use NSF-sponsored computing clusters.

Perfect Information

Another basic assumption made in both mechanism design and general market equilibrium is that the consumers (in this case, users and applications) are have symmetric (if not perfect) information about resources.

Utility-maximizing users are assumed to know their own private information. This does not seem like an unreasonable assumption, but when applied to computational markets, it is clear that this assumption can break down. As discussed earlier, users may need to solve a computationally difficult problem to determine private valuation [63], and instead of doing this, users may be more comfortable making an approximate decision [23]. On the other hand, some users simply cannot pre-determine their value for a job, or value from use of a resource until after the fact. For example, users of high-performance computing systems are notorious for being unable to even estimate their job run-time length, even when given incentive to do so [66]. Therefore, even honest and

motivated users may not accurately reveal their private information to a sound elicitation mechanism.

Secondly, users are assumed to have symmetric common value information about market goods. With respect to a computational market, this information may mean characteristics of market items, such as resource availability, capacity, and performance. In many large-scale systems, resource characteristics are known to exhibit highly variable temporal characteristics (particularly when demand is high, which is precisely when the information is most needed [82]), which may make this information inconsistent across users depending on their vantage point [120]. If this information is inconsistent, then a user's ex-post satisfaction for resource consumption (how happy they are after an allocation) may differ greatly from their ex-ante strategy (e.g., bid in an auction, or determination of willingness to pay for an item prior to actual purchase); inconsistencies of this nature may reduce the aggregate utility (i.e., ex-post utility of users) delivered by the system.

Identifying Client and Service Provider Needs

The effectiveness of any market mechanism is dictated by its ability to extract *meaningful* utility information and use it to make relevant decisions for both clients and providers. For example, users may have complementary resource needs, and if a provider only allows preferences for resources to be stated independently — as in much of the existing market-based designs — then the allocation of the market may not be ex-post efficient. However, supporting such complementarity increases the computational complexity of the allocation algorithm and may decrease the efficiency of the system (i.e., by using a heuristic algorithm in lieu of an optimal algorithm), therefore, it is also clear that a practical market-based system cannot provide prices for every conceivable resource configuration.

Furthermore, many resource infrastructures are sponsored by service providers who themselves are clients of underlying resource providers. Many proposed market mechanisms deal with this scenario: reserve prices and combinatorial exchanges [84] extend the allocation problem as a two-sided exchange where the optimization criteria is to maximize the utility of all parties, rather than simply the buyers. As we mentioned in the context of mechanism design, this model presents a significantly more challenging setting in which to design robust mechanisms. In practice, the uncertainty which under-

lies the time-varying availability or cost of resources may make it difficult for resource providers to use these traditional mechanisms. Balancing the feasibility of a market design with its utility to users and resource providers will be a significant challenge in a live deployment.

2.3.3 Summary of Related Work

There is a large body of related work in the general area of market-based allocation systems, with the focus either being on the mechanics of the market itself, or on using user preference information to improve scheduling decisions. We briefly discuss each type of approach, but defer detailed discussion of related work to each specific chapter.

There are many designs for computational markets for resource scheduling, such as a time-shared PDP system [104], parallel systems [49, 62, 84, 103, 109, 116], data centers [18] and Internet computing systems [69, 94] (see Yeo and Buyya for a more comprehensive taxonomy [117]). These systems represent seminal work in designing a market-based allocation system, and concentrate on providing mechanisms to establish equilibrium prices for resources in the market, thereby focusing on many of the theoretical design challenges of a computational market. These systems differ primarily in how they define resources and in their target applications. Our work is inspired by many of these designs and applications, but our focus is primarily on addressing practical issues, such as the implementation challenges we describe earlier in the chapter.

On the other end of the spectrum, there are also several designs for *utility-based* allocation systems. This body of research uses per-job utility functions to specify the client's value of job completion: the Alpha OS [28] and Millennium [25] handle time-varying utility functions; Chen *et al.* [21] discuss how to do processor scheduling for them; Lee *et al.* [65] look at tradeoffs between multiple applications with multiple utility-dimensions; Muse [20] and Unity [110] use utility functions for continuous fine grained service quality control; and Petrou *et al.* [88] describe using utility functions to allow speculative execution of tasks; Bailey Lee *et al.* and Kelly [59] consider different algorithmic approaches a genetic algorithm approach to solving the intractable task assignment problem, and Irwin *et al.* [54] considers the problem of job admission control in the face of uncertainty in future job demand.

The distinction between this category and a pure market-based system is that

these systems do not consider traditional market issues such as currency policy and user incentives. Nonetheless, this prior body of work provides valuable insight in developing practical heuristic algorithms based upon market principles in order to use domain-specific utility functions to allocate resources in large-scale systems. Our work differs from this body of work in that we also concentrate on market-related issues, but leverage this existing work to improve practical scheduling algorithms.

2.4 Our Approach

We have discussed at length the potential benefits a market-framework can provide over existing scheduling approaches, and the challenges to designing and implementing such a system, but without a rigorous empirical analysis of a deployed market-based allocation system, we cannot begin to quantify the expected benefit or actual challenges from a market-based approach. Our approach follows an iterative process of design, implementation and deployment of a market-based system, and based upon lessons from an initial deployment, we present empirical analysis and refined design for a practical market-based allocation system for today's emerging resource infrastructures.

In the next chapter, we discuss two real systems that we have design, implement and deploy for real users of two large-scale resource infrastructures. Based upon our experiences and data from this work, we reveal limitations in existing market-based designs, and offer suggestions for ways to improve them. In subsequent chapters, we present trace-based simulation studies of these suggestions, an extension of a market-based design to support commercial service providers, and conclude with an implementation of a market-inspired mechanism in a time-sharing system.

Chapter 3

Case Studies

This chapter presents two case studies of market-based mechanisms designed to support sophisticated usage policies in real computing environments. First, we present Bellagio, a market-based resource allocator designed for the worldwide PlanetLab research network. Second, we describe Mirage, an allocator deployed on the Intel Research Berkeley sensor network testbed. We discuss the initial challenges these systems are designed to address and our deployment experience with each. We conclude with a set of suggestions to help guide future deployments of market-based resource allocation systems.

3.1 Motivation

PlanetLab [90] and the Intel Research Berkeley (IRB) sensor-network testbed are two early examples of large-scale federated resource infrastructures which were unable to handle periodic peaks in resource demand. Early on, each of these systems employed a traditional best-effort proportional-share and FCFS, scheduling policy, respectively. Since users have no incentive to restrain resource consumption, these systems typically suffer from a "tragedy of the commons" for prolonged periods of time, where unconstrained usage lead to a dramatic decrease in the accessibility and usability of resources [24, 47].

In early 2004, we were given the opportunity to build and deploy a market-based mechanism to address these issues. At a high-level a market framework is attractive to system administrators to encourage users to restrain resource consumption during

times of peak demand, and at the same time, give the system a framework to prioritize jobs when resources are severely constrained. We designed, implemented and deployed a market-based allocation mechanism on each of these systems.

Similar to many prior market-based designs, our allocation mechanism uses an auction, which requires domain users to express their resource preferences as a bid, which is subsequently used by the system to prioritize demand. We employ additional mechanisms to address some of the challenges not addressed by previous systems, such as resource discovery, a rich bidding language, and implementation of a currency (see diagram in Figure 3.1; we will describe each of these components in more detail). While our systems are based upon a set of shared design principles, each of these components are designed to work within their specific domain of PlanetLab or the IRB sensor network. We summarize the results below.

3.1.1 Summary of Results

The primary contributions of this chapter is the implementation and deployment of Bellagio and Mirage. Since they are two of the first large-scale market-based systems to have been implemented for real users, we have several new design choices, system components and results that contribute to the existing literature of market-based designs. These contributions can be summarized as follows:

- **User-driven design choices.** To address the user needs for simultaneous resources, we depart from traditional single-item auction-based resource allocation systems and use a combinatorial bidding language to allow users to express complementary and substitutable preferences.

- **Support for symmetric information.** To address the challenge of asymmetric resource information, we use a resource discovery mechanism to allow users to state an *abstract* resource preference, and aggregate information about the system to resolve these requests into physical resources upon run-time.

- **Repeated combinatorial auction heuristic algorithm.** In order to support a combinatorial bidding language and mechanism for providing symmetric information, we design a fast heuristic allocation algorithm as a necessary alternative to the computationally complex optimal algorithm.

- **Virtual currency policy.** We provide a distribution and collection infrastructure for a virtual currency. We support different endowments of currency (initial balances to support exogenous priorities), and a taxation policy to promote stability in currency distribution over time.

- **Empirical data.** We observe predicted, unanticipated, and unwanted user behaviors, including utility-maximizing, opportunistic and naive actions from users; we see evidence that users exhibit distinct preferences for resources and that an auction can elicit this information. However, there is also a cost imposed on users of the auction-based mechanism.

Next, we discuss the general system architecture for our market-based system, and the specific instantiations (Bellagio and Mirage) in subsequent sections. We conclude with our deployment results and lessons learned.

3.2 System Design

There are two primary shortcomings of using simple proportional-share and FCFS allocation policies that we wish to address with our deployments. First, users reside in different administrative boundaries, so it is difficult to statically assign priorities to users and jobs when resources are constrained. Second, users lack both an incentive and mechanism to restrain their consumption behavior during times of peak load. Designing Bellagio and Mirage as market-based allocation systems is a natural framework with which to address both of these challenges.

In constructing our markets, we have two critical design decisions: how to structure the market, and how to handle currency. In this section, we describe the high-level design used by both Bellagio and Mirage: the basic market mechanism used for allocation, and the implementation of currency used in both systems.

3.2.1 Market Mechanism

Economists have long proposed pricing as a mechanism to decentralize resource allocation [50, 51, 58, 100, 72], but it is not clear how to set prices in our particular market. The fundamental role of prices in our system is to change user behavior (i.e., shift demand) according to resource supply and demand, and force users to self-select who can afford a particular resource at any point in time. There are two significant

challenges to setting prices in PlanetLab and the IRB testbed. First, price is a function of demand, which is often hard to predict. In the absence of steady demand [26] and good prediction, prices may fluctuate wildly. Second, users require combinations of resources (e.g. CPU, network bandwidth, memory, etc.), but it is not feasible to price all possible combinations. Taken together, these two issues place considerable uncertainty on a user, who must anticipate both the price changes over the course of a job and the changes in the level of contention on resources, and repeat this process for each resource in a desired bundle.

Recognizing that, while demand volume and user values are typically unknown, and capacity is known, an *auction*-based approach turns this problem around. Auctions can schedule resources for individual jobs while accounting for the per-job utility expressed in user bids. Clearing prices can be determined in terms of the demand information implicit in a set of user bids. Auction-based approaches for resource allocation in computer systems have been explored by a number of previous efforts across a broad range of distributed systems including clusters [25, 109], computational Grids [62, 116], parallel computers [103], and Internet computing systems [69, 94].

However, these existing auction-based approaches are limited to allocating a single type of good, and to allocating a single good at once. In the case of SPAWN [109], this is CPU shares of a single machine, and in Tycoon [62], each resource (CPU, disk, memory and network bandwidth) is allocated independently. As with these systems, our designs also rely on an auction to allocate resources. However, we adopt the model of a *combinatorial auction*, in which users can bid on *bundles* of resources as opposed to individual resources. This ability to bid on resource combinations in space and time allows users to more accurately express their preferences on different resources. For example, a user can express that some resources are interchangeable ("substitutes") and some are required together ("complements"). To the best of our knowledge, Bellagio and Mirage are the first deployed systems that support allocation of combinations of heterogeneous computational resources via flexible auction methods.

The goal of our auction is to determine an allocation that maximizes user utility (as measured, for example, in units of currency). If the market is reasonably competitive and bids reflect the true utility of a user for a particular allocation, then by clearing a market to maximize the total bid value the market will tend to achieve our goal of maximizing aggregate utility.

Implementation

Both Bellagio and Mirage systems use a repeated combinatorial auction[1], which runs an auction periodically, and allocates resources to competing users based on the results of the auction. As we discuss earlier, the problem of determining winners in a combinatorial auction is computationally intractable, so we adopt a greedy algorithm to ensure that the auctions clear quickly, irrespective of the number or complexity of user bids. This heuristic orders bids by *value density*, which is defined as the bid value (b_i) divided by its size (s_i) and length (l_i). In order to minimize the risk of settling in at a local minima, our algorithm compares the outcome of k different ordering of the top k bid value densities, and selects the best allocation among these k attempts. In our deployment we set $k = 10$. Also, we adopt a first-price auction format rather than a Vickrey-style auction, in part to make auctions clear as quickly as possible. As we mention in Chapter 2, a first-price auction is not incentive-compatible; however, we are unaware of any efficient, incentive-compatible *and* computationally-tractable mechanism that supports combinatorial preferences. Furthermore, it has been observed that the advantages of an incentive-compatible VCG mechanism can break down when moving from a static setting to our repeated setting [78]. In this implementation, our primary goal is to maximize aggregate utility, and, as we also mention in Chapter 2, we do not require that our auction mechanism be *strictly* incentive compatible, merely efficient[2].

In addition to requiring multiple resources, users in PlanetLab and the IRB testbed are often interested in a *class* of resource as opposed to a unique resource. For example, a PlanetLab user may simply want any 10 machines, as long as each has at least 1 GB/s of available outbound bandwidth, or an IRB testbed user may want any 20 sensor nodes as long as they will not interfere with particular frequency bands. Since these properties may change over time, or may be difficult for a client to determine, we provide a *resource-discovery* service to assist user resource selection. Users specify the type of resources they are interested in by expressing an abstract resource specification. The resource discovery service then maps these specifications to concrete resources that meet the desired constraints. We use this level of indirection for three reasons. First, it frees the user from having to manually identify candidate resources. Second, it allows

[1]The initial deployment of Mirage used an open-bid English auction format, which was later changed to a sealed-bid format for reasons we will discuss later; Bellagio uses a sealed-bid format.

[2]As we will see later in the chapter, it may be difficult to achieve efficiency without also providing incentive-compatibility.

users to automatically take advantage of new resources as they are introduced into the system. Finally, as mentioned, testbed users are frequently interested in acquiring sets of nodes and are often indifferent to which specific nodes they are allocated as long as the candidate resources meet their constraints. The resource discovery service allows users to discover all possible candidates and thus provides the system with the maximal amount of information on substitutes when clearing the auction.

Our system continually accepts resource bids and runs the auction on collected bids periodically (in our deployments, we choose once every hour, which we will discuss later). Once bids are received the auction must determine winning bids and the associated resource allocation. A user in our systems submits bids to the auction using a two-phase process (Figure 3.1). First, she adopts a resource discovery service to find candidate resources that meet her needs. Second, using the concrete resources identified from the first step, such a user can place bids using a system-specific bidding language.

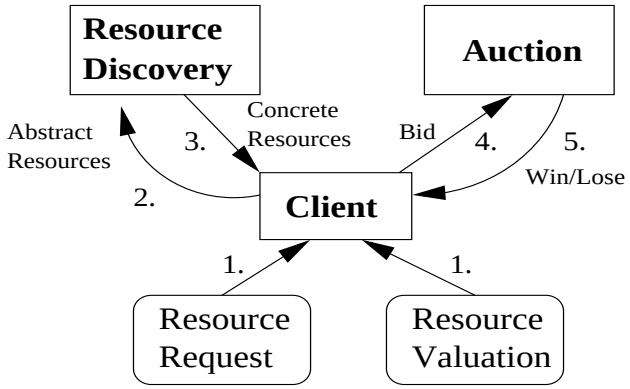


Figure 3.1: Bidding and Acquiring Resources.

3.2.2 Currency System

As we mention in Chapter 2, currency is used to create incentives for globally desirable user behavior. In PlanetLab and the IRB testbed, currency is the token with which users bid in the auction, and since users have a finite budget of currency, they will have an incentive to save and spend it according to their own needs. However, like most existing computing systems, neither PlanetLab nor IRB have any existing notion of user currency. And because introducing real currency isn't a feasible option in either deployment, we design a *virtual* currency system for use in both Bellagio and Mirage.

The design of a good currency policy encourages desirable behaviors and discourages undesirable behaviors. The lack of proper policies can render the system useless. For example, if users can obtain large amounts of virtual currency very easily then they will of course bid arbitrarily high values all the time. Such a system reduces to resource allocation based on social conventions, since there is no disincentive for a user to not always bid the maximum possible value permitted within the bidding language. Simply stated, the currency must be "real enough" that users care not to spend it because they will be able to benefit from saving unspent currency for future use. In order to support virtual currency, we rely on a central bank that enforces a currency policy by controlling the aggregate amount and flow of virtual currency in the system.

Since users have no direct way of earning virtual currency, the system must decide how to distribute the currency. Because users enter the system with no virtual currency, we also need to provide new users with some initial amount of currency. In addition, as users spend currency over time, we also need a way to infuse their accounts with new currency. Clearly, there are many virtual currency polices one could employ to meet these requirements and different policies will result in very different economic systems and resource allocations.

Our virtual currency policy is based on two principles: (i) prioritizing users based on an exogenous policy (we will describe the specific policies for Bellagio and Mirage in the next sections); and (ii) penalizing/rewarding users based on usage or lack of usage during times of peak demand. Examples of factors that may affect prioritization include the types of usage (e.g., research vs. coursework), the amount of contribution made to a system, and fairness concerns, which may be defined in terms of the cumulative resource consumption of an individual. Allowing the prioritization metric to be determined exogenously permits flexibility in crafting a good policy for a specific domain. In addition, it seems natural to reward the user who refrains from using the system during times of peak demand (or, more generally, does not waste resources) and penalize the user who uses resources aggressively when resources are scarce. Consider a user who monopolizes the entire system for several days prior to a major deadline, for instance.

Each user is associated with an account at a central bank which stores virtual currency. Each bank account is assigned a baseline amount of currency based on priority, and a number of currency shares which influences the rate that currency will subsequently flow into the account. Given an initial currency allocation, users can begin to bid for

resources in the auction. Each time the auction clears, trades are settled and revenue is collected from the accounts of winning bids. This currency is then distributed back to all accounts through profit sharing in a proportional-share fashion based on the number of shares in each account (Figure 3.2). It is this profit-sharing policy that allows users who do not waste resources to save additional currency which can be used for a subsequent burst of activity later.

In addition to profit sharing, the system also imposes a fixed-rate savings tax on all accounts that have excess currency above their baseline values, again with proportional-share distribution. The motivation for the savings tax is based on expected resource consumption. For example, in PlanetLab (and also other distributed systems testbeds), resource consumption is often highly imbalanced with a small fraction of the users consuming the majority of the resources and many users often going idle for long periods of time [26]. Similarly, parallel batch computing systems often exhibit a diurnal load pattern varying between extremes of light utilization and heavy utilization [27]. Assuming similar resource consumption patterns, and in the absence of additional policy, the implication would be that heavy users would eventually be working out of accounts with very little currency even if there is little demand for resources in the system. To mitigate this effect, we impose a fixed rate savings tax that makes operational the concept of "use it or lose it" policy employed by agencies such as the Federal Aviation Administration in allocating scarce resources; i.e., it is fine to defer the consumption of currency for a while but at some point it is desirable to allow others to gain the benefit of resources that a user is not using by redistributing some of the currency via the savings tax. Users should be rewarded for not wasting resources, but such a reward should not last forever. In the absence of any activity in the system, the savings tax works such that all accounts eventually converge back to their baseline values. The savings tax is collected every 4 hours, at a rate of 5% of an account's savings. These parameters were chosen such that an exhausted bank account can recover half of its balance within a few days, and the full amount in a week [3].

By controlling the distribution of wealth in a virtual economy, the system is able to indirectly control the share of resources received by individual participants over different time-scales, despite artifacts introduced by the virtual currency and the continued

[3]Kash, Friedman and Halpern have recently initiated a research agenda on developing a theory for how to allocate virtual currency within "scrip" systems in which agents both contribute and consume resources.

presence of load imbalance. In order to control the distribution of wealth among users, there are two policies that we must determine: how will users be able to earn virtual currency, and how (if at all) will the system limit the wealth of users. We discuss the policies chosen for our specific implementations, Bellagio and Mirage, next.

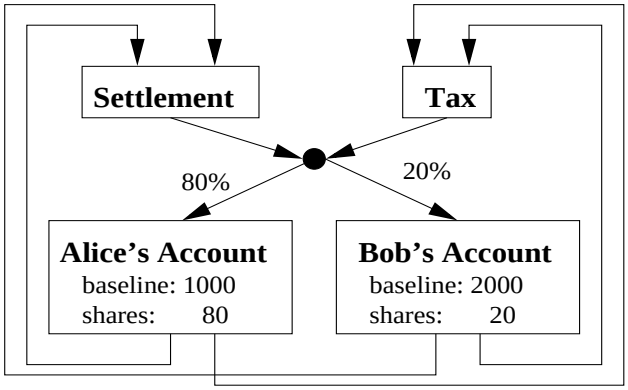


Figure 3.2: Virtual currency policy.

3.3 Bellagio

In this section we describe the Bellagio architecture and implementation on PlanetLab.

3.3.1 Target Platform

PlanetLab is a distributed, wide-area, federated testbed consisting of over 800 machines hosted by more than 400 sites across the world (Figure 3.3). Machines are owned and operated locally by each site, but primarily administered centrally by PlanetLab Central (PLC). PLC performs access control and maintains a consistent software image on each machine. Each site is in charge of the physical upkeep of its machines, such as uptime and providing adequate and persistent network bandwidth. Additionally, each site pays a periodic fee to PLC to support the central administration. In return, a site's members, such as research scientists and graduate students, are granted access to use any machine in the system. Each user can obtain a *slice*, which grants her access to the resources of any PlanetLab machine. The instantiation of a slice on a particular machine is called a *sliver*. A sliver shares basic information about its associated slice, such

as authentication keys and basic configuration files, and is the physical manifestation of the user's isolated environment on a machine. A sliver is therefore intended to export an interface similar to that of a virtual machine or other virtualization technology. As of this writing, PlanetLab implements virtualization using the Linux V-Server technology.

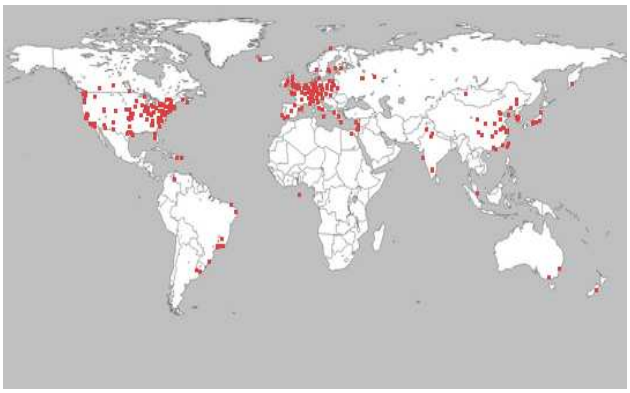


Figure 3.3: Layout of PlanetLab resources as of 2008.

A typical scenario for a PlanetLab user is that she will first obtain a slice from her local administrator (by way of PLC). Once she has a slice, she creates slivers on any number of machines. The most common use for the machines is as part of a wide-area network experiment, with each component machine acting as an end-point or providing some other function within the experiment. The value in these machines is not only in their local resources, but their geographic location; a wide-area configuration of machines provides a realistic setting for many distributed systems and network experiments. At any point in time, a particular machine can multiplex its resources across processes from many different slivers. These tasks can vary in length (i.e., a few minutes to months) and in size (i.e., a few machines versus all machines). Each machine's resources are time-shared among active slivers using an equal-weight, proportional-share scheduling policy. For example, if there are n active slivers on a machine, each active sliver is expected to receive a 1/n*th* time-slice of a machine's CPU cycles and network bandwidth over every scheduling window. This type of scheduling is termed *work-conserving* [60], since resource access is multiplexed among only *active* slivers. PLC performs limited admission control such that the number of active slivers on a machine is limited to n = 1000 (i.e., no more than 1000 simultaneous slices can run on a machine). This constraint is imposed due to the physical limitations on memory and disk space on each machine.

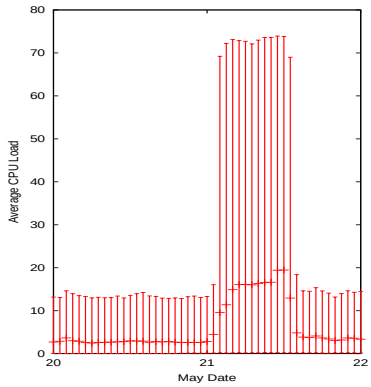


Figure 3.4: 5th, average and 95th percentile load on 220 PlanetLab nodes leading up to the May 2004 OSDI deadline.

The need for a more sophisticated resource allocation policy became apparent during deadlines for the major systems and networking conferences in 2003 and 2004, when resource contention became a persistent problem in PlanetLab. Figure 3.4 illustrates a (then) common occurrence for PlanetLab users. Due to synchronized conference paper deadlines among users, the system experiences large spikes in resource demand in the days leading up to a deadline. The proportional-share resource allocation policy provides users with little control over their share of resources during such times, little incentive to reduce consumption, and even less means to coordinate usage with one another. The end result is that significant resource contention leaves the system unusable for the majority of users. Bellagio is designed to help address this problem.

3.3.2 Architecture

Bellagio makes allocation decisions using a combinatorial auction. We provide additional mechanisms on PlanetLab to allow users to bid for these resources. In this section, we describe these mechanisms.

Resource Discovery

The decision to use a combinatorial auction rather than a series of single-item auctions is motivated by the fact that most users on PlanetLab require simultaneous use of a large number of machines. Selecting these machines poses a challenge to users because there is a large set from which to choose, and machines — while largely homo-

geneous in resource capacity — often exhibit dynamic characteristics. For example, a typical usage scenario involves selecting machines that exhibit low load and an abundance of network capacity. In addition, a user might want machines that have a particular physical network topology of interest, such as within the same continent, or explicitly spanning continents.

Distributed resource discovery allows end users to identify available resources based upon both of these characteristics. Bellagio uses SWORD [81] for this purpose, which is a resource discovery service implemented on PlanetLab. It exports an interface that allows PlanetLab users to locate particular resources matching various criteria. For example, users can search for resources based on resource-specific attributes (e.g., machines with low CPU load and large amounts of free memory), inter-resource attributes (e.g., machines with low inter-node latency), and logical (e.g., machines within a specific administrative domain) or physical attributes (e.g., geographic location). SWORD returns a set of candidate machines matching the user's description.

Once users submit a bid, there may be a substantial delay between bid submission and the actual allocation. Because resources can exhibit dynamic temporal characteristics, it might not be useful for a user to bid on specific machines that she has discovered from SWORD. Instead, the intended use of the resource discovery mechanism is to express abstract resource specifications. Therefore, Bellagio alternatively allows a user to specify a SWORD resource specification (i.e., a SWORD query) in her bid instead of providing details on specific machines, and Bellagio will resolve these queries immediately prior to determining a resource allocation for users.

Bidding Language

The previous section describes how resources bundles are identified by users. This section describes how users then communicate demands for resource bundles to Bellagio. Designing a bidding language for Bellagio involves a number of trade-offs. First, the expressiveness of the language, such as size and types of allowable bids directly affects the computational complexity of the allocation algorithm. On the other hand, having a language that is both expressive and concise for users can make their life simpler. One consideration of relevance within Bellagio is the frequency with which auctions must be cleared. Very frequent auctions would require a restrictive bidding language with simple allocation problems while less frequent auctions would allow for a more complex bidding

language and hard clearing problems. Of course, less frequent auctions would also make the system less reactive and less useful for very impatient users.

The particular choice of parameters is hand-tuned based on studies of PlanetLab usage and our own experience. The end result is a 4-hour time slot of CPU share as the basic unit of allocation in Bellagio. The combinatorial auction correspondingly clears every 4 hours. In each period, there is a rolling window of $T = \{1, 2, 3..., 42\}$ time slots forward from the current time in the auction (where $T = 1$ denotes the immediately next time slot that will be available). We let N denote the set of resources available to allocate (i.e., different PlanetLab nodes). Note that immediately prior to running the auction clearing algorithm, any bid that contains an abstract resource specification (SWORD query), Bellagio translates the resource specification to a set of concrete candidate machines. The bidding language allows a user to specify a required allocation duration from the set $D = \{1, 2, 4, 8\}$, which represents the number of time-slots to reserve. Taken together, a bid b_i from a user is constructed as follows:

$$bid = (s_0, t_1, d, \{n_1, n_2, ...\}, \{ok_1, ok_2, ...\}, v),$$

where $s_0, t_1 \in T$ denote the range of possible start times for the bid to be valid, $d \in D$ is the duration of the job, $\{n_1, n_2, \ldots, n_k\}$ and $\{ok_1, ok_2, \ldots, ok_k\}$ denote the required quantities $n_l \geq 1$ on resource equivalence classes $ok_l \subseteq N$. An example of an equivalence class is $ok_i = \{*.\texttt{princeton.edu}\}$, where ok_i represents the subset of PlanetLab nodes located in the `princeton.edu` domain. The corresponding n_i parameter represents the desired quantity of nodes satisfying this equivalence class. $v \geq 0$ is the bid price (willingness to pay) for any bundle of resources that satisfies this bid. These parameters allow PlanetLab users to reserve CPU shares for a duration of up to 32 hours, and up to a week in advance. For example, a user might request "any 10 nodes from Princeton, and any 10 nodes from Berkeley, for 32 consecutive hours anytime in the next 24 hours." The corresponding bid of value v would be:

$$bid = (1, 6, 8, \{10, 10\}, \{\{*.\texttt{princeton.edu}\}, \{*.\texttt{berkeley.edu}\}\}, \texttt{v}).$$

Currency Policy

The currency distribution policy in Bellagio is performed on a per-organization basis. Each site begins with an initial balance of virtual currency that is proportional to

the number of machines contributed to PlanetLab. This policy is designed to encourage contributions to PlanetLab, and in particular to reward those who contribute the most. New sites that join PlanetLab are assigned currency in this way, as this initial balance is only way that currency can be introduced in the economy. In PlanetLab, individual users and slices are associated with a particular site or institution. All users and slices associated with that institution have rights to the site's bank account. When a user makes a bid, her balance is temporarily frozen into an intermediate Bellagio account. Each time an auction clears, revenue is collected from the accounts of winning bidders.

As described earlier, each bank account is also associated with a currency share which determines the site's portion of profit sharing. Initially, the currency share for a bank account was the same as its initial balance — the number of machines their site has contributed to PlanetLab. However, we modified this policy upon observation that certain machines in the system are used much more often than other machines. We want a policy that rewards organizations that contribute the most valuable (highly utilized) resources. Therefore, we revise profit sharing such that it is only divided among each of the organizations that owns a machine in a winning allocation; profit (revenue) from each auction-clearing period is funneled directly back to the account of the PlanetLab organization that owns the machine. One implication of this policy is the promotion of long-term system growth. Users wishing to receive a larger revenue share will now have an incentive to contribute useful resources to the shared infrastructure.

Once the payment amount for each winning bid has been determined, the user's bank account is charged by the appropriate amount, and resource bindings are performed by returning resource capabilities in the form of tickets to the winning bidders using a system such as SHARP [47].

3.3.3 Deployment

In the Summer of 2004, PLC provided the opportunity for research groups on PlanetLab to experiment with more sophisticated scheduling policies. Beginning in December 2004, timed with the release of PlanetLab Kernel version 3.0, Bellagio was given a fraction of each PlanetLab machine's CPU resources, which it could then allocate to different user slices. For example, if a machine has n active slices, each slice receives 1/n time-share of the machine's CPU. A full allocation of Bellagio's share of resources to a single slice would provide the slice with 3/n+2. For large n, the relative boost a slice

could receive through the market is 200%. Therefore, for a single machine, the full allocation of Bellagio's resource can double or triple a slice's relative CPU scheduling share. We stress that the use of Bellagio was strictly optional: PlanetLab users automatically received a default proportional share allocation; our system only sells an increase in these soft resource shares. This particular resource boost is an artifact of the scheduling technology used by PlanetLab, and the amount of resources that we are provided to allocate within our market.

Because we are introducing a market-based allocation scheme to a live system which previously had no notion of markets, we anticipated the learning curve for new users would be high. The Bellagio user interface is designed to encourage early adoption by the PlanetLab user community and promote frequent use. In order to lower the barrier to for the user community, every registered PlanetLab user is automatically registered to use Bellagio; a user simply supplies her authentication credentials to the Bellagio Web interface, which are then verified against the central PlanetLab database. The Web interface is deployed on a cluster of Linux machines with a PHP front-end and a PostgreSQL database back-end. The database contains account balances for virtual currency and manages user authentication.

Once authenticated by the Web interface, a user can view the status of her account and previous bids and allocations. To facilitate the bidding process, we provide several useful guides. First, to address the issue of "valuation uncertainty", which is an issue related to whether or not users will understand how to value resources in units of the virtual currency, we provide historical prices in Bellagio as a reference point to the current level of demand in the system. Second, to help formulate bids, we provide several "one-click" bidding options, such as "bid on any N nodes in my slice" or "any N nodes from K distinct Autonomous Systems". Users also have the option of formulating complex queries by defining equivalence classes of resources on which to bid. And, as mentioned, Bellagio provides an interface to SWORD to perform these queries, which is a tool familiar to many PlanetLab users.

In the next section, we discuss the Mirage platform and implementation, before proceeding to discuss the deployment experience of each system.

3.4 Mirage

In this section we describe the Mirage architecture and implementation on the Intel Research Berkeley sensor-network testbed.

3.4.1 Target Platform

The initial motivation for this work became apparent during the construction of a 148-node testbed at the Intel Research laboratory (IRB) in Berkeley, CA (Figure 3.5). This testbed is comprised of 97 Crossbow MICA2 and 51 Crossbow MICA2DOT series sensor nodes, or motes, mounted uniformly in the ceiling of the lab. The motes incorporate an Atmel ATmega128 8-bit microcontroller, 4KB of RAM, 128KB of flash memory, and a Chipcon CC1000 FSK radio chip. The MICA2 series devices in the testbed operate in the 433 MHz ISM band and incorporate a sophisticated sensorboard that can monitor pressure, temperature, light, and humidity. The MICA2DOT devices operate in the 916MHz ISM band and do not include sensorboards.

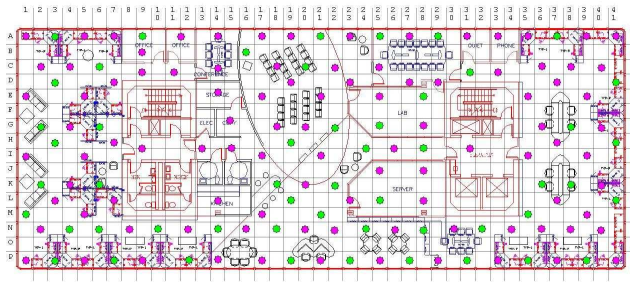


Figure 3.5: The Intel Berkeley lab testbed layout.

Users of a sensor-network testbed such as IRB's are frequently interested in acquiring combinations of resources; e.g., resources that meet certain constraints. For example, consider a machine learning researcher who is interested in testing distributed inference algorithms in sensor networks. Such a user might be interested in evaluating algorithms at a moderate scale while performing inference over temperature and humidity readings of the environment. The user's code might also be tailored to a particular type of device (e.g., a MICA2 mote) and needs to run on a different, appropriately-spaced frequency to avoid crosstalk from other experiments. Thus, a user's abstract resource requirement might be something like "any 64 MICA2 motes, operating on an unused

frequency, that have both a temperature and a humidity sensor".

3.4.2 Architecture

As described in the Bellagio architecture, Mirage makes allocation decisions using a combinatorial auction. We describe in this section the methods that we provide in Mirage to allow users to bid for motes.

Resource Discovery

Similar to Bellagio, Mirage users can express an abstract resource specification to specify constraints on the types of resources they seek to acquire. For example, testbed users often need to specify constraints on per-node attributes. In the machine learning example earlier, for instance, a logical conjunction on per-node attributes (mote type and sensor board type) combined with a desired number of nodes is required. In other cases, constraints on inter-node attributes may be necessary. For example, a user might wish to acquire "8 motes where each pair of motes is at least 10 meters apart" to ensure that the network causes a multi-hop routing layer to form. Currently, Mirage supports resource discovery using per-node attributes including mote type, sensor board type, and supported frequency range.

Bidding Language

The bidding language in Mirage is similar to Bellagio's. The only difference is that Mirage users are also interested in allocating a radio frequency range ($[f_{min}$,$f_{max}]$) for their allocation. Mirage must allocate frequencies in such a way that does not generate crosstalk between user experiments. Formally, a bid b_i in Mirage is specified as follows:

$$b_i = (v_i, s_i, t_i, d_i, f_{min}, f_{max}, n_i, ok_i) \tag{3.1}$$

Bid b_i indicates the user wants any combination of n_i motes from the set ok_i (obtained through resource discovery) for a duration of d_i hours with a start time anywhere between times s_i and t_i and a frequency anywhere in the range $[f_{min}, f_{max}]$. The associated bid price is v_i, representing the units of virtual currency that the user is willing to pay for these resources. Continuing with the distributed inference example, a user thus might say: "any 64 MICA2 motes, which have both a temperature and a humidity

sensor, operating on an unused frequency in the range [423 MHz, 443 MHz], for 4 consecutive hours anytime in the next 24 hours". Suppose the user used the resource discovery service and found 128 motes meeting the desired resource specification and valued the allocation at 99 units of virtual currency. The corresponding bid in this case would be:

$$b_i = (99, 0, 20, 4, 423, 443, 64, \text{list of 128 motes}) \tag{3.2}$$

Mirage uses a greedy heuristic algorithm to compute the set of winning bids, similar to the algorithm used by Bellagio. Like Bellagio, the resources that a user of the IRB testbed cares about can exhibit both substitutes and complements. For example, in the machine learning example, the user does not care which specific MICA2 motes are allocated as long as a total of 64 of them are allocated. Hence, MICA2 motes are substitutes for one another. Similarly, the user does care that 64 motes are allocated simultaneously. A partial allocation of, say, 8 motes would not meet the user's needs in this case since the user's intention was to test at a moderate scale. (The extreme case would be a partial allocation of a single mote.) Thus, the 64 motes can be viewed as being complimentary to one another.

Currency Policy

As in Bellagio, each user is associated with a project that has an account at a central bank which stores virtual currency. Each project's bank account is assigned a baseline amount of currency based on priority, a number of currency shares, which influences the rate that currency flows into the account, and is initialized with a baseline amount of currency. Priority and currency shares are determined exogenously, and mostly based on the type of usage (e.g., research vs coursework). Most accounts have a baseline balance and currency share of 1000, while 2 local users have a balance and share of 2000. Note that the profit sharing policy differs from the one used in Bellagio. In Mirage, the sensor nodes are all owned by the IRB lab, and therefore, profit cannot be associated with a particular project, whereas in PlanetLab a resource can be directly attributed to a site.

3.4.3 Deployment

We deployed Mirage on Intel Research Berkeley's 148-mote sensor-network testbed in December 2004, and it is still in operation as of this writing. The implementation

is comprised of three types of components: clients, a server, and a front-end machine that users log in to and which provides controlled physical access to the testbed (Figure 3.6). Clients provide users with secure, authenticated command-line (the mirage program) and Web-based access to a server (`miraged`) which implements a combinatorial auction, bank, and resource discovery service. The server provides service to clients by handling secure, authenticated XML-RPC requests using the SSL protocol with persistent state stored in a PostgreSQL database. Lastly, the front-end physically enforces resource allocations from the auction using Linux's per-uid `iptables` packet filtering capabilities. By default, all users are denied access to all motes. Based on the outcome of the auction, rules are added as needed to open up access to users of winning bids for specific periods of time.

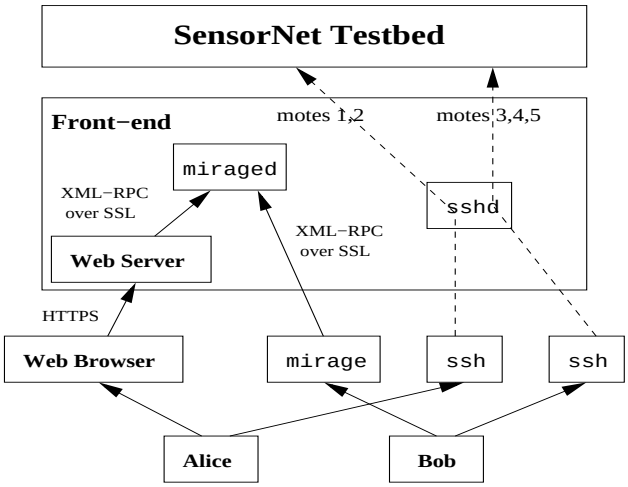


Figure 3.6: Mirage implementation.

The auction is parameterized by several variables: number of resource slots, resource slot size, and acceptable bid durations. Our deployment includes 148 motes where access to those motes is based on 1-hour slots, the minimum time unit of resource allocation. To accommodate users who might require a range of different times with the motes, users may bid for either 1, 2, 4, 8, 16, or 32 hour duration blocks. To allow users who wish to plan ahead (e.g., perhaps near a conference deadline), the auction will sell resources for up to three days into the future. Given our slot size of 1-hour, this works out to a total of 72 slots. Thus, we can view the resources being allocated as a matrix of 148 motes by 72 time slots. When the system boots, all slots are available. Over time,

slots become occupied as bids are allocated resources and new slots become available as the window of slots opens up over time.

To use the system, users register for an account at a secure web site by providing identifying information, contact information, a project name, and by uploading an SSH public key. Each user is associated with a project and each project has an owner. An administrative user is responsible for enabling accounts for project owners and assigning each project a baseline virtual currency value and a number of virtual currency shares. Project owners can subsequently enable their own users, thereby eliminating the administrative user as a centralized bottleneck.

Users bid securely in the auction using either the command-line tool mirage, which acts as an XML-RPC/SSL client, or through the web-based interface, where PHP scripts on the back-end act as XML-RPC/SSL clients to the relevant servers. The command-line tool provides full access to the entire RPC interface exposed by `miraged`. Use of this program is useful for various types of scripting and automation. To accommodate users who prefer a graphical interface, the web-based interface provides a simple, integrated interface to the system where users specify what resources they want and how much they are willing to pay using an HTML form. The web server, in turn, maps the user's abstract resources to concrete resources using the resource discovery service and places a bid in the auction on the user's behalf.

To use testbed resources, each winning bid results in members of the associated project being given access to a specific set of motes for a period of time specified in the bid. Motes are made physically accessible to project users through the front-end by doing the following for each project member: (i) creating a temporary Unix login on the front-end machine using a global username (MD5 hash of the user's SSH public key), (ii) enabling access to the front-end via SSH authentication using an SSH `authorized_keys` file, and (iii) setting up firewall rules on the front-end such that only the user can access the particular motes assigned to the winning bid.

3.5 Experiences

In this section we compare and contrast our deployment experience in Mirage and Bellagio, and discuss the lessons we've learned about a market mechanism based upon these two deployments.

3.5.1 System Usage

Our deployment experience in Bellagio was very different from that in Mirage. Bellagio was released for public use on February, 2005. Each site was given a balance of 100 units of virtual currency per node; the initial balance of sites ranged from 100 to 2200. During its lifetime, it saw *only* 23 bids from PlanetLab users representing 13 different organizations, but allocated a total of *242,372 hours* of CPU shares. In other words, there were very few users, but those users were very active in the system. In part due to this lack of activity, Bellagio was taken off-line in the Summer of 2005.

On the other hand, Mirage was deployed in December 2004, and it is still in operation as of this writing. In the first six months of use, 18 research projects were registered to use the system and 322 bids were submitted resulting in a total of 312,148 allocated node hours. Compared to Bellagio, Mirage had roughly the same number of users, but saw significantly more usage: more than 10x as much bid activity (the number of node hours is difficult to compare since PlanetLab has a higher capacity).

The primary difference between Mirage and Bellagio is the value of the resources available in the market. Participation in Bellagio was optional, as users were guaranteed best-effort resources through the standard PlanetLab interface. Thus users were able to choose between using "free" resources, or managing currency and using an auction to pay for additional resource shares. This lack of participation thus indicates an inherent user cost to using the auction mechanism in PlanetLab. On the other hand, participation in Mirage is mandatory in order to use the resources. Anecdotally, a few PlanetLab users who chose not to engage in the market did not want to tolerate the delay between submitting a bid and subsequently waiting for the auction outcome before being able to proceed with an experiment. Since PlanetLab applications are time-shared, users are able to run experiments immediately and use resources on-demand. On the other hand, in Mirage, users are accustomed to "waiting in line" for sensor nodes. Prior to Mirage, experiments were scheduled in batches, and the only significant burden imposed by Mirage is for IRB testbed users is to use the bidding interface. We see this dichotomy as an artifact of the opt-in nature of Bellagio, and not a negative result about the applicability of a market-based scheduler on the PlanetLab domain or time-sharing systems in general.

While Bellagio did not provide enough data to draw significant conclusions, we present data from our Mirage deployment and discuss the implications.

3.5.2 Discussion

The goal of both Bellagio and Mirage is to promote efficient allocation of resources, and in particular, guide allocation decisions when demand exceeds supply. In this section, we reproduce figures from an analysis performed by a colleague [78] on the data generated by our deployment of Mirage.

Figure 3.7 (reproduced from [78]) indicates that resource contention continues to persist on the IRB sensor network testbed, as it had prior to the deployment of Mirage. We can see that during times of heavy system load, the bidding process is able to allocate resources at a higher price (Figure 3.8, reproduced from [78]), and resolves contention by allocating the scarce resource supply to users who valued them the most. Figure 3.9 (reproduced from [78]) demonstrates that individual users place bids that range over four orders of magnitude. From this data, we can conclude that users indeed place different levels of priority on resources (assuming that the bids don't represent grossly misstated valuations) at different times, which suggests that extracting this information can improve the allocation efficiency of a system, i.e., the aggregate utility achieved by its user base.

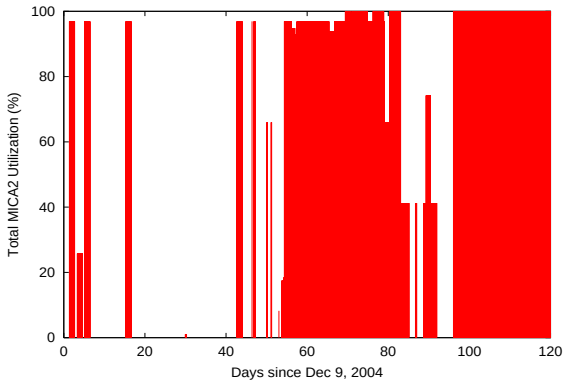


Figure 3.7: Testbed utilization for 97 MICA2 motes. [78]

Similar observations about resource valuation were made from our deployment on Bellagio; however the data is significantly less reliable because of the few data points available from deployment. As emphasized earlier, use of Bellagio was strictly optional to PlanetLab users. As a result, the market did not capture the full demand of the system. In particular, when overall system demand was low, there was virtually no activity on Bellagio.

This example also serves to illustrate that users will often choose an easier-to-user

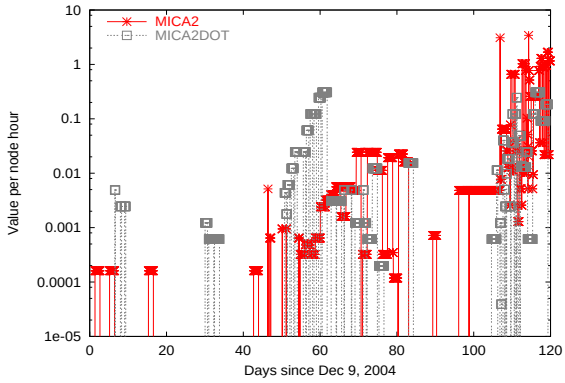


Figure 3.8: Median node-hour market prices. [78]

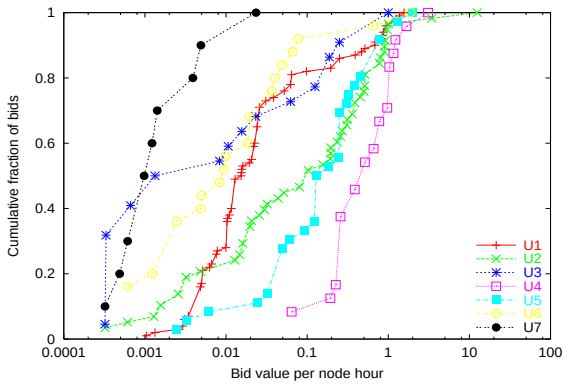


Figure 3.9: Bid value distributions by user. [78]

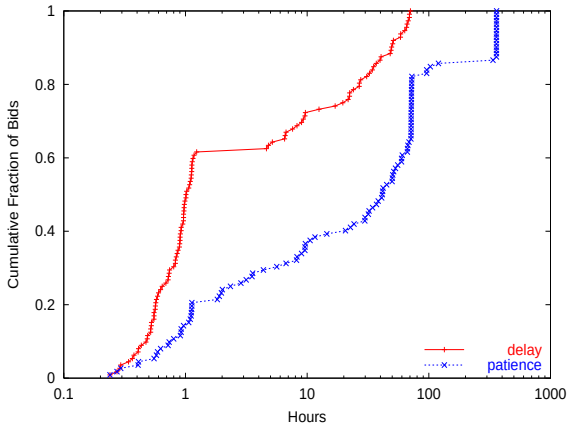


Figure 3.10: Distribution of delay and patience from bids submitted to Mirage from January 20 to March 22, 2005. [78]

approach, even if it results in slightly less utility. One particular burden of our auction-based approach is the artificial delay between node allocations. Since auctions are run every hour, allocations to winning bids are correspondingly delayed until the hour. There is a trade-off between imposing little delay on users and maximizing the efficiency of an allocation: increasing the frequency of auctions reduces the delay imposed on users, but it also reduces the amount of demand information captured in an auction, potentially reducing the quality of the allocation.

In Mirage, the auction-clearing period of 1 hour is designed to mitigate the impact of the imposed auction delay. However, based on the available data, we observe that a significant fraction of the bids from users could not be satisfied given the 1-hour delay. Each bid in Mirage includes both a delay and patience field, indicating an earliest start time and latest start time allowed for an allocation. These fields are used to communicate time-based constraints such as a deadline for an allocation. Figure 3.10 (reproduced from [78]) indicates that 10% of user bids could not wait for the 1-hour duration. This effect was more obvious in Bellagio, where a majority of PlanetLab users chose not to bid at all.

Bellagio and Mirage were both designed to reduce the usage overhead imposed on the users. In Mirage, we provided some information transparency by using a first-price, open auction in order to help guide user bidding behavior. Despite the potential for strategic manipulation in an open auction, we initially decided to prioritize our goals for usability and efficiency improvement over incentive-compatibility. Perhaps not unexpectedly, users not only learned how to use the system effectively, but a few users

eventually developed strategies to manipulate allocations in their favor. These results provide evidence that some end users may exhibit the sophisticated usage patterns of rational economic agents and serves to justify the use of market based methods in seeking to address these kinds of manipulations. The following are descriptions of the four primary bidding behaviors that we observed during the initial Mirage deployment.

Behavior 1: *underbidding* based on current demand. Since all outstanding bids in our initial deployment were publicly visible, users could observe periods with low demand and submit low bids in these periods. For example, one user would frequently bid a low value, such as 1 or 2 when no other bids were present. Underbidding is not necessarily a problem in this situation, because it can still result in a utility-maximizing allocation; e.g., if supply exceeds demand then all interested users receive an allocation. This can be a problem in for-profit systems, however, where it would be important to use a reserve price to provide good revenue properties. Moreover, it suggests that users need to be strategic in thinking about how and when to bid, which indicates that such a system might be difficult to use.

Behavior 2: *iterative bidding.* The possibility of user underbidding coupled with uncertainty about the bid values of other users results in "iterative" bidding, i.e. a behavior in which a user adjusts her bids while an auction remains open and in response to price feedback. This poses a problem for system performance in Mirage because the auctions need to have a definite closing time (because the associated resources are perishable), and users may still be adjusting their bids when an auction closes. The impact of this strategy is a potential efficiency loss in the allocation.

Behavior 3: *rolling window manipulation.* Unlike auctions for tangible goods, resource allocation in computer systems are not allocated permanently, but rather allocated for particular intervals of time. Since many experiments by Mirage users can span several days, we permitted users to bid for allocation blocks of 1, 2, . . . , or 32 hours in size. In order to allow users to plan in advance, we auctioned off resources over a rolling window of 72 hours into the future. In periods of over-demand (e.g., during the SenSys 2005 conference deadline) the entire window of resources becomes fully allocated, and we found that the design of the rolling window can lead to unintended consequences. For example, consider a scenario with only two users, A and B, where user A requires a 4-hour block of nodes, and user B requires only a 2-hour block. If the window of resources is fully allocated, user A must wait at least 4 hours before a contiguous 4-hour

block is available. However, after only 2 hours, user B will win her allocation, regardless of her valuation relative to user A's. Furthermore, if user B continues this behavior, it is possible that user A will be starved indefinitely. During times of heavy resource load, we observed that no bids involving a block larger than 2-hours could be satisfied. It is possible that there were outstanding bids involving larger blocks and proportionally larger bid amounts that could not be allocated because of the rolling window.

Given the negative impact of these observed user behaviors, we responded by adjusting the Mirage auction protocol. First, we instituted a sealed-bid auction format, thereby discouraging Behaviors 1 and 2. We also responded to Behavior 3 by increasing the allowable time window to be 104 hours, with bid start times constrained to be within the next 72 hours. To understand the rationale for this change, consider the following example. Assume that we have users A and B, where user A requires a 32-hour block of nodes (the maximum allowed), and user B requires a 16-hour block of nodes. If the entire 72-hour window is allocated, the next available 32-hour block occurs 104 hours in the future. By expanding the rolling window to 104 hours and restricting the last 32 hours (of the window) from being reserved, bids from both user A and B will be considered for the next available 32-hour block. Under the original auction format, the bid from user B would win the allocation before the bid from user A could even be considered.

Behavior 4: *auction sandwich attack.* While our changes eliminated Behavior 3 and significantly reduced Behaviors 1 and 2, a fourth behavior exploited the available information about awarded allocations. In this attack, termed the "auction sandwich" attack [78], a user exploits two pieces of information: (i) historical information on previous winning bids to estimate the current workload and (ii) the greedy nature of the auction clearing algorithm. In this particular case, we observed a user employing a strategy of splitting a bid for 97 MICA2 motes across several bids, only one of which has a high value per node hour. During times of high resource demand, most users request a majority of nodes (most often all 97 motes, since a conference deadline requires large-scale experiments). Since Mirage uses a greedy (first-fit) heuristic in its auction-clearing algorithm, the user's single high value bid is likely to win and because the bids from other users are then blocked and unable to fit in the remaining available slots, her other low-valued bids fill the remaining slots. We produce an actual occurrence of this behavior in Table 3.1. Here, user A submits three bids, the main one being a bid with value 130 (value per node hour $130/(4\dot{4}0) = 0.813$) and used to outbid a bid from user

B, with value 1590 (value per node hour $1590/(32\dot{9}7) = 0.0512$). Once the high valued 40-node bid has occupied its portion of the resource window, no other 97-node bids can be matched. Consequently, the user wins the remaining 57 nodes using two bids: a 24-node bid and a 33-node bid, both at low bid prices.

Table 3.1: Strategy S4 on 97 MICA2 motes. [78]

Time	**Project**	**Value**	**#Nodes**	**#Hours**
04-02-2005 03:58:04	user B	1590	97	32
04-02-2005 05:05:45	user A	5	24	4
04-02-2005 05:28:23	user A	130	40	4
04-02-2005 06:12:12	user A	1	33	4

3.6 Conclusions

From our deployments, we see that a market-based allocation system *can* significantly reduce the management burden and simultaneously improve user satisfaction on large-scale federated infrastructures. The allocation decisions in Bellagio and Mirage are autonomously driven by user-provided job utility information, with otherwise, very limited human intervention. Also, when user behavior is properly constrained, the scheduling policies can lead to utility-maximizing resource allocations. Our deployment experience also highlights a few challenges to using market-based systems with real users and applications.

- *Markets can increase allocation efficiency, but only if users understand how to use the market.* There exist many simpler alternatives to markets for resource allocation; e.g., traditional methods that do not require users to bid or provide demand information a priori. We hypothesize that users might sometimes prefer this simplicity over the potential benefits of a market; we saw this behavior in Bellagio. But we also observe that given enough incentive, users are willing to put forth the effort to use a market system, which we saw in Mirage. It is likely that not all users will be able or willing to use market mechanisms correctly. We consider this problem in the next Chapter, where we analyze the sensitivity of our market-based system to this particular type of user difficulties. Specifically, we use workload data from Mirage to compare the sensitivity of different allocation policies — ranging from traditional, simplistic policies, to our market policy —

to various forms of inaccurate information, such as user uncertainty, haphazard bidding, or imperfect currency assignments.

- *The cost of using a market may be prohibitive to some users.* In the course of the Bellagio deployment, we learned that there is a cost to using the auction interface, and that many users prefer both *immediate* and *easier* access (i.e., standard use best-effort, proportional-share) over a potentially larger resource allocation acquired from a more complicated bidding interface. For example, in Mirage, we observe that many users require more immediacy for a significant fraction of their resource requests. We hypothesize that it is possible to provide system support to reduce the amount of planning a user must do to engage the allocation mechanism. In Chapter 5, we address the issue of immediate use by using service contracts over sets of user jobs (however, we frame this problem within the scope of a larger problem setting) with, and in Chapter 6, we address the inherent user costs imposed by our auction mechanism by providing system support to help determine application resource preferences in a time-sharing system.

3.7 Acknowledgements

Chapter 3 is in part a reprint of material that appears in the Proceedings of the OASIS Workshop, 2004, by Alvin AuYoung, Brent N. Chun, Alex C. Snoeren, and Amin Vahdat; and Chapter 23 of *Market Oriented Grid and Utility Computing* published by Wiley, 2009, written by Alvin AuYoung, Phil Buonadonna, Brent N. Chun, Chaki Ng, David C. Parkes, Jeff Shneidman, Alex C. Snoeren and Amin Vahdat. The dissertation author is the primary investigator and author in this paper.

Chapter 4

Information Accuracy

In the previous chapter, we discuss the decision to trade-off usability for incentive-compatibility in both Mirage and Bellagio. As we alluded to in Chapter 2, the inability to achieve an incentive-compatible auction mechanism does not necessarily preclude us from achieving efficiency. However, given the additional complexity of an auction over simpler allocation systems, and the possible side-effects of using a virtual currency, this begs the question: how sensitive is our market-based approach to such imperfect conditions compared to existing best-effort policies?

In order to investigate the robustness of our market-based allocation policy, we conduct an extensive simulation study that compares the performance of market-based policy relative to existing alternatives under a range of these imperfect conditions. We develop a model for varying the range of fidelity in market information: from complete accuracy to complete noise. Using workloads based upon two different systems, we try and generalize the expected performance of a market-based approach in other domains.

To our knowledge, ours is the first study of a market-based scheduling system to analyze the potential impact of the imperfect information likely to be submitted to the scheduler in a live deployment, and also one of the few to analyze its performance in the context of real user utility information. Quantifying the effect of inaccurate information is vital to understand how robust the underlying market mechanisms [49, 79] must be in order to effectively support a market-based scheduling approach, or, conversely, to understand when the inability for users to provide sufficiently accurate information might prevent the effective deployment of a market-based mechanism.

4.1 Motivation

From our deployment experience with Mirage and Bellagio, we see evidence that market-based systems have the potential to work well, but that many concerns about the fragility of and burden imposed by these systems are founded. In fact, most of the studies touting the potential benefits of these systems assume perfect operating conditions, and as we see from our deployments, such assumptions do not always hold in practice, and it is unclear how market-based systems will perform relative to these idealized results. Not only might they not improve performance, but the potential exists to significantly harm aggregate utility and fairness in existing systems should users provide *inaccurate information*. While the simpler, more traditional scheduling policies do not provide as much *value* to users in the best case, it would appear that they offer less fragility upon deployment.

In fact, these apprehensions may contribute to the resistance to using market-based policies in supercomputers and high-performance computing centers. Although such clusters continue to increase resource capacity — with systems already achieving petascale performance — resource allocation for jobs remains a significant challenge [93]. For example, some jobs may require high-speed interconnects for its many coupled tasks, other jobs may require sufficient local storage for its data-intensive tasks, and some jobs require both; a scheduling policy that does account for these needs will continue to suffer from artificial resource bottlenecks. Despite the continued inability of Grid and supercomputing systems to prioritize jobs based upon user needs [66], these systems continue to deal with over-subscription through either a *first-come-first-served + backfill* (FCFS+backfill) allocation policy, or some form of *fixed-price priority-based* scheduling.

It is worth noting that even many of these simpler, non-market-based scheduling approaches are known to be sensitive to user input [107], and require significant engineering and parameter-tweaking [56] when deployed. Therefore, any apprehension to the potential fragility of a market-based policy may already be deeply rooted.

The purpose of this chapter is to use our deployment experience with Mirage and Bellagio to help compare a class of market-based scheduling policies with traditional scheduling policies. We use an extensive trace-based simulation to evaluate the sensitivity of a market-based scheduling approach when subjected to imperfect conditions likely to exist in a real deployment, as well as practical considerations, such as a heuristic clearing algorithm and currency system. Partially to address concerns from the area

of high-performance computing, we base our simulation study upon fifteen months of user data from Mirage, as well as similar data from the San Diego Supercomputing Center SDSC-SP2 cluster [1]. Since we lack sufficient data from Bellagio, we do not consider time-shared jobs in this chapter. We begin by addressing the fundamental concern that the additional information gathered by market-based schedulers, namely users' expressed job utility, to be inaccurate with respect to their actual (internal) value for jobs. Our evaluation considers two distinct sources of this inaccuracy, which stem from imperfections in user knowledge, and a market-based scheduler's ability to extract this knowledge (due to possible limitations of the currency system used) respectively. We term these sources of inaccuracy *user uncertainty* and *wealth inequity*.

Uncertainty exists, for example, when a user is unsure how to assign a value to a job (e.g., due to insufficient information about future demand or job importance [63, 85, 101]), or is otherwise unable to accurately determine job-specific characteristics prior to submission, such as estimated running time [43, 66, 106, 107]. Markets-based schedulers use a pricing mechanism to extract truthful user valuations. Wealth inequity among users may decrease the effectiveness of such an approach; for example, wealthy users may consistently overstate the value of their jobs (i.e., over-pay for resources), and therefore consistently receive a larger share of resources than relatively less wealthy users, regardless of need. If a virtual currency is used in lieu of real money, such as service units in Supercomputing clusters like SDSC, *all* users may have less incentive to save or spend currency according to true, private values, thereby decreasing the fidelity of resource value information available to a scheduler.

4.1.1 Questions to Address

In this chapter, we focus on the following specific questions:

- How fragile is the performance of a market-based allocation system when subject to imperfect information, specifically, from a range of user uncertainty and effects of wealth inequity?
- How much imperfect information can a batch-scheduled market-based approach tolerate and still add value over existing scheduling policies, such as FCFS+backfill or a priority-based approach?
- Can we estimate the expected information fidelity in, production environments,

and therefore, project the expected benefit of a market-based system over existing approaches?

4.1.2 Summary of Results

For both sources of inaccuracy, we develop a parameterized model that allows us to vary the level of inaccuracy from zero—meaning all user-provided information is perfectly accurate—to one, where user-provided utility is entirely uncorrelated with true values. We use this model to perform trace-based simulations on two distinct workloads. Based on simulations driven by our two workloads, we present the following main results:

- When faced with extreme levels of uncertainty or wealth imbalance, a market-based scheduler can *under-perform* traditional approaches, delivering roughly *half* the aggregate utility of a traditional FCFS+backfilling scheduler in the worst case.

- However, market-based scheduling is sufficiently in-sensitive to levels of inaccurate information that occur in existing computing environments. Specifically, if we expect user utility information to be at least as accurate as observed inaccuracies in run-time estimates, we can expect an increase in aggregate utility of at least 20-100%.

- Finally, we argue that in general, the effectiveness of any market-based allocation mechanism is dictated by its ability to accurately extract user valuation. Specifically, we demonstrate that a fixed-price priority queue allocation mechanism can degrade to the performance of a simpler, FCFS+backfill approach under a variety of conditions. Therefore, we argue that the design of future pricing mechanisms (e.g., fixed prices, auctions) or monetary policies (e.g., virtual currency distribution, spending limits) must explicitly consider the expected fidelity of elicited job value information from users.

Our results indicate that despite the potential for poor performance in extreme cases, market-based scheduling is robust to reasonable levels of inaccuracy and seems likely to outperform traditional scheduling techniques in practice.

4.2 Related Work

The primary focus of this study is to quantify the impact of errors in user-provided information on both a market-based scheduling policy and traditional scheduling policies. We are unaware of any such studies of market-based scheduling systems, particularly with respect to a comparison against traditional scheduling approaches. However, there are many studies that focus on the impact of user errors in traditional scheduling systems.

The effect of errors in user run-time estimates on parallel scheduling algorithms in batch computing environments is the subject of much study [22, 43, 76, 107]. Bailey Lee *et al.* find that even with an incentive (a raffle) for users to improve their estimates of job run-times, most users are *inherently* unable to do so [66]. Tsafrir *et al.* develop a model for characterizing the error in supercomputing users' job run-time estimates [106], and others have characterized the impact of this error in batch environments through simulation [22, 43, 76, 107], or in simpler environments using a theoretical approach [115].

The primary difference between these studies and ours is that we are considering the effects of imperfect *utility* information, which poses a significant set of additional challenges. First, even with usage data from Mirage, it is impossible to quantify error in utility information; job run-time estimates provide a measurable quantity that can be modeled and compared. To address this challenge, we must develop a model for a *range* of user error. The second challenge is in considering the different sources of error. In a market-based system, there are many moving parts (i.e., virtual currency, users' private information, incentive compatibility of the market mechanism), and the information may exhibit errors from a variety of sources. Our model must capture how any of these sources may manifest in practice.

4.3 Models

We wish to characterize the sensitivity of a market-based batch scheduling system system using realistic user and job data. At a high level, we use real-world data points from Mirage and SDSC-SP2 to establish a ground truth for user job characteristics and user valuation information (defined in this section). Our Mirage data provides one of the few real-world data points for user valuation information, and the SDSC-SP2 data

set provides examples of job characteristics and limited utility information (also defined in this section) in a similar-sized parallel computing cluster. We use these data to drive our simulations.

In this section, we describe how we model the entities in our simulations: the scheduling environment, job utility functions, and the information inaccuracy.

4.3.1 Job Scheduling Algorithm

Developing an effective and deployable market-based scheduler for parallel batch environments is challenging and the subject of much prior work [8, 12, 25, 54, 67, 91]. In these algorithms, users submit a *utility function* along with each job, which conveys the utility, or value, the user will receive from the job, as a function of the job schedule (e.g., turnaround time for the job). The goal of the scheduler is to determine a schedule that maximizes the aggregate utility delivered to all users. In these systems, utility typically represents not only the value the user will extract from the job, but also the price she must pay to the system upon completion. Hence, rational users have an incentive to truthfully reveal their utilities.

Thus, one way to view these prior algorithms is that they implicitly run a repeated auction, as is done in Mirage. In our simulations, we use the same *FirstPrice* scheduling heuristic used in Mirage. In this context, the basic idea of the heuristic is to calculate the value density for each job, which is the initial value of the utility function (i.e., value at time 0 in Figure 4.1), divided by its size and length. The heuristic performs a greedy "first-fit" for each job, ordered by value density. We defer considering extensions to the FirstPrice heuristic, such as a variable cost model like that of Popovici *et al.* [91], pre-emptible jobs to the next chapter.

4.3.2 Utility Functions

Prior work characterizes each job utility function by its initial value and its decay over time (see Figure 4.1). It is possible that the decay can cause the utility function to become *negative*, but in our model and in previous models, we are implicitly assuming *individual rationality* (as defined in Chapter 2), such that users will not incur negative utility when using the system. One way to think about this assumption is that users will not submit a job to the scheduling system if it would impose a net cost on them. In all of our simulations, none of the schedulers will execute jobs that deliver no positive

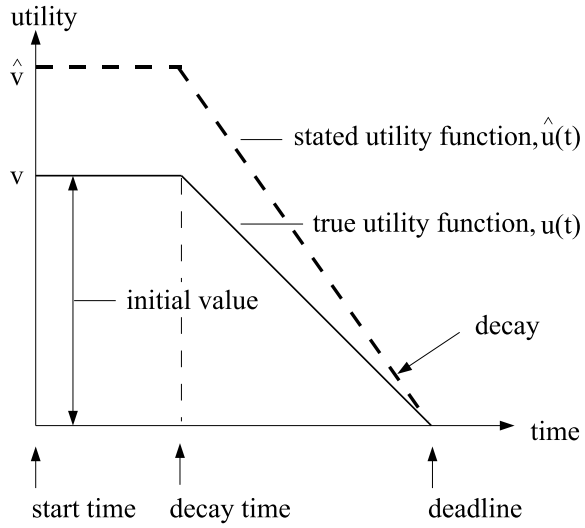


Figure 4.1: Job utility function (linear decay) as a function of v,$\hat{v}$.

utility to the user. We extend the model of utility function to handle the possibility of negative values in Chapter 5.

In this chapter, we focus on varying the initial value, and decay of a utility function. A study by Bailey Lee *et al.* [66] suggests that these two components are sufficient to characterize the time-varying utility of jobs from real users. Lacking deployments from which to observe the values and decays of utility functions belonging to real system users, previous studies have relied on generating these components based on artificial values to drive their simulations [8, 25, 54, 91]. In contrast, we can use workload data from Mirage to create a distribution for initial values and job deadlines (both of which are contained in the job logs), and explore the effect of different types of utility decay (not contained in the job logs) in between the supplied initial value and job deadline information. In particular, for utility function decay, we examine the impact of the decay types observed by Bailey Lee *et al.* in their study of job utility functions among real users at the San Diego Supercomputer Center (SDSC) [66].

4.3.3 Information Inaccuracy

We capture the potential fragility of a utility-based scheduling approach by considering the impact of inaccuracies in the user-provided utility information. Specifically, we model two types of inaccuracy commonly associated with utility information: valuation uncertainty, and wealth inequity. These types of inaccuracy are challenging to

model because there is very little data available that describes how these inaccuracies may manifest in practice. Therefore, we create a parameterized model that can capture a range of possibilities associated with each type of inaccuracy. Namely, for a given utility function $u(t)$, we create a perturbed utility function $\hat{u}(t)$ for each of these error types, and parameterize the *magnitude* of perturbation by the variable k. For the purposes of this study, we ignore possible inaccuracies in job deadlines as, in many cases, job deadlines are externally imposed upon the user. We defer investigation of misstated deadlines to future work.

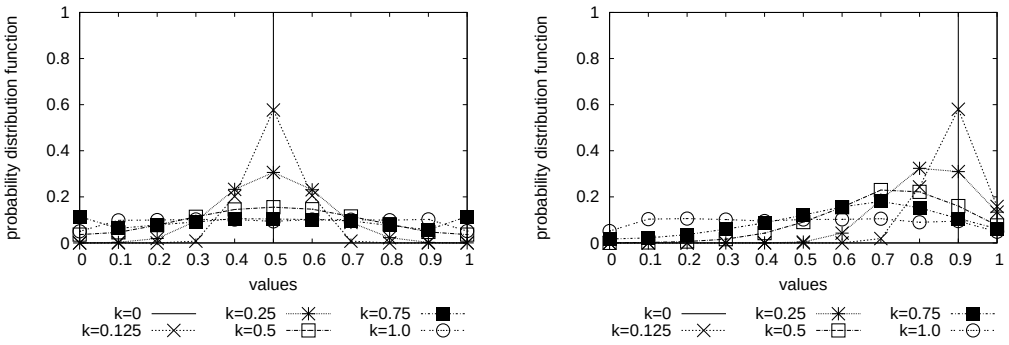
Uncertainty

A first source of inaccuracy has to do with uncertainty. In many cases, the output (or utility) of a job depends upon the completion of other jobs [7, 8] and therefore a user is unable to determine its value prior to submission. More generally, a user simply may not be able to express her utility for a job due to cognitive or computational complexity [63, 85]. Therefore, we expect many honest and well-intentioned users to express job valuations that are weakly correlated or uncorrelated with their true valuations.

We assume that each job's true utility function has a start value v that is drawn from a fixed distribution V. In our simulations, V is the distribution of per-node-hour valuations from the workload trace. To model uncertainty we generate a perturbed value, $\hat{v}$, which represents the start value of a user's *stated* utility function (Figure 4.1). We define the parameter $k \in [0, 1]$ as the level of uncertainty, with larger values of k representing less certainty. For example, $k = 0$ indicates absolute certainty (i.e., *no* uncertainty), and $k = 1$ implies absolute uncertainty. If $k = 0$, we have the property that $\hat{v} = v$, and for $k = 1$, $\hat{v}$ will be uncorrelated with v.

The key challenge in defining a model for uncertainty is remaining independent of the distribution V; this is important in case V exhibits significant skew, as is the case for the value distribution in Mirage. Our procedure for doing this is as follows. Define the function $CDF : V \rightarrow [0, 1]$ as map from a value in v to its percentile within the distribution V, and its inverse function, $CDF^{-1} : [0, 1] \rightarrow V$. Therefore, if we draw $v \in V$, and v is in the 75th percentile of all values in V, then $CDF(v) = 0.75$, and $CDF^{-1}(0.75) = v$.

For a given value v, we find its percentile in V (as a fraction) $f = CDF(v)$. From f, we draw $\hat{f} \in Gaussian(\mu = f, \sigma = k/2)$. We determine our perturbed value $\hat{v}$



(a) Uncertainty ($CDF(v) = 0.5$). (b) Uncertainty ($CDF(v) = 0.9$).

Figure 4.2: Generating $\hat{v}$ from v for uncertain users.

from the inverse CDF as $\hat{v} = CDF^{-1}(\hat{f})$. By using a Gaussian distribution with mean f, and standard deviation $k/2$, we have an intuitive way to understand how the value $\hat{v}$ deviates from v. For example, if $CDF(v) = 0.5$, this means that there is a "good chance" (from the definition of a Gaussian distribution, this means 68%) that $\hat{v}$ will be within k standard deviations of the percentile of v.

Figure 4.2 illustrates a sampled distribution when choosing $CDF(\hat{v})$ from $CDF(v)$. We see that when $k = 0$, $CDF(\hat{v})$ is exactly $CDF(v)$ with probability 1, and as k approaches 1, $CDF(\hat{v})$ is drawn from an increasingly noisy distribution centered at $CDF(v)$. Note that since a Gaussian distribution is unbounded, we clip $\hat{f}$ such that $\hat{f} \in [0, 1]$. As we can see from Figure 4.2 ($CDF(v) = 0.5$), the end points are raised for values 0 and 1 due to the effects of clipping. However, since clipping is significant only as k approaches 1, we do not expect it to adversely affect our results.

Figure 4.2(b) plots the distribution from which $\hat{f}$ is drawn for a large value of $CDF(v)$.

Wealth Inequity

The second source of inaccuracy has to do with a user's wealth. All market-based scheduling systems assume the existence of a currency — either real or virtual — with which users will express their value, or willingness to pay for a job. In either scenario, we assume that a user with a larger budget of currency will receive a larger share of resources, regardless of the true value derived from that job. For example, a user with a disproportionately large share of wealth has more "disposable income", and is therefore,

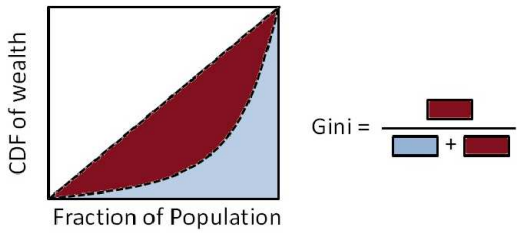


Figure 4.3: Graphical depiction of Gini coefficient.

Table 4.1: User wealth inequity: ϵ is approximately zero; the last table row corresponds to a population larger than 5 to illustrate how wealth inequity can approach 1.

$\{w_1, w_2, w_3, w_4, w_5\}$	Gini coefficient (k)
$\{1, 1, 1, 1, 1\}$	0
$\{1, 1, 1, 1, \epsilon\}$	0.2
$\{1, 1, 1, \epsilon, \epsilon\}$	0.4
$\{1, 1, \epsilon, \epsilon, \epsilon\}$	0.6
$\{1, \epsilon, \epsilon, \epsilon, \epsilon\}$	0.8
$\{1, \epsilon, \epsilon, ...\epsilon, \epsilon, \ ...\}$	1

more likely to forgo an additional dollar than a user with comparatively less wealth (this a consequence of the Diminishing Marginal Utility of Wealth). For systems that use a virtual currency (e.g., supercomputing service units, or currency in Mirage), this effect is exaggerated since *all* users will be motivated to spend their remaining balance on their remaining workloads, since the currency can only be redeemed for this purpose.

To model this behavior, we define a wealth parameter, $w_i > 0$, for each user i, with the property that $w_i > w_j$ indicates that user i has more wealth, or ability to pay, than user j. For any user i, we generate $\hat{v}$ by scaling v by w_i (i.e., $\hat{v} = w_i \cdot v$).

Similar to the case with uncertain users, we use parameter k to indicate the level of wealth inequity across users in the system. For $k = 0$, we have $w_i = w_j$ for all users i, j, and for $k = 1$, nearly all wealth is held by a single user. There is a formal economic metric used to describe such wealth (im)balance in a society: the *Gini* coefficient [14]. As we illustrate graphically in Figure 4.3 (the x-axis is in ascending order of wealth), this coefficient is defined as a fraction between 0 and 1 where 0 represents perfect wealth *equality* and 1 indicates perfect *inequality*. Therefore, we simply define k as the Gini coefficient of the wealth coefficients in the population. Table 4.1 contains the Gini coefficient for various values of w_i in a given population.

Finally, given $\hat{v}$ as a result of user uncertainty or wealth inequity, we generate a user's stated utility function, $\hat{u}(t)$, by scaling the non-decaying portion of a user's utility function by $\hat{v}/v$, and decay the job from $\hat{v}$ to the original deadline in the same manner as the original decay. Figure 4.1 illustrates such an example for a linear-decay utility function.

4.4 Experimental Setting

In this section, we discuss our simulation setup. Based upon data from Mirage and SDSC-SP2 workloads, we create statistical distributions to drive our simulation environment. Based upon these distributions, we generate multiple instantiations of a workload that we run both our market-based scheduler and alternative scheduling policies against, and compare the outcomes against varying levels of information inaccuracy.

4.4.1 Simulator

We have constructed a concurrent simulation environment to mimic a generic parallel computing cluster and use it to compare the performance of the FirstPrice market-based scheduling policy against that of two common scheduling policies: first-come, first-served (FCFS) ordering with EASY backfilling [70], and a priority queue that uses four levels of priority (SDSC supports four levels of priority). While neither the FCFS+backfill nor the priority scheduler explicitly considers job deadlines, we assume that jobs that pass their deadlines are canceled or otherwise removed from the queue such that neither algorithm will consider scheduling a job that cannot complete before its deadline passes, and therefore, only jobs with positive value are scheduled.

4.4.2 Workloads

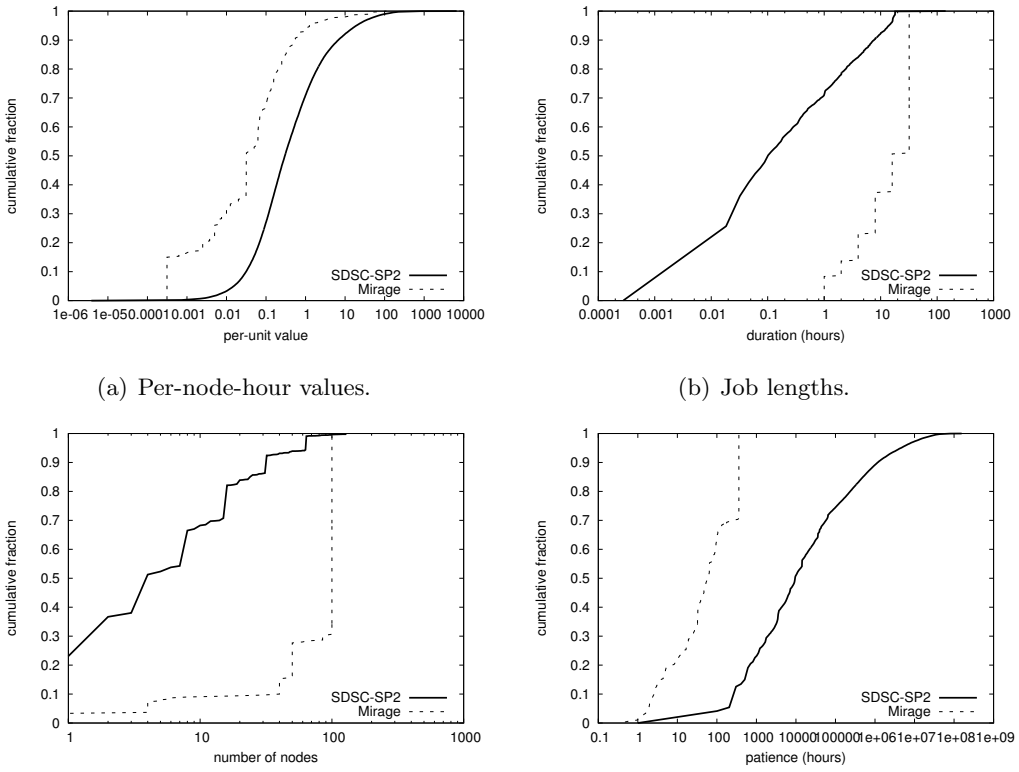
As opposed to the graphs we present in the previous chapter that use raw data, the graphs in this chapter use *distributions* of data from our chosen traces. The Mirage trace contains 15-months (2006-2007) worth of data, and the SDSC-SP2 trace contains 24 months of job submissions (1998-2000). In Figure 4.4, we plot distribution data for job size, length, value and deadlines for both the Mirage and SDSC-SP2 workloads. Using distributions instead of raw data to drive our simulations allows us to extract important features from the workload, and repeated different experiments over the same

dataset. The features we focus on in for each workload are the job characteristics, and job arrivals.

Distributions of these job characteristics are presented in Figure 4.4. The Mirage data-set has various discontinuities in the CDFs due to restrictions on bids placed upon users. For example, a minimum bid value of 1 unit is required for all bids, which leads to the minimum per-node-hour bid seen in the CDF of Figure 4.4(a). The SDSC-SP2 workload, on the other hand, creates random bid values based on different Gaussian distributions; these values are generated by the aforementioned tool by Bailey Lee *et al.* [66] based upon the priority queue it was submitted to and observed queuing time. More details about this tool can be found in the authors' paper. Similarly, node reservations are partitioned into into hour-long blocks (in increments of powers of 2) between one and 32 hours (Figure 4.4(b)), and users are able to select no more than 100 of the available 150 nodes in a given Mirage bid (Figure 4.4(c)), whereas in SDSC-SP2, users are only limited by a maximum run-time of approximately 10 hours, and 128 nodes. Finally, users are able to submit job requests up to 2 weeks in advance, as illustrated by the maximum value seen in the CDF of Figure 4.4(d).

One interesting thing to note from these distributions is that workloads from previous simulation studies of similar market-based scheduling algorithms often use a bi-modal distribution of valuable and less-valuable jobs, and a bi-modal distribution of urgent and less-urgent jobs [8, 25, 54, 91]. In comparison, the workload data we see here exhibits an even wider distribution of job valuations and deadlines than those used in previous studies.

The other dimension of our workload — job arrival patterns — is more challenging to model. Job arrival patterns are highly variable in both Mirage and SDSC-SP2, making it difficult to draw conclusions from the trace as a whole. Instead, we partition the trace data into two operating regimes based upon short-term demand: *light* and *loaded*. We define the light regime to be any time period where almost all incoming requests can be completely satisfied by available system capacity; in other words, total resource demand can satisfied on average. (This regime includes many instances where the system is completely idle.) In the loaded regime, incoming resource demand exceeds system capacity. For each operating regime, we extract all job requests and calculate the distribution of inter-arrival times, bid sizes, lengths, patience and values and simulate the results separately. Table 4.2 contains a few workload values for each demand regime.



(a) Per-node-hour values.

(b) Job lengths.

(c) Job sizes.

(d) Job deadlines.

Figure 4.4: Distributions of the workloads used to drive our simulations.

Note that bid values and arrival rates are larger in the loaded regime than the light regime. However, jobs appear to be less urgent during the loaded regime. The increased patience seems to indicate additional planning by users in order to use the system.

To account for the possibility of hidden demand or future peak usage, we create a synthetic operating regime that we call *extreme*; it maintains the same job distribution as the loaded regime, but with double the number of requests (i.e., inter-arrival times are halved).

Finally, although the Mirage trace reports each user's job valuation and deadline, it contains no information about how the utility of the job decays over time. We consider the following scenarios to capture the possibilities of job decay (inspired by study of SDSC users from Bailey *et al.* [66]):

- *convex*: Job utility decays as a convex curve from arrival time until the deadline.
- *linear*: Job utility decays linearly from arrival time+run time until the job deadline

Table 4.2: Workload characteristics for different demand regimes; deadline represents a relative offset from job submission time.

	Light			Loaded		
Mirage	**Mean**	**Min**	**Max**	**Mean**	**Min**	**Max**
Value (per node-hour)	0.16	0.0025	2.5	1.5	3e-4	111
InterArrival (hr)	18	0	44	8.6	0	24
Deadline (hr)	81	0	359	133	0	359

	Light			Loaded		
SDSC	**Mean**	**Min**	**Max**	**Mean**	**Min**	**Max**
Value (per node-hour)	0.002	4.7e-5	1.84	0.001	1e-9	1.84
InterArrival (hr)	0.317	0	22	0.312	0	22
Deadline (hr)	231.8	0	40209	274	0	43238

(Figure 4.1). This model is used frequently in previous work [8, 25, 54, 91].

- *flat*: Job utility does not decay until the deadline, at which point it drops instantaneously to zero.
- *mix*: Job utility decays as either *flat*, *linear*, or *convex*. Each type is equally probable.

4.4.3 Schedulers

In addition to the Mirage FirstPrice scheduler, we consider two alternatives. As a baseline, we simulate FCFS+backfill, a traditional, non-utility based scheduler. In addition, we study a simple four-level priority scheduler modeled on the SDSC scheduler, which can be viewed as a simplistic form of market-based scheduling (that captures starting value, but ignores decay). At SDSC, users are charged for their jobs in accordance with the priority level they assign [96], so there is motivation to accurately classify job priority.

In our baseline experiments, we consider two configurations of the priority scheduler: static (*PRIO+static*) and dynamic (*PRIO+demand*). The former uses the statically defined priorities observed in SDSC trace logs. Specifically, rates are fixed at $0.5x$, $1x$, $2x$, and $1.8x$ service units per CPU hour, respectively, for each of the four increasing priority levels. (Note the highest priority is reserved for "express" jobs which are restricted in their duration, hence the lower cost.) If we define priority level $1x$ to be the median job value in our workload, then jobs with values twice the value of the me-

dian have priority level $2x$, and sufficiently small jobs with at least $1.8x$ the value of the median qualify for the express queue, or highest priority.

In both workloads however, most of the important jobs are several orders of magnitude more valuable than less important jobs, so the static priority settings do not appropriately segregate jobs of different values. Therefore, we also consider a dynamic approach (PRIO+demand), which classifies job priority based upon observed demand. Specifically, we determine the appropriate priority level for each job based on its value and the total distribution of job values in the workload. We use an implementation of the Expectation Maximization (EM) algorithm [33] to decompose the distribution of observed values into four separate Gaussian distributions, each with its own weight, mean and standard deviation. Each of the component distributions represents a single priority level (e.g., the distribution with the highest mean represents jobs with the highest priority level), and based on this decomposition, the *PRIO+demand* algorithm determines to which priority level a job corresponds.

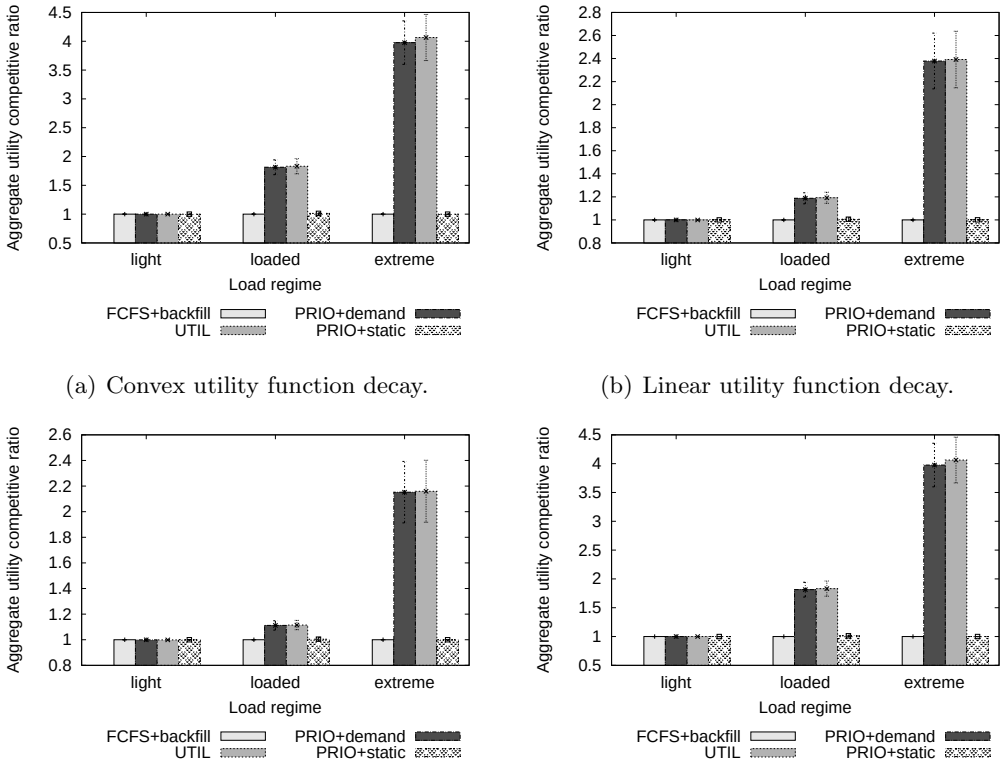
4.5 Results

We quantify the performance of a market-based scheduling approach *relative* to traditional, non-market-based approaches under various operating conditions — particularly in the face of inaccurate utility information. We are primarily interested in two metrics: the *aggregate utility* delivered to users, and the *distribution of utility* across its users. Our primary goal is to identify how much uncertainty or wealth inequity a market-based approach can tolerate before degenerating. Secondarily, we seek to determine whether such levels of uncertainty arise in real workloads.

4.5.1 Baseline Performance

In order to calibrate our study with previous studies, we quantify the baseline (i.e., assuming perfect information) performance improvement of market-based scheduling over a non-market-based approach under each workload. Specifically, this experiment measures the *competitive ratio* of each approach (measured as a ratio of aggregate utility delivered by a scheduler to the FCFS+backfill scheduler).

The results for each workload are plotted in Figures 4.5 and 4.6 (unless otherwise noted, standard deviation bars are depicted along with the averages). As expected, the



(a) Convex utility function decay.

(b) Linear utility function decay.

(c) Flat utility function decay.

(d) Mixed utility function decay.

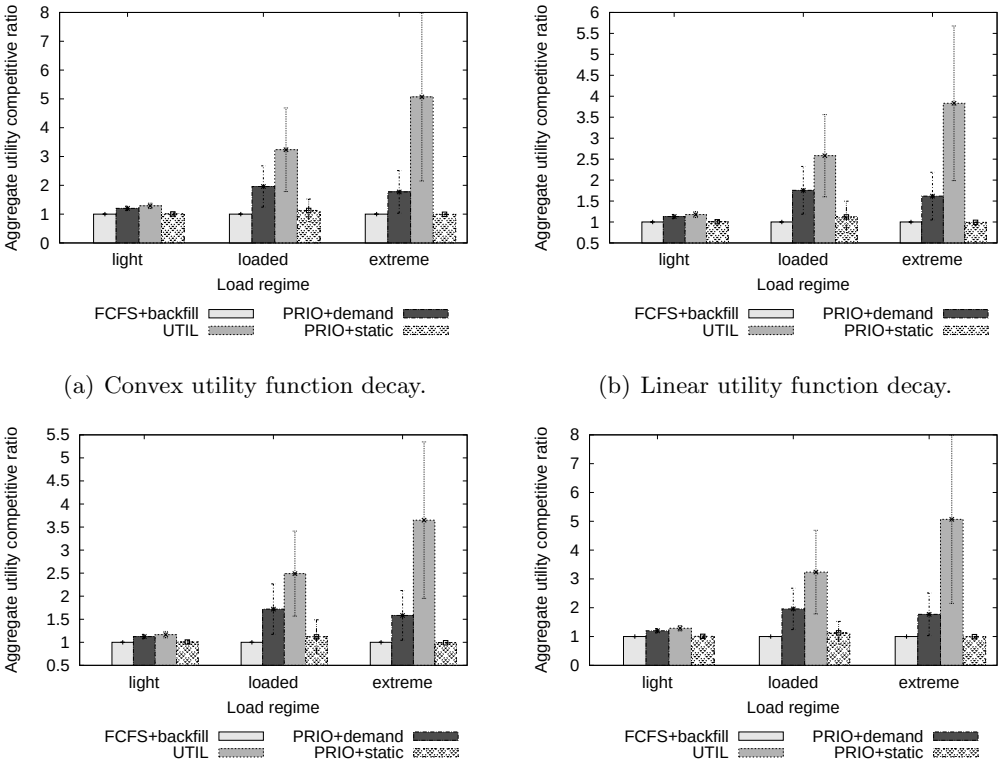
Figure 4.5: Mirage: Baseline competitive ratio.

performance for all the schedulers is the same under light conditions, regardless of utility function decay or workload.

Under *loaded* demand, the competitive ratio for the FirstPrice scheduler (*UTIL*) and *PRIO+demand* are similar, and range from 1.1 to 1.8 in the Mirage workload. In the SDSC workload, we see that the competitive ratio for *UTIL* ranges from 2.5 to 3, and *PRIO+demand* ranges from 1.75 to 2. For the *extreme* demand regime, the competitive ratios range from 2.2 to 4 in Mirage, and from 3.5 to 5 in SDSC-SP2, depending on utility decay.

In the Mirage workload, the median job size is 100 (out of 100 possible nodes), and in the SDSC-SP2 workload, the median job size is 5 (out of 128 possible nodes). Therefore, we can account for the difference in competitive ratios across workloads by recognizing that there are more scheduling decisions to make in the SDSC-SP2 workload, which provides an opportunity for the *UTIL* and *PRIO* schedulers to make a "smart" decision, and at the same time, for the *FCFS* scheduler to make an unwise decision.

Interestingly, the *PRIO+static* algorithm performs poorly compared to the other



(a) Convex utility function decay.

(b) Linear utility function decay.

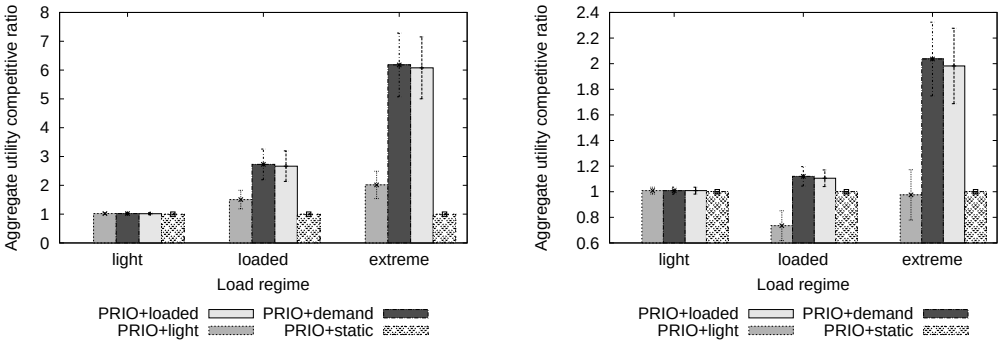
(c) Flat utility function decay.

(d) Mixed utility function decay.

Figure 4.6: SDSC-SP2: Baseline competitive ratio.

market-based algorithms. In both workloads, it is unable to effectively differentiate the highest-valued jobs, and therefore performs about as well as the FCFS+backfill algorithm. In Figure 4.7, we plot the baseline performance of the *PRIO+static* algorithm against algorithms which take into account the distribution of value densities in job workload. For example *PRIO+light* establishes 4 levels of priority based on value-density observed during light demand, and *PRIO+demand* established priority levels based on value-density observed during the *entire* trace. We see that a priority-based algorithm *must* take into account the value density of the incoming job stream in order to approach the performance of the market-based approach.

The preceding graphs illustrate the overall allocation efficiency in the system, but do not consider how utility is distributed among users. We define *fairness* to be a measure of the utility share received by each user. Formally, we measure the fraction of utility received by a user divided by the maximum possible utility he *could have* received from all of his submitted jobs. In Figure 4.8, we see the average, minimum and maximum utility share received by each user (the error bars represent minimum and maximum).



(a) Mixed utility function decay.

(b) Flat utility function decay.

Figure 4.7: Mirage: Competitive ratio using different priority-based schedulers.

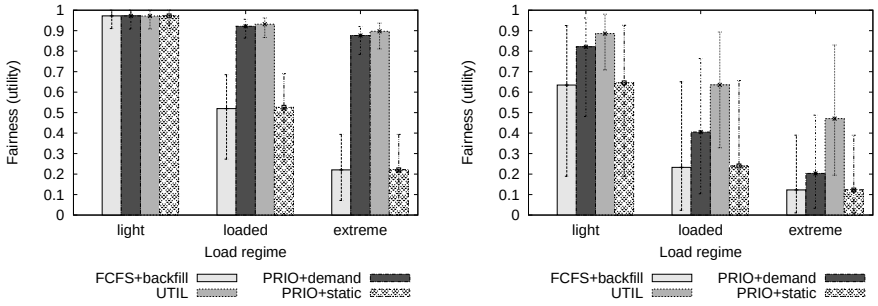
For light demand, all scheduling approaches satisfy over $60-90\%$ of the utility in each workload, but, as demand increases, the utility share of the FCFS and PRIO+static schedulers decrease rapidly. Interestingly, while all algorithms schedule roughly the same fraction of jobs for each user (not shown for space considerations), both market-based approaches (UTIL and PRIO+demand) schedule jobs with $2-3x$ (for SDSC-SP2) to $1.8-4.5x$ (for Mirage) more value to the average user, and $1.5-7x$ to the minimum (worst-case) user. In general, the relative improvement of utility and priority scheduling decreases (and fairness increases) as the utility function decay moves from convex to linear to flat. In the interest of space, we plot only the *PRIO+demand* scheduler for the remainder of the simulations.

4.5.2 Information Inaccuracy

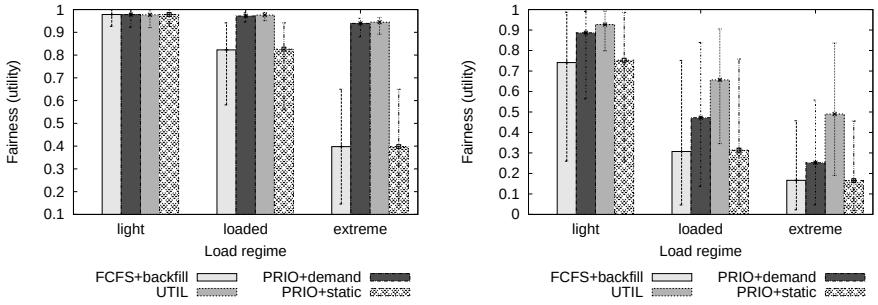
In our next set of experiments, we measure the impact of imperfect utility information on the competitive ratio. We consider the impact of each type of information inaccuracy separately.

User Uncertainty

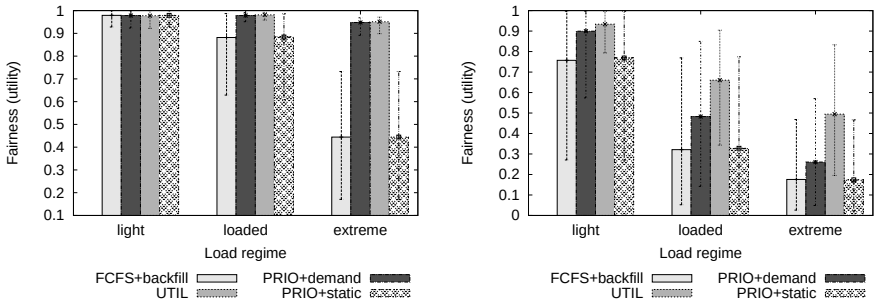
In Figure 4.9, we plot the competitive ratio under user uncertainty as a function of k (degree of user uncertainty). We see that again, for a lightly loaded system, there is effectively no difference among the schedulers. However, as demand increases, the impact of uncertainty on the competitive ratio depends directly on the utility function decay. In fact, without sufficient demand, a linear or flat decay and $k = 0.75$ uncertainty



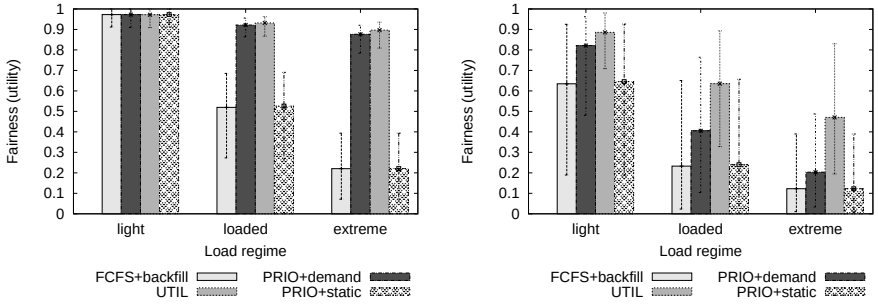
(a) Convex utility function decay.



(b) Linear utility function decay.



(c) Flat utility function decay.



(d) Mixed utility function decay.

Figure 4.8: Baseline utility fairness in Mirage (left), SDSC-SP2 (right)

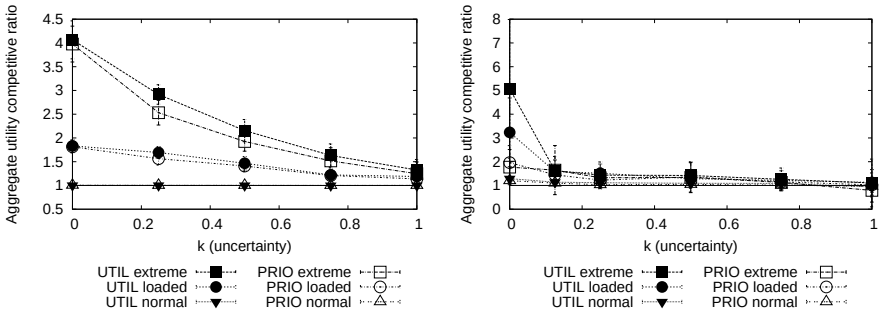
renders both priority or market-based scheduling approaches 20% *less* effective than a FCFS+backfill approach in the Mirage workload. In the SDSC workload, uncertainty of $k = 0.125$ can drop the competitive ratio of the market-based approaches by a factor of $2 - 3$, depending on utility decay. Note that while the SDSC workload provides more opportunity to deliver utility to its users (i.e., higher competitive ratio than in Mirage workload for baseline experiments), it is also more vulnerable to error from user uncertainty.

All schedulers are effectively fair under light load (light loads not shown for the remainder of this chapter for space considerations). For the loaded regime, however, we see that when k exceeds 0.5, that the market-based approaches can be, on average (and max-min), *less fair* than the traditional approach: Figure 4.10 (Mirage on the left, SDSC-SP2 on the right). For the extreme regime, this breaking point occurs even earlier, at $k > 0.25$ (Figure 4.11). Again, as with the competitive ratio for aggregate utility, we see that generally, as user certainty decreases, the user utility-share also decreases, and that the uncertainty has a much more pronounced effect in the SDSC-SP2 workload since there are more scheduling opportunities for error.

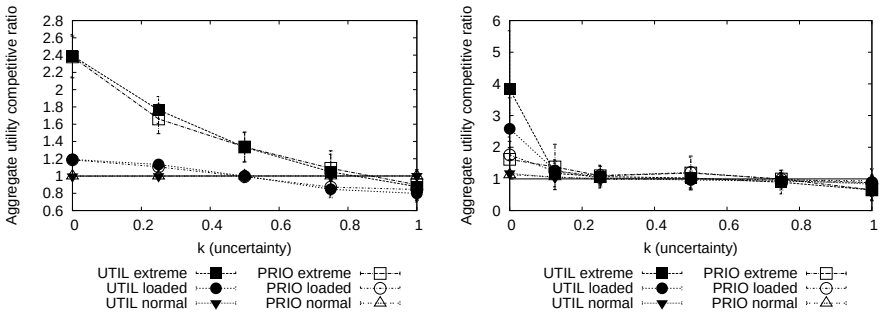
Wealth Inequity

This section considers the impact of wealth inequity between users on a market-based approach. In these experiments, the degree to which users have different levels of wealth is parameterized by the variable k. Figure 4.12 shows the aggregate utility competitive ratio for each market-based scheduler as a function of k.

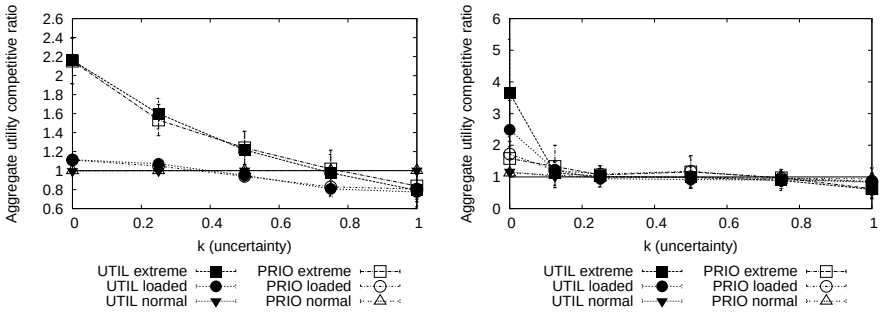
Similar to the scenario with uncertain users, there is effectively no difference in *aggregate utility* among the scheduling approach for a lightly loaded system, but as demand increases, the competitive ratio for both market-based schedulers increase, depending upon the steepness of utility function decay. For the Mirage workload, the competitive ratio for *UTIL* incurs its minimum at $k = 0.3$ (loaded demand), while *PRIO* incurs its minimum as k approaches 1. To understand why the worst-case performance for *UTIL* occurs at a different value of k from that of *PRIO*, consider the difference between a scenario with $k = 0.5$ and $k = 0.9$. With $k = 0.5$, half of the users hold most of the wealth in the system, and they are able to consume most of the available resources. However, as k increases to 0.9, more of the wealth is held by a single user, in which case he does not consume most of the resources. The remaining resources are



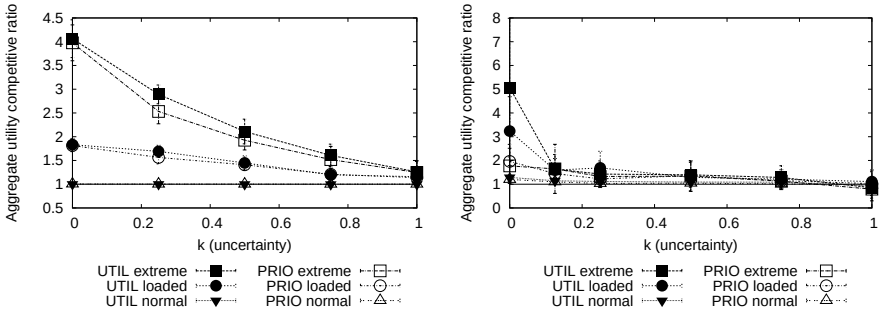
(a) Convex utility function decay.



(b) Linear utility function decay.

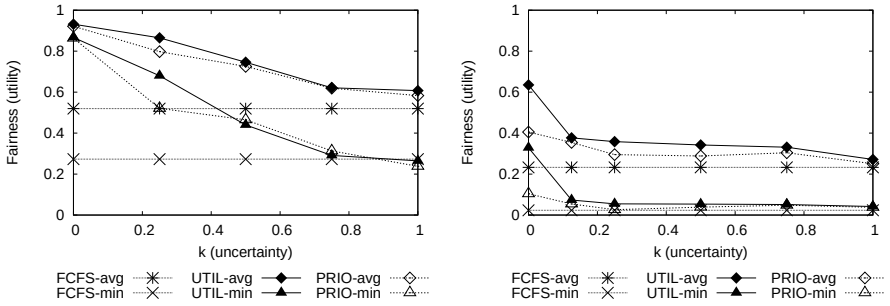


(c) Flat utility function decay.

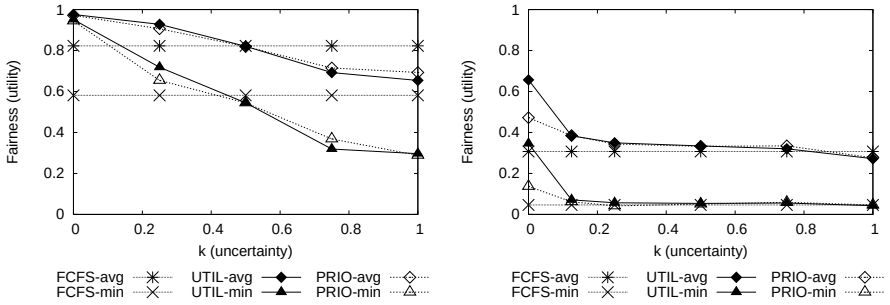


(d) Mix utility function decay.

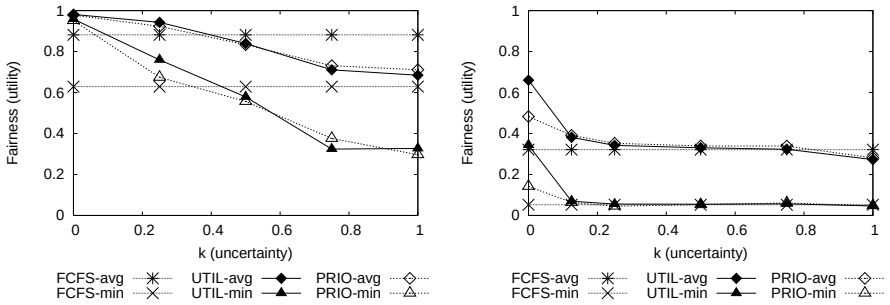
Figure 4.9: Utility with uncertain users in Mirage (left) and SDSC-SP2 (right).



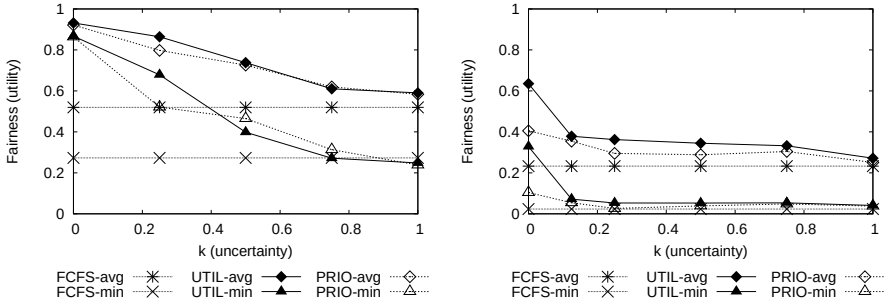
(a) Convex utility function decay



(b) Linear utility function decay

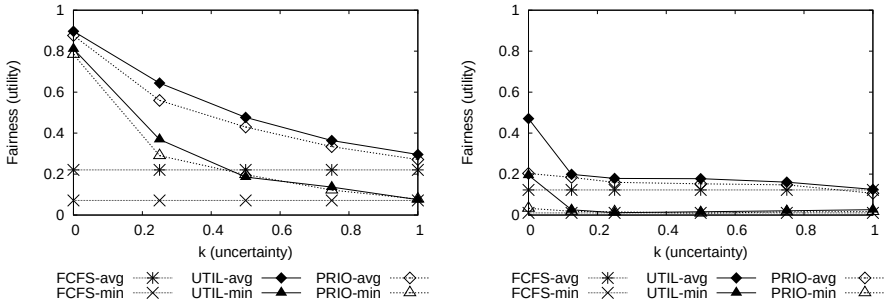


(c) Flat utility function decay

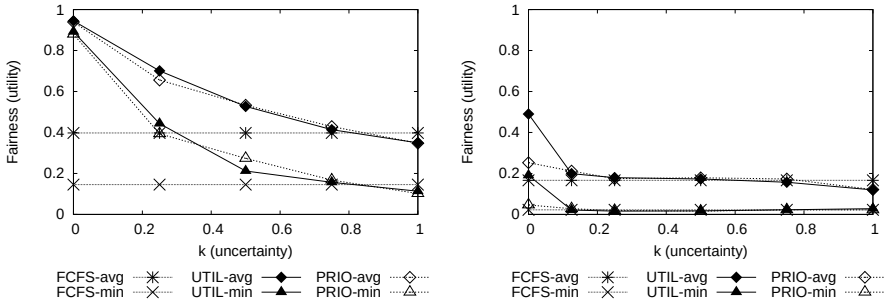


(d) Mix utility function decay

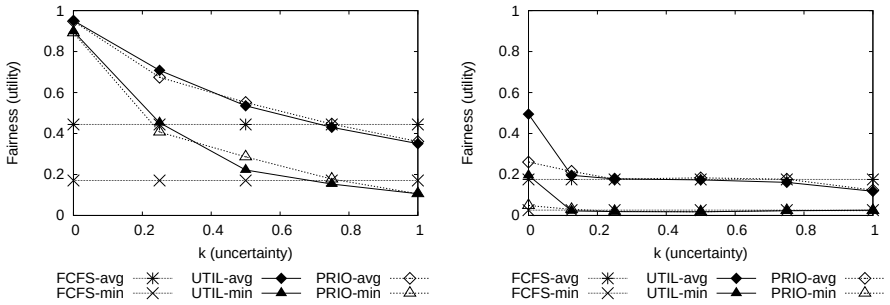
Figure 4.10: Fairness with uncertain users and *loaded* demand.



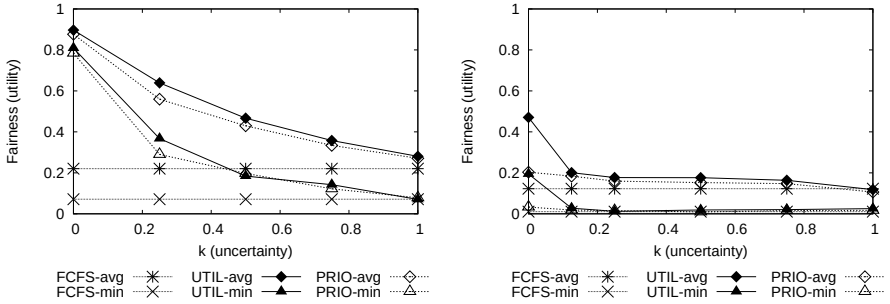
(a) Convex utility function decay.



(b) Linear utility function decay.

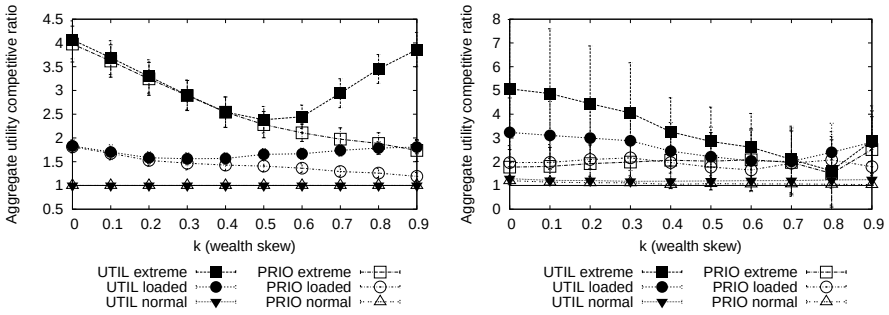


(c) Flat utility function decay.

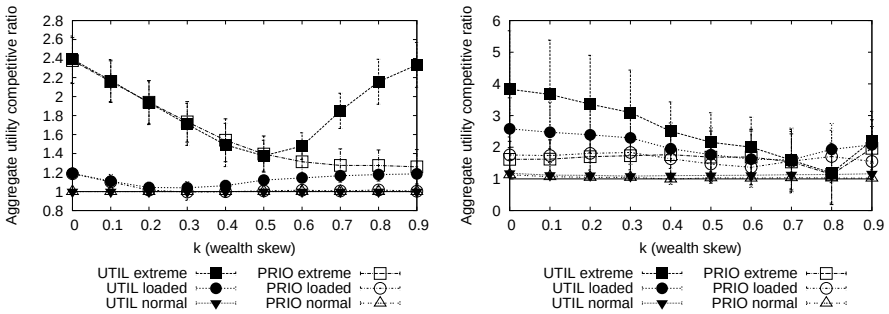


(d) Mix utility function decay.

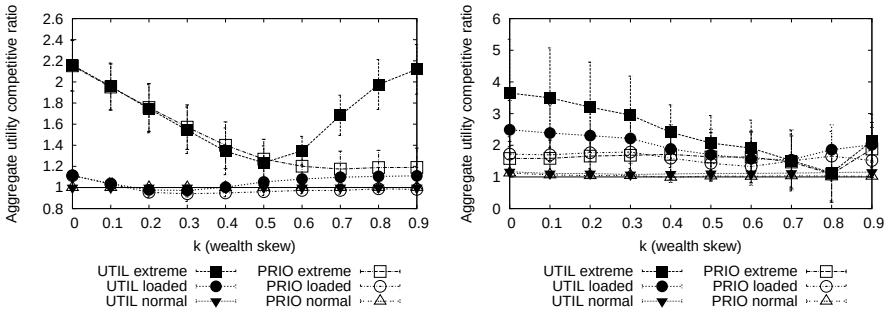
Figure 4.11: Fairness with uncertain users and *extreme* demand.



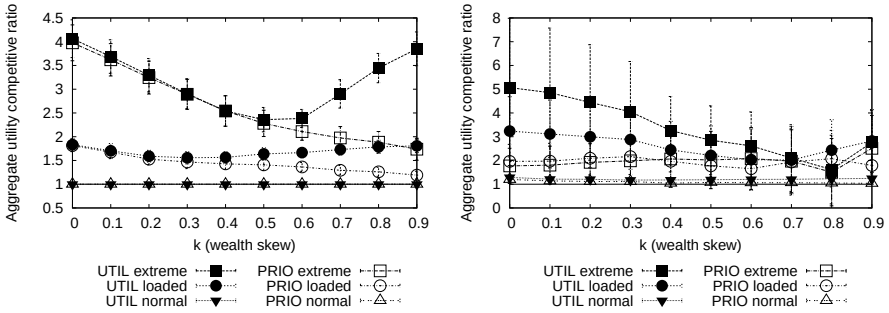
(a) Convex utility function decay.



(b) Linear utility function decay.



(c) Flat utility function decay.



(d) Mix utility function decay.

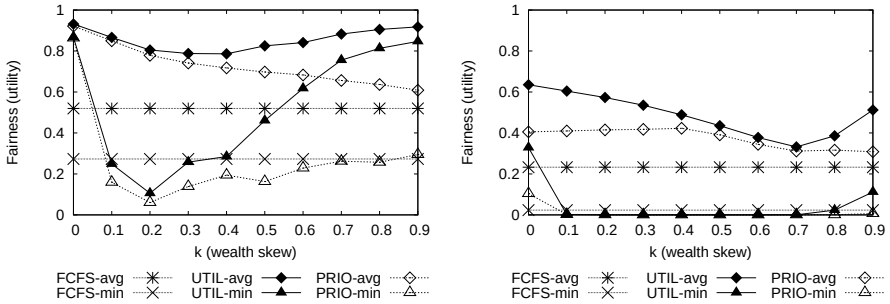
Figure 4.12: Utility with wealth inequity in Mirage (left) and SDSC-SP2 (right).

scheduled among the remaining users who are of the same wealth level. Therefore with *UTIL*, the jobs of these users will be prioritized based upon expressed utility, however with *PRIO*, all of the jobs are classified as having the same (low) priority, and thus scheduled in order of arrival, without regards to job value. For the SDSC workload, the competitive ratio for *UTIL* incurs its minimum at $k = 0.7$ (loaded demand).

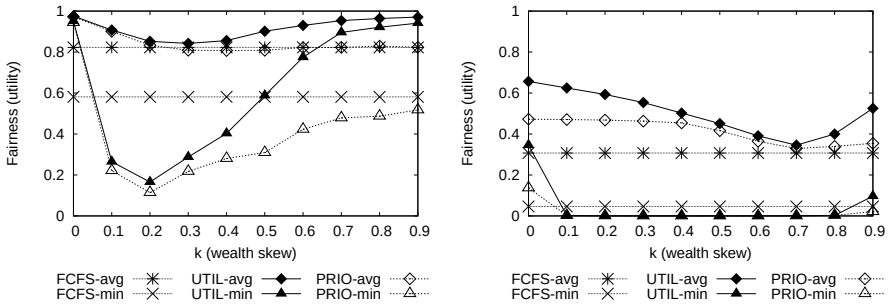
In Figures 4.13 and 4.14, we plot the minimum and average utility share under loaded and extreme demands, respectively (Mirage on the left, SDSC-SP2 on the right). Generally, we see that the *average* utility share received by each user is no worse with either market-based scheduler. However, we also see that the *minimum* utility share may be lower depending on k. Specifically, in Figure 4.13, we see that for $k = 0.2$ (Mirage), the minimum utility share received by the worst-case user is less than half of that received by the worst-case user in the FCFS approach. In the corresponding graph for the SDSC workload, we see that for $k > 0$ and $k < 0.9$, the worst-case user gets *no* utility share. In Figure 4.14, we see a similar effect, except that the "breaking point" occurs with a larger value of k. Also, as mentioned in the user uncertainty simulations, the SDSC-SP2 workload (right-hand column) presents more scheduling opportunities for a given load, and therefore more opportunities for error. Therefore, in general, we expect that the effect of wealth inequity is more pronounced than in Mirage.

Simultaneous Information Inaccuracy

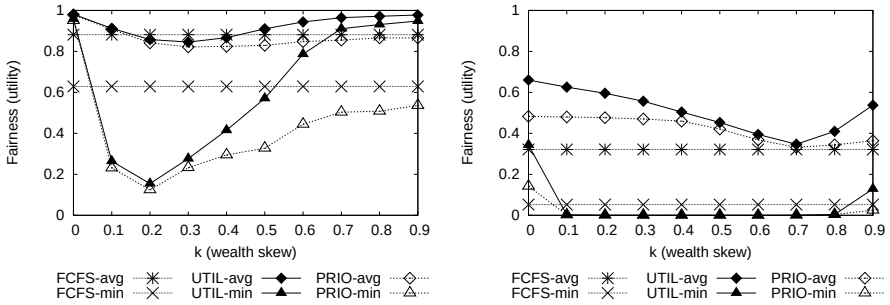
Finally, we examine the impact of a combination of user uncertainty and wealth inequity. In Figures 4.15 and 4.16, we plot the competitive ratio for each workload under a combination of uncertainty and wealth inequality levels, for users with a flat utility decay, and mix decay, respectively. In general, we observe that user uncertainty has a larger impact on the competitive ratio than wealth inequality. With steep utility function decays (Figure 4.16), *UTIL* is robust to both forms of inaccurate information, and *PRIO*, while robust to most forms of inaccuracy, exhibits performance that is sensitive to the incoming resource demand. However, as utility function decay becomes more shallow (Figure 4.15), both approaches are more sensitive, and therefore, less robust to inaccuracy. And, as we have seen in previous graphs, the difference between the utility-based scheduling approaches (UTIL and PRIO) increases with demand.



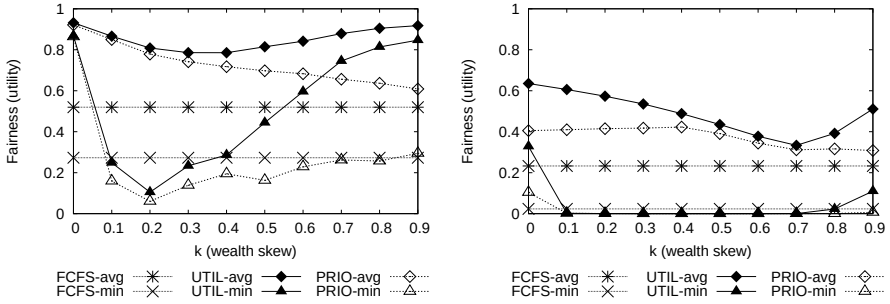
(a) Convex utility function decay.



(b) Linear utility function decay.

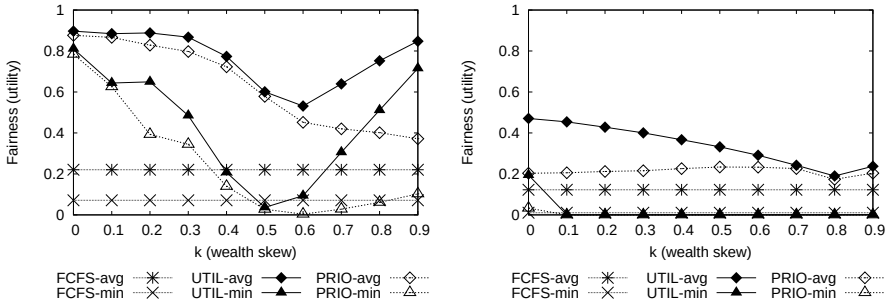


(c) Flat utility function decay.

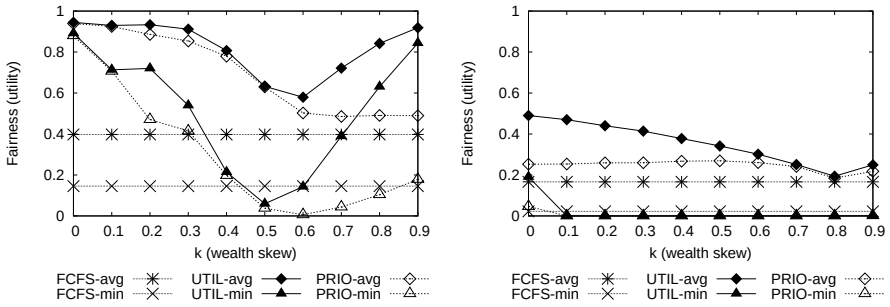


(d) Mix utility function decay.

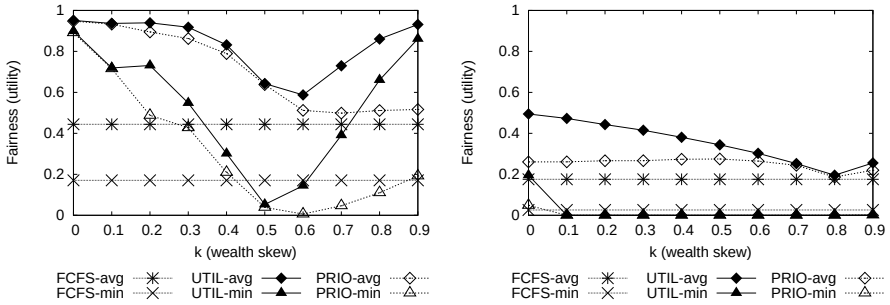
Figure 4.13: Fairness with wealth inequity and *loaded* demand.



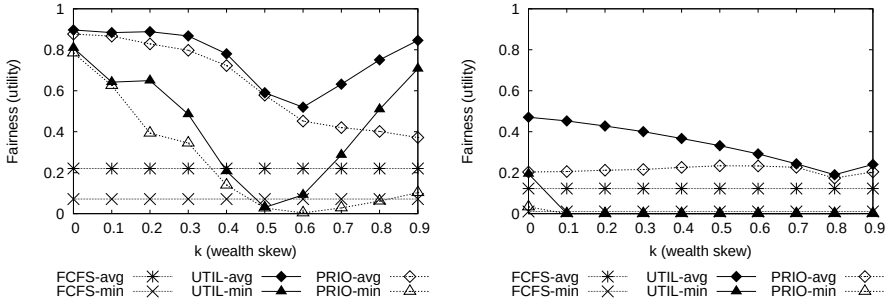
(a) Convex utility function decay.



(b) Linear utility function decay.

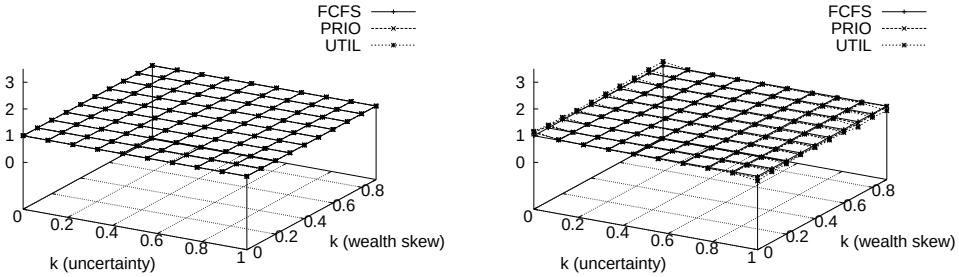


(c) Flat utility function decay.

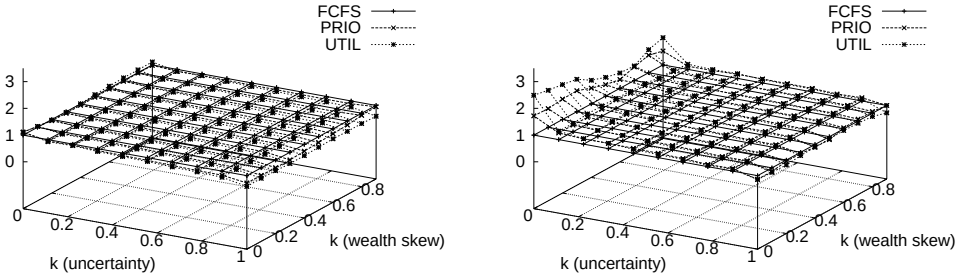


(d) Mix utility function decay.

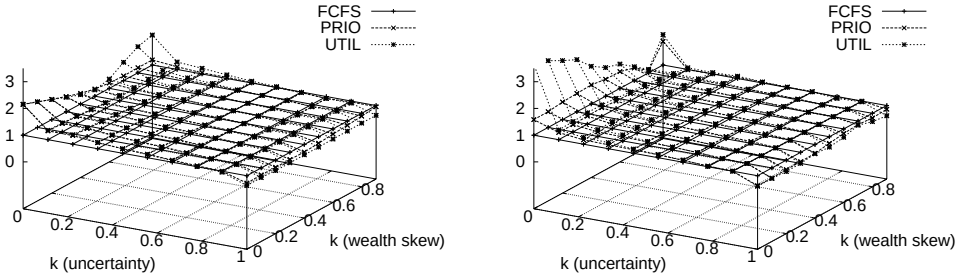
Figure 4.14: Fairness with wealth inequity and *extreme* demand.



(a) light demand

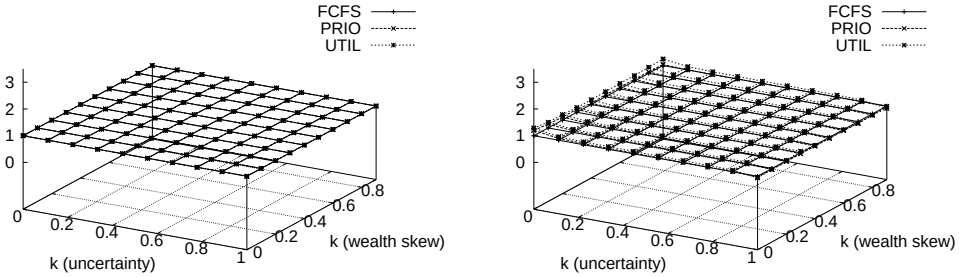


(b) loaded demand

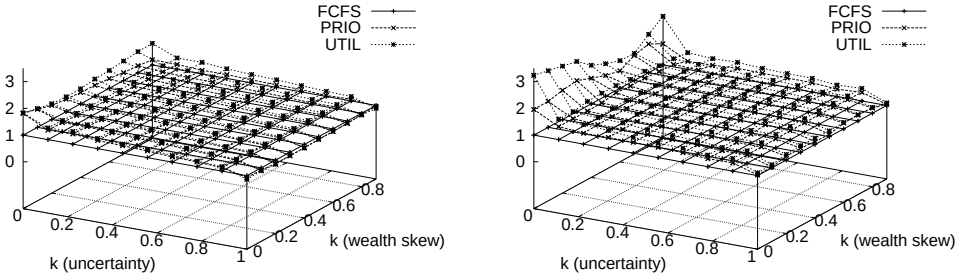


(c) extreme demand

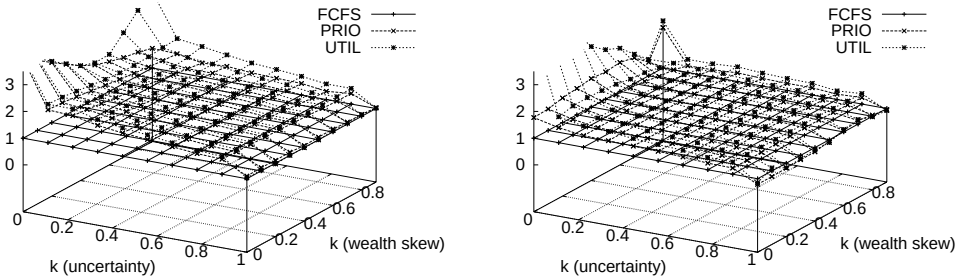
Figure 4.15: Competitive ratio with both uncertainty and wealth inequity, and users with *flat* decay; Mirage (left) and SDSC-SP2 (right).



(a) light demand



(b) loaded demand



(c) extreme demand

Figure 4.16: Competitive ratio with both uncertainty and wealth inequity, and users with *mix* decay; Mirage (left) and SDSC-SP2 (right).

Table 4.3: Uncertainty in job run-time estimates.

Cluster name	**Number of jobs**	k
SDSC SP2 [41]	54,006	**0.107**
SDSC BLUEHORIZON [41]	224,065	**0.165**
SDSC DATASTAR [41]	71,484	**0.111**

4.5.3 Levels of Inaccuracy in Practice

Our simulation results suggest that the robustness of a market-based scheduling approach depends significantly upon the distribution of utility function decay and the assumed value of k for information inaccuracy. In this section, we consider which decay model and values of k might be expected in practice. The study of SDSC users by Bailey Lee *et al.* [66] suggests that the *mix* decay most accurately reflects reality, particularly for the SDSC workload. Unfortunately, there is little available data to suggest what values of k occur in production systems.

For user uncertainty in particular, it is not possible to determine k from available workload logs. However, accuracy information about other user-specified job characteristics can provide a comparison point. For example, studies show that user-provided job *run-time estimates* used for backfill algorithms are inherently inaccurate [66].[1] Using our definition of uncertainty from Section 4.3 we list the corresponding values of k for job run-time estimates for both the SDSC-SP2 workload considered here, as well as two other supercomputing workloads, in Table 4.3. The magnitude of k in all instances is within the threshold for user uncertainty that a market-based scheduler can tolerate.

For the case of wealth inequity, we can use information about the distribution of virtual currency endowments to directly determine k. In Mirage, most users receive the same endowment of virtual currency, with a single group receiving twice as much as the others. Since the user population is restricted to fewer than 20 groups, this distribution results in an inequity level of $k = 0.09$, which is within the threshold for wealth inequity that a market-based scheduler can tolerate. While we lack similar data for SDSC-SP2, we have SU allocation data from other production clusters (Table 4.4) which show that k ranges from 0.75 to 0.85 [38]. From Figure 4.12, we see that these values of k in SDSC-SP2 may adversely impact the performance of a market-based scheduler. One way to

[1] All of our presented simulations assume perfect run-time estimates. However, repeating all of the experiments using the actual run-time estimates from users does not significantly change the results; we omit these graphs for space considerations.

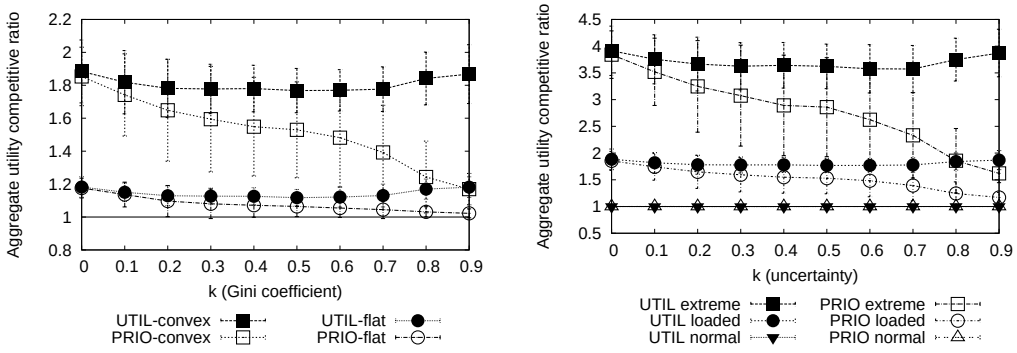
Table 4.4: Wealth inequity in SU allocations.

Cluster name	Number of groups	k
SDSC BLUEGENE [38]	1120	**0.824**
SDSC DATSTAR [38]	1818	**0.851**
SDSC IA-64 [38]	1758	**0.750**

mitigate the impact would be to limit the quantity of high-priority jobs across groups, or limit the accrual of service units by any group, as is done by the savings tax policy in Mirage. This policy, in effect, lowers the average k over tax-collection periods.

In our discussion of wealth inequity, we also observe that an inequity level of $k = 1$ (Figure 4.12) appears less harmful than a level of $k \approx 0.5$. This result is dependent on the wealthiest users not exhibiting enough resource demand to consume all available resource capacity.

In Figure 4.17(a), we illustrate that for the Mirage workload, a demand of $8x\cdot$ *loaded* is sufficient to evoke this worst-case behavior. At present, however, the wealthiest Mirage users consume roughly *half* of available resource for the duration of the trace. Based on these consumption patterns, Figure 4.17(b) illustrates the competitive ratio as a function of wealth inequity; we see that the competitive ratio does not degrade as severely as in theoretical the worst case.



(a) Inflated demand (load=$8x$).

(b) Actual Mirage demand.

Figure 4.17: Effect of wealth inequity and varying distribution of demand in Mirage (*mix* utility function decay).

4.6 Conclusions

One indirect result of our simulation studies is that we can corroborate the findings from previous simulation studies that a market-based scheduler can potentially increase the value delivered to users when compared to a FCFS+backfill scheduler, and we quantify this increase in real workloads at 10–300%, depending on demand and utility function decay. We also find that utility is distributed more equitably across users than in the case of a traditional approach, in both the average (80–450%) and minimum (45–700%) utility share per user.

More importantly, we find that these results are surprisingly robust based upon projections from real world data. Despite a theoretical worst-case performance of up to 50% degradation when compared against a FCFS+backfill approach, we provide projections from information error in real systems that suggest a market-based approach can indeed deliver more equitable values to users even under potentially imperfect operating conditions. Specifically, if we expect user utility information to be *at least* as accurate as observed inaccuracies in run-time estimates ($k < 0.2$ for all logs in Table 4.3), we can expect an increase in value of at least 20–100%.

4.7 Acknowledgements

Chapter 4 is in part a reprint of material that appears in the ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis, 2009, by Alvin AuYoung, Amin Vahdat and Alex C. Snoeren. The dissertation author is the primary investigator and author in this paper.

Chapter 5

Service Providers and Aggregate Utility Functions

In the previous chapters we allocate resources with the implicit assumption that maximizing user utility also maximizes aggregate utility. This assumption is adequate for the academic and non-commercial settings we've considered, but for commercial environments reflective of emerging cloud-based and service-oriented systems, we must also consider the utility of the resource service provider. In this chapter, we model our system as a job-execution service provider which, as before, executes jobs on the behalf of clients, but must also maintain profitability. We examine different market-based allocation policies to allow such a service provider to maintain profitability (that is, maximizing its own utility) when both user demand and the underlying availability of resources is uncertain.

5.1 Motivation

The explicit goal in Mirage and Bellagio is to improve aggregate user utility. We have demonstrated how combinatorial bids and job valuation information are used by clients[1] to communicate the value of a piece of work and other QoS aspects such as its timely completion. However, in both of these systems, allocation decisions are driven solely by user needs. In this chapter, we consider the more general case of a system where its needs, or profitability, is also critical. In particular, we demonstrate how to

[1]The terms *client* and *user* will be interchangeable in this chapter.

extend our model of an auction-based allocation mechanism to cope with such a setting, where we have both a utility-maximizing service provider (who in of itself, may be a client to a resource provider) and utility-maximizing clients. We find that as defined thus far, job utility functions on individual work items do not capture how important it is to complete all or part of a batch of items, which can result in reduced satisfaction for both client and service provider; to capture this need, a higher-level construct is required. We present a multi-job *aggregate-utility function*, and show how a resource service provider that executes jobs on rented resources can use it to drive admission control and job scheduling decisions. Using a profit-seeking approach to its policies, we find that the service provider can cope gracefully with client overload and varying resource availability (Figure 5.1). The result is significantly greater value delivered to clients, and higher profit (net value) generated for the service provider, the sum of which represents our previously defined notion of aggregate utility.

5.1.1 Examples

In the previous chapter, we consider the sensitivity of a market-based allocation algorithm in a parallel job environment: users have many parallel jobs to run, and the scheduling system prioritizes jobs based upon user utility information. When demand exceeds resource capacity, the scheduling algorithm can simply select among the jobs with the highest *per-unit utility* to maximize aggregate client satisfaction. We argue that this behavior does not adequately capture the needs of users in many commercial environments.

Consider the example of a user who has a workload consisting of 100 independent jobs. Each of these jobs is a part of a larger task, and he needs *at least* 75 of the jobs to complete (it does not matter which 75), but can tolerate having fewer than 100 complete. If he submits each job individually, he risks having fewer than 75 jobs completed, but if he sends all of the jobs as a large parallel job, he risks not having *any* of the individual jobs complete.

While a system lacking this type of expressiveness may still benefit users in the non-commercial or academic infrastructures that we have considered thus far, for a system that must sustain itself on profits and serving client needs, the approach in the previous chapters is not adequate. This client example is motivated by several others we have encountered in real systems:

- We have a colleague who often runs 100,000 jobs over a weekend on a shared compute cluster in order to perform an experiment. Each job takes a few minutes to run and produces one data point on a graph. The graph is nearly useless if too few data points have been obtained by Monday morning, but completing 90% is almost as good as completing all of them. No particular job is more important than any other — it is the aggregate set of results that counts.

- Computer-graphics film animators often compete with each other for access to a compute farm on which they run multi-hour rendering jobs overnight [13]. For any particular animator, getting a particular image back the following morning has some benefit, and having more images rendered is better — but sometimes the majority of the sequence needs to complete for any of it to be useful. Some frames are considerably more expensive to render than others, yet no particular frame is more important than its peers — it is the overall effect that matters.

- Outsourced business services often have service level agreements (SLAs) or contracts that include penalties for poor performance: if the response time is too high, for too many transactions, the service provider will earn less, and may even have to pay out more than it takes in. Sometimes, from the perspective of a user, no individual transaction in a set of work is any more important than any other — the percentage of transactions that violate the bounds is what is important.

Simple per-job or per-work-item information does not capture the true intent of the client in these examples, leaving the scheduler to do the best it can, but risking unhappy clients, under-utilized services, or both. In the absence of any higher-level control or incentive, the service provider is free to cherry-pick jobs, and accept only the most profitable ones, leaving the client at risk of not getting its less-profitable work done. To prevent this behavior, the client needs to constrain the service provider somehow. As the examples show, simple per-job metrics will not work, and imposing binary constraints of the form "you must finish all of these" would be sub-optimal in the presence of competition from other clients that may not be known at the time an agreement is made. What is needed is additional mechanisms that can express the client's desires while not unduly constraining the service provider. This chapter presents such controls — contracts and aggregate utility functions — and evaluates their behavior.

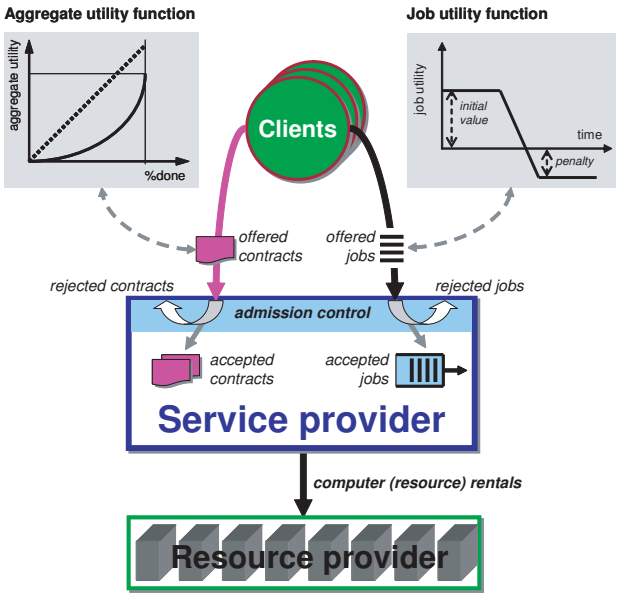


Figure 5.1: Problem overview: The relationship between clients, service providers, and resource providers.

5.1.2 Questions to Address

We focus our study by considering the concrete example of a *job-execution service provider* that runs batch jobs on behalf of its clients, who can be commercial clients, scientific partners, or other entities (Figure 5.1). We focus on the following questions:

- How can a job execution service provider remain profitable while also increasing user utility?
- How can a job execution service provider remain profitable when it must also consider underlying resource availability?

5.1.3 Summary of Results

The primary contributions of this chapter are to:

- Introduce *aggregate utility functions*, which are used as a part of a higher-level agreement between a service provider and client which let clients specify the overall value of completing a set of work, in addition to the values of individual work items.
- Present algorithms that allow a *service provider* to make both per-contract and per-job admission-control and scheduling decisions that take such *client* aggregate

utility functions into account.

- Evaluate these algorithms by means of a simulation study, in the context of a service provider that obtains resources from an external source, from which *resource availability* may be volatile.

Our evaluation covers a range of operating conditions: load, resource cost and quantity variability, per-job utility function shape, and aggregate utility function shape. Our experiments show that the new algorithms consistently extract higher utility for clients and higher profit rates for service providers than previous approaches.

The next section is a survey of broadly related work. Section 5.3 introduces our model of services and the contracts they support; section 5.4 describes our job execution service in greater detail; section 5.5 describes the resource provider service it uses. The evaluation portion of the chapter starts with a description of our setup in section 5.6 and is followed by our results in section 5.7. A discussion of the results and our conclusions close out the chapter.

5.2 Related Work

This chapter extends the previous work on Popovici *et al.*, which uses a system model similar to ours in which a service provider rents resources instead of owning them, and whose goal is both to decide on the *quantity* of resources to obtain, and the *allocation* of resources to client jobs. We directly build upon this work, but instead of exploring the effects of resource availability uncertainty, we explore the effects of known variability, and the problem of aggregate performance constraints, which has not been considered previously.

In this chapter, we discuss the use of client aggregate utility functions to control service provider behavior across multiple jobs. Kumar *et al.* [61] use a single utility function that aggregated data from multiple sources. We use both per-job and aggregate utility functions.

We use client contracts to indicate how a service provider should perform in the face of changing underlying conditions and conflicting service contracts. Balazinska *et al.* use off-line bilateral contracts [9] to specify service quality for distributed stream-processing applications; SNAP [35] performs service and resource allocations based on three levels of agreements for resource management in grid computing: application per-

formance, resource guarantees, and binding of applications to resources. Unlike SNAP, our job-execution service, rather than the client, determines how to bind applications to resources. Dan *et al.* [36] is similar, but also includes client level objectives and abstract resource objectives, and focuses on the service provider's need to manage resources and applications; however, they do not look at the effects of aggregate objectives nor the performance of a service provider to meet their objectives. GRUBER [39] uses contracts for job scheduling based on the amount of CPU a group is allowed to consume over a period of time, but does not assign values to individual jobs.

5.3 Models

In this section, we present our model for a service provider in a cloud-based or service-oriented world. We define each of the components in Figure 5.1, and how a particular service provider — in this case, our job execution service — communicates with clients and its own resource provider.

5.3.1 Contracts

In a cloud-based or service-oriented world, clients need control over their service provider's behavior, and service providers must be able to constrain the behavior of their clients. This mutual control is provided by means of *service level agreements*, or *contracts*, which specify the service to be provided, its quality and quantity levels (e.g., the load that the client can impose), price, and penalties for non-compliance.

Too much specificity in a contract may prevent helpful optimizations behind the scenes; too little leaves the the service provider to second-guess the intentions and desires of its clients, which exposes the clients to the risk of being surprised, disappointed, or both.[2]

For our job-execution service, each of our clients negotiates a contract with the service provider to run a single sequence of jobs. The client binds itself too, by including a description of the contract's workload in sufficient detail to allow its aggregate load and value to be estimated, but not the precise timings of when jobs will arrive, or their individual sizes or values. This description includes estimates of the number of jobs,

[2]Not everything needs to be explicitly specified in a contract: anything for which there is little risk of misunderstanding can safely be omitted. Ascertaining what is mutually understood is itself an interesting problem.

their sizes, arrival rates, and utility functions, in the form of distributions (in our case, the distributions used by our client workload-generators).

We model well-behaved clients that submit jobs that conform to the contracts they negotiate; coping with malicious clients is outside the scope of this chapter. For simplicity of exposition, each job demands only one processor; handling multi-node, moldable or reshapable jobs is a relatively straightforward extension.

5.3.2 Job Utility Functions

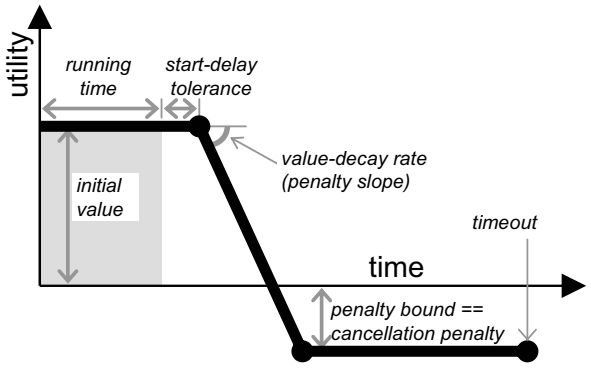


Figure 5.2: Per-job utility functions: How much value a job delivers as a function of when it is completed.

As before, each job has an associated time-varying utility function that expresses the maximum price that the client is willing to pay for that job to be run, and how this price decreases with elapsed time (see Figure 5.2). We follow our definition from the previous chapters, but in this chapter, we allow the utility function to decrease *below zero*, which reflects a negative penalty of not having the work complete. This change is necessary when considering binding contracts and service-level agreements between clients and providers. As with before, we equate the value of a job with the maximum price the client is willing to pay.

Once the job execution service accepts a job, it will either run it and deliver its results, or it will cancel it. If the job is not completed by its timeout, it is always cancelled. The job value to the client equals either the job's utility function value at the moment that the job completes and returns its results, or the maximum penalty if the job is cancelled.

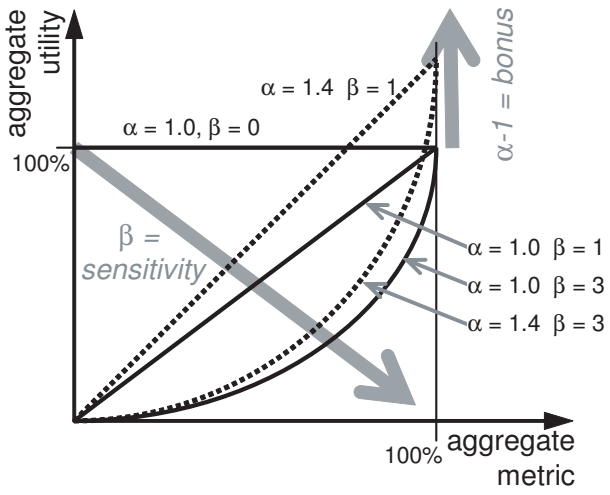


Figure 5.3: Aggregate utility functions: Some representative functions, showing the effects of changing α and β on the total pay-out for a contract.

5.3.3 Aggregate Utility Functions

In order to properly balance the desires of a provider to execute only the most profitable jobs, and the desire for clients to execute less-profitable work, we build on top of per-job utility functions and choose the following: the overall payment for a contract is the sum of the per-job prices multiplied by the value of an *aggregate utility function*, which is a function of an aggregate metric measured across the entire contract. This function allows the client to express near-arbitrary consequences for different aggregate behaviors in a simple way. We believe that this mechanism is simple, powerful, easy to communicate, captures important client concerns, and is easy for the service provider to interpret.

The aggregate utility function could be of nearly arbitrary shape. To explore a range of behaviors, we pick a family of functions that can be generated using only two parameters: $aggregate\_utility = \alpha x^{\beta}$ for an aggregate metric value x in the range 0–1 (see Figure 5.3). We call $(aggregate\_utility - 1)$ a "bonus" when it is positive, and a "penalty" when negative.

The parameter β is a measure of the client's sensitivity to the aggregate metric: when $\beta = 0$, the client is indifferent to its value; when $\beta = 1$, the client is moderately sensitive (the relationship is linear), and for higher values of β, the client is increasingly sensitive. When the aggregate metric is the fraction of jobs completed, then as β increases, the client is expressing increasing concern about completing all of the jobs; for

$\beta < 1$, as β approaches 0, the client is expressing increasing indifference to having the last few jobs run.

The parameter α describes the potential upside to the service provider of good performance: for example, with $\alpha = 1.4$ and $\beta = 1$, the client is offering a bonus of 40% of the sum of individual job utility values for completion of all the jobs in a sequence (the tallest straight line in Figure 5.3).

The overall pay-out for a contract is calculated at its end; we assume that payment can be deferred until then, and any disputes can be arbitrated by a third-party auditor [10, 15].

Here, we use the fraction-of-jobs-completed as the aggregate metric, but it could as easily be *any* such metric, such as the average job completion time, the average start-time delay, or even the correctness of the results.

Composite utility functions could certainly be constructed using more than one aggregate metric. Essentially, they become objective functions for the service provider, guiding its tradeoffs along different operating dimensions. Using such functions might be an interesting way to augment the penalty clauses that are traditionally used in SLAs to handle QoS violations for properties such as availability, reliability, correctness, timeliness, and security — but it remains future work.

We also believe that aggregate utility functions that span multiple contracts would be a powerful tool to capture concerns about overall customer satisfaction and most-favored customers; such functions are also potential future work.

5.3.4 Using Contracts in the Service Provider

Faced with a proposed contract from a client, a service provider has to decide whether to accept or refuse it. Once a contract has been accepted, the service provider is bound to it: it cannot be cancelled, although it can effectively be abandoned. Payment is determined by the combination of jobs completed and the aggregate utility function.

Even in the absence of penalties for refusing contracts, the service provider still faces a tricky question when a new client contract arrives: is accepting the new one likely to give it more profit than completing an already-accepted one that it might have to abandon, or even some possible future one? Remember that the contract details provide only estimates of future client behavior — the details of exactly which jobs will arrive when are unknown, for both the new and the existing contracts.

The number of resources available to the service provider may fluctuate with time, affecting which contracts it can service profitably. The desirability of abandoning an existing contract is affected by both the likely cancellation penalties for its jobs, and by what fraction of the achievable aggregate-level benefits have been achieved from the already-completed jobs: if it has nearly completed a contract with a high β value, it may well be worth completing it, because much of the payout will result from only a little more work. All these factors complicate the service provider's decision.

A similar problem occurs when the client submits a job: the service provider has to decide whether to accept it or not, bearing in mind the likely profitability of the job by itself, its impact on other work that it has already agreed to do, and the effect it might have on the aggregate utility function for the contract the job is associated with — or even other contracts, if those jobs have to be cancelled to make way for this one.

The next section describes some of the algorithms we use to solve these problems.

5.4 Job Execution Service

The primary metric we use to evaluate the job execution service is the *profit-rate* it achieves: the difference between its income and expenditures per unit time. Income corresponds to the utility (value) it delivers to its clients, as measured by what they pay; expenditures are its costs to rent processors on which to run the jobs. In turn, client value is specified by the combination of a contract's per-job utilities and the client's aggregate utility function.

We are also interested in the total client utility achieved, which we equate with the total value (utility) the clients pay — i.e., the service provider's revenue.

The job-execution service provider runs two types of admission control algorithms: one for client contracts and one for individual jobs. It also has a scheduler that decides when to run jobs. We discuss these algorithms in the remainder of this section.

5.4.1 Contract Admission Control

The *contract admission control algorithm* determines which client job-sequences to accept. Its purpose is to avoid long-term service over-commitments, and to establish a binding contract between the service provider and its client. The algorithm is run whenever a new contract arrives. It first runs a feasibility check to determine if it *can* accept the contract (i.e., it will be able to get enough resources to do so), and then a

profitability check to see if it *should* (i.e., if its profitability is likely to increase if the contract is accepted). If the contract passes both tests, it is accepted; if not, it is declined, and the client seeks service elsewhere. There is no penalty for refusing a contract, but once accepted, it is mutually binding on both parties.

The contract-feasibility check is selected from the following policies:

1. *contract-load=oblivious*: always accepts contracts.
2. *contract-load=average*: accepts a contract only if its average load plus the existing average load is within the predicted resource availability for the contract's duration.
3. *contract-load=conservative*: like average, but uses load estimates that are 2 standard deviations above the average, to provide some resilience to time-varying loads.
4. *contract-load=preempt-conservative*: accepts a contract if it passes the *contract-load=conservative* admission test, possibly by cancelling an overlapping contract that would generate less total revenue.
5. *contract-load=high-value-only*: accepts a contract if its expected value per hour exceeds a threshold. Setting the threshold requires knowing the expected value per hour of future contracts; however, running this algorithm provides a useful upper bound on profit.

If the contract is feasible, its profitability is then checked, using one of the following tests:

1. *contract-cost=any*: the cost to rent resources to execute the contract is ignored; no contract is rejected for this reason.
2. *contract-cost=variable*: profitability predictions are calculated separately for each different cost period and added; only contracts that increase the overall profit-rate are accepted.

In theory, multi-round negotiations could occur at the time a contract is offered [40]. For simplicity, we just consider a contract once, and accept or reject it as it stands: no attempt is made to adjust the contract to make it more acceptable to either side.

Once a contract is accepted, clients can submit jobs against it.

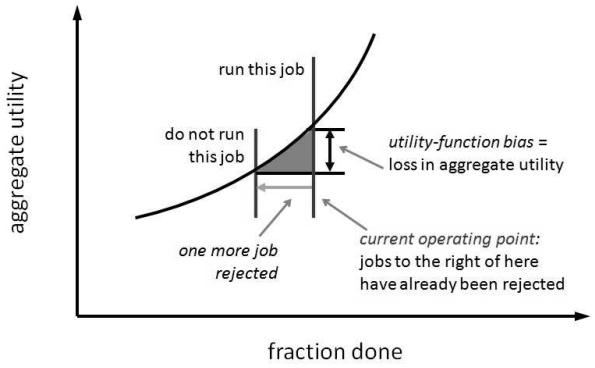


Figure 5.4: Calculating the effective utility bias: The bias is calculated from the loss in aggregate utility of not running a job, starting at the current operating point.

5.4.2 Job Admission Control

To avoid short-term overload, the job execution service provider also runs a *job admission-control* algorithm when a new job arrives, which decides whether it should accept individual jobs. Rejected jobs are not further considered, although they do affect the aggregate utility metric. Furthermore, we consider the case where some jobs are individually unprofitable: even if there were no other jobs in the system, the cost of executing them would exceed the job's utility.

If the admission control algorithm accepts a job, it is placed into a work queue, from which it is selected to be run at some future time by our job scheduler.

In the absence of aggregate utility functions, the job admission decision is made by comparing the profit rate of a tentative new schedule that includes the new job against the existing schedule that does not.[3] If the new profit rate is higher, the job is accepted.

Aggregate utility functions complicate job admission control: it may be more profitable to run an individually-unprofitable job than to reject it. A useful way to think about this scenario is to consider the cost to the service provider of *not* running a particular job. The lost aggregate utility may be larger than the cost of running a non-profitable job. Our solution is to make the job appear to be sufficiently profitable for it to be accepted and run.

We capture the increased desirability of a job by constructing an *effective* job utility function for it that includes a *bias* to the job's utility, and use it in admission and scheduling decisions, rather than the original utility function.

[3]As make-span for this calculation, we use the time to first free resource; using time to last job completed gave less satisfactory results.

Figure 5.4 shows how the bias is computed, and Figure 5.5 provides the equivalent pseudo-code. We first determine the current *operating point* (*old_fraction*), the fraction of jobs that would be finished if this job and all subsequent jobs were run to completion (i.e., 1 minus the fraction of jobs that have been rejected or cancelled so far). The bias is the drop in the aggregate utility function from completing one fewer job, multiplied by the expected (i.e., average) value of a job. It is added to all of the original job utility-function y-values to generate the effective utility function.

The effect of the bias is to assign much higher values to jobs that would have a large effect on the aggregate utility function if they were abandoned — for example, at an operating point near 100% for large-β functions (highly-sensitive clients) — but not to alter the values of jobs that are at relatively insensitive portions of the aggregate utility function.

As a concrete example, suppose that the contract specifies 20 jobs with a mean job value of 15, and further suppose that the service provider has already failed to finish 2 of 12 jobs so far. When the next job arrives, the operating point *old_fraction* is 0.90, the *new_fraction* is 0.85, and *potential* is 18. Therefore, the bias is $15 \times (18 \times f(0.90) - (17 \times f(0.85) - 1)$. If the aggregate utility function $f(x) = 2x$ ($\alpha = 2, \beta = 1$), then the bias is 37.5. A more sensitive aggregate utility function $f(x) = x^3$ generates a bias of only 25.2, because the 10% of jobs that were already missed have pushed the operating point to a place where the aggregate value is heavily degraded. The same $f(x) = x^3$ correctly generates a bigger bias of 41.25 when the current operating point is 1.0 because the downside of not running even one job is so large near the 100%-complete operating point.

The jobs' effective utility functions are only used to help the service provider make decisions: they are not used for charging the client.

5.4.3 Job Scheduling

Once a job is accepted, the service provider's job scheduler decides when it should be run. We adapt the *FirstPrice* algorithm from Mirage to consider variable cost; instead of prioritizing by per-unit value, we prioritize by per-unit *profit*, or profit-rate. We call this algorithm *FirstProfit*.

The job scheduler maintains a preferred schedule of pending jobs, and attempts to execute that schedule on the resources available to it. The scheduler is invoked

```
computeEffectiveUtility(
 input:
   U(t)       // original job utility function
   f(x)       // aggregate utility function
   total      // number of jobs in the sequence
   dropped    // number of jobs already abandoned
   mean_value // average for all [future] jobs

 output:
   U'(t)      // effective utility function
{
  // "old_fraction" is the position on the
  // aggregate-utility curve before this job
  // arrived - i.e., "x" in f(x)
  old_fraction = 1 - (dropped / total);

  // "new_fraction" is the new position on the
  // aggregate-utility curve if this job
  // were to be dropped
  new_fraction = 1 - ((dropped + 1) / total);

  // potential is how many jobs could complete
  potential = total - dropped;

  // bias is how far to boost the job's
  // effective value
  bias = mean_value *
          ( (potential     * f(old_fraction))
          - ((potential-1) * f(new_fraction))
          - 1 );

  // create the effective utility function
  U'(t) = U(t) + bias;
)
}
```

Figure 5.5: Calculating effective utility: How effective utility is calculated for a job.

whenever a job arrives, a job completes, or the number of available resources changes. If the scheduler decides that a job can be run, it selects a resource from those obtained from a resource provider, and assigns the first job in the schedule to it; this procedure is repeated until the scheduler has no more jobs that can be run or no remaining available resources.

Once a job starts running, we assume it will run to completion: it will not be preempted or aborted. To avoid getting tangled up in all the issues related to managing uncertainty, we deliberately assume that the job-execution time is known in advance, and construct the schedules so that the resource provider never needs to take away resources being used by a running job.[4]

If a job's start is delayed too long, its effective value may become negative. The scheduler will then cancel the job.

5.5 Resource Provider Service

So far in this dissertation we have assumed that the service provider owns the machines on which it runs its service. Besides the obvious disadvantage of representing a static capital investment in a single service offering, this approach biases the decisions made by the job-execution service towards maximizing the utilization of its processor nodes, even at the cost of declining marginal utility.

We believe that there is an alternative model that can cope with the realities of emerging environments, such as a service-oriented computing world, where service providers can be clients of other service providers. In particular, we model a job-execution service provider that rents compute nodes from one or more *physical resource service providers*, or just *resource providers*.[5]

Such resource rental has many benefits: the job execution service can scale up or down its capabilities as its business fluctuates; it does not have to be in the capital and operating-expense intensive business of running data centers; it can benefit from competition across multiple resource providers; and it can aggregate resources from several of them. There are disadvantages, primarily that the number and cost of resources avail-

[4]We realize that this is a strong assumption, but it is a conscious one, because it makes it easier to focus attention on the new results. Nevertheless, we believe that our results would be similar without this assumption — just harder to interpret, and with yet more workload parameters to set.

[5]The resources rented could be virtual machines rather than physical ones, as in Tycoon [62]. This represents a level of indirection that does not affect our story, other than to add the complexity of managing potentially dynamically-changing resource performance.

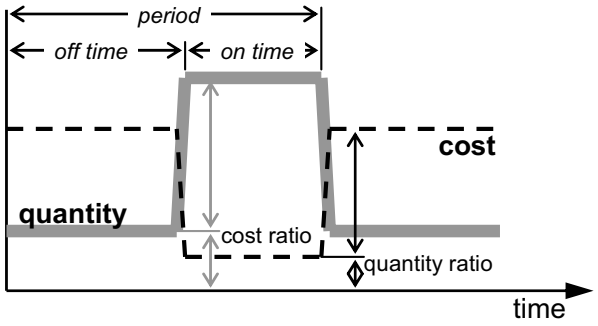


Figure 5.6: The on-off model for variable resource providers: "On" periods correspond to times of high resource availability and relatively lower resource cost.

able to the job-execution service may fluctuate as a function of other service providers' demands on the resource provider.

Since the resource provider is not the focus of this chapter, we adopt a model of its behavior that is sufficient to capture several of its salient behaviors and exercise the job-execution service provider's algorithms, while eliminating what we feel is unnecessary complexity.

Our *variable resource provider* models changing resource availability by alternating between *on* and *off* modes (see Figure 5.6), with more resources available in the former than the latter, and potentially different per-hour rental costs in the two modes. The special case of identical numbers and costs of resources in both on and off modes is called a *static* resource provider.

We construct different scenarios by considering a number of ratios between the properties of on and off periods:

- *quantity-ratio*: the number of resources in an on-period divided by the number in an off-period.
- *cost-ratio*: the cost of a resource in an on-period divided by the cost in an off-period — we expect that an excess of resources might cause the resource provider to lower its CPU-hour cost in times of plenty.
- *on-ratio*: the length of on-periods divided by the sum of on- and off-periods.

We always use equal-length on and off times (on-ratio = 0.5); the static provider has all other ratios equal to 1.0.

The resource provider offers accurate descriptions of how many resources it will have available at a specific time, using a set of tuples of the form <*start-time, duration, resource-quantity, cost*>. Prior work [91] has shown how to handle inaccurate resource capacity in a batch-scheduling system context such as ours, so we omit that feature here. We restrict this analysis to homogeneous processor resources; the techniques we describe can readily be generalized to handle multiple resource types. Finally, we note that the resource provider's revenue is the same as the job-execution service provider's cost.

5.6 Experimental Setting

Like our approach in Chapter 4, we rely on a simulation to study our job-execution service provider's behavior across a wide range of operating conditions, varying the offered workload and contract conditions, the policies and algorithms used by the service provider, and resource-provider behavior. This section describes our experimental setup, and the default parameters used in our experiments. The next section presents the results we obtained.

5.6.1 Simulator

Unlike our simulation-based environment in the previous chapter, we started by constructing a system with a set of independent Java processes to act as clients, a service provider and a resource provider. This system is able to perform real job-execution, for a set of "fake" jobs, with the intent that the code base can be easily adapted for use in a production environment. This code base forms the basis of a simulator, which mimics the behavior of the real system, and was used for all the results reported here.

In our experiments we use a single job-execution service provider, renting computers from a single resource provider, as this is sufficient to stress our algorithms.

5.6.2 Workload

Unlike the simulation study in Chapter 4, we do not base our simulation workload on trace-data. First, we are unaware of any existing workload data with client aggregate utility information. Second, our primary reference point is prior work in this space, which relies on mathematical distributions (as opposed to our sampled distributions in Chapter 4). Since we can characterize our workload with known distribution types, we

Table 5.1: Default simulator parameter-settings: The notation *distribution(x, y)* means a distribution of the given type with a mean of x and a standard deviation of y.

Parameter	Default value
Simulation length	1000 hours
Runs per data point	10
Client inter-arrival time	exponential(1.0) hours
Client (contract) duration	Gamma(100.0, 25.0) hours
Aggregate-utility α, β	1.0, 0.0
Job inter-arrival time	exponential(0.15) hours
Job length	Gamma(1.0, 0.25) CPU-hours
Low-value job value	Gamma(12, 2.4)
High-value job value	Gamma(36, 7.2)
Low:high-value clients ratio	80:20
Delay before value decays	1.5 × job-length
Mean decay rate (steep)	to 0 value in 1 job-length
Mean decay rate (shallow)	to 0 value in 5 job-lengths
Shallow:steep clients ratio	80:20
Max penalty value	= −(job-value)
Job-cancellation penalty	= max-penalty-value
Resources available	20 processors
Resource cost	10 per hour
On- and off-period duration	125 hours each
Quantity- and cost-ratio	1.0 (static)

list the important parameters in Table 5.1. We describe how each parameter is used in the text below.

We begin with a set of exploratory runs that are designed to tease out the behavior of the system under relatively straightforward conditions, before proceeding to more challenging situations. This first set of experiments establishes a baseline operating environment with the static resource provider: our goal is to set up a workload that has a reasonable profit margin (about 50%), operating at or near saturation on the available resources, while still rejecting some contracts and individual jobs. The first set of results we report present this behavior (see Section 5.7.1).

Since the most closely related work to ours is by Irwin *et al.* [54] and Popovici *et al.* [91], we opt to model the remaining parameters of our synthetic workload distributions upon their chosen simulation distributions rather than our Mirage data; this

choice allows for a more direct comparison of our work with their prior work. Each job utility function has a fixed value-decay rate (see Figure 5.2) that reduces the job's value from its initial value to its maximum penalty value; the decay starts at 1.5 times the job's running time. We use a mixture of high and low-value jobs and both shallow and steep value-decay rates. RiskReward[54] discusses how these synthetic loads relate to real workloads. Not all offered jobs can be executed profitably with the cost, computation time, and value settings used, even if the service provider is otherwise idle. We find that the Mirage data and the described data herein differ primarily on the distribution of job values: in these workloads, job values have a bi-modal distribution, whereas in Mirage, we observe a quad-modal distribution.

The default parameter-settings for our runs are shown in Table 5.1. Each client generates one sequence of jobs with an exponentially-distributed inter-arrival time. Such clients are created at exponentially-distributed inter-arrival times throughout the run, with a contract duration designed so that contracts may span on/off boundaries. Contract negotiation is simulated as occurring at the time a client is created; the client's first job is submitted one job inter-arrival time later.

We used Gamma distributions to generate bounded values such as job length or utility values.[6] Unless otherwise noted, all graphs present averages over 10 runs. We take care to avoid end-effects as much as possible. In particular, we run all jobs to completion, make the experiment duration much larger than the average job duration, and undo the effects of jobs that are incomplete at the end of a run.

We found that the execution times for the algorithms of the job execution service provider are small compared to the time for running the jobs.

5.7 Results

We now present our simulation results. We begin by establishing the baseline behavior for our system in the absence of aggregate utility-aware clients, and then add them, followed by varying the behavior of the resource provider.

[6]Gamma distributions are chosen with parameters such that they behave roughly like normal distributions but with the attractive property that they do not generate negative values. Using a normal distribution and suppressing such values would result in a new, not-quite-normal distribution with a slightly different mean than intended.

5.7.1 Baseline Performance

The policies used by the baseline service provider are *contract-load=oblivious*, and *contract-cost=any* for contract-admission and *first-profit-rate* for job admission control and scheduling.

Figure 5.7 shows how the service provider and clients behave in the absence of an aggregate utility function (or, more strictly, if the aggregate utility function is "indifference"), and with a static resource provider.

As the offered load increases, the overall utility delivered to both clients and service providers increases. The service provider costs (which equal resource-provider revenues) stop growing significantly at around 60 jobs/hour, at the same point that resource utilization saturates.

Revenue and profit also stop growing at resource saturation. The service provider finds and runs more higher-valued (and hence more profitable) jobs at higher loads, as shown by the breakdown of CPUs assigned to jobs of different values (Figure 5.7(b)). However, accepting higher-value jobs is achieved at the expense of cancelling more lower-value jobs and the net result is a flat profit curve.

Note that the service provider's tendency to cancel low-value jobs when higher-value jobs arrive is what motivated our desire for client aggregate-utility functions.

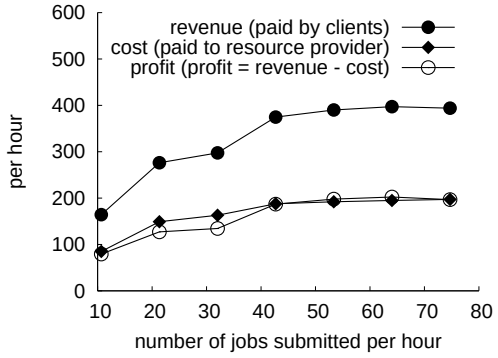
We used these first results to establish a baseline operating point for the remaining experiments. The parameters that resulted are shown in Table 5.1.

5.7.2 Aggregate Utility Functions

Figure 5.8 shows the effect of introducing client aggregate utility functions under different service-provider contract-admission policies (section 5.4.1). For this experiment, the aggregate utility function is $f(x) = x^2$ ($\alpha = 1$ and $\beta = 2$).

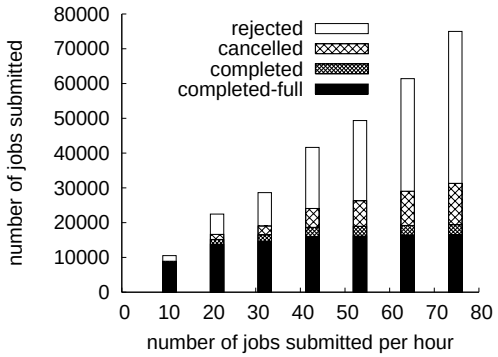
Figure 5.8(a) shows profit earned using each policy. In the presence of aggregate-utility aware ("sensitive") clients, profit drops dramatically as load increases for the oblivious admission control policy, which admits all contracts. The other three policies, which limit the number of contracts and hence the number of jobs submitted, see increasing profit with increasing load.

Figure 5.8(d) shows that while just as many (in fact, more) jobs are completed using the oblivious policy, there are also a lot of cancelled jobs. Fortunately, since first-profit-rate job scheduling always prioritizes the high-value jobs, virtually all (95-100%) of



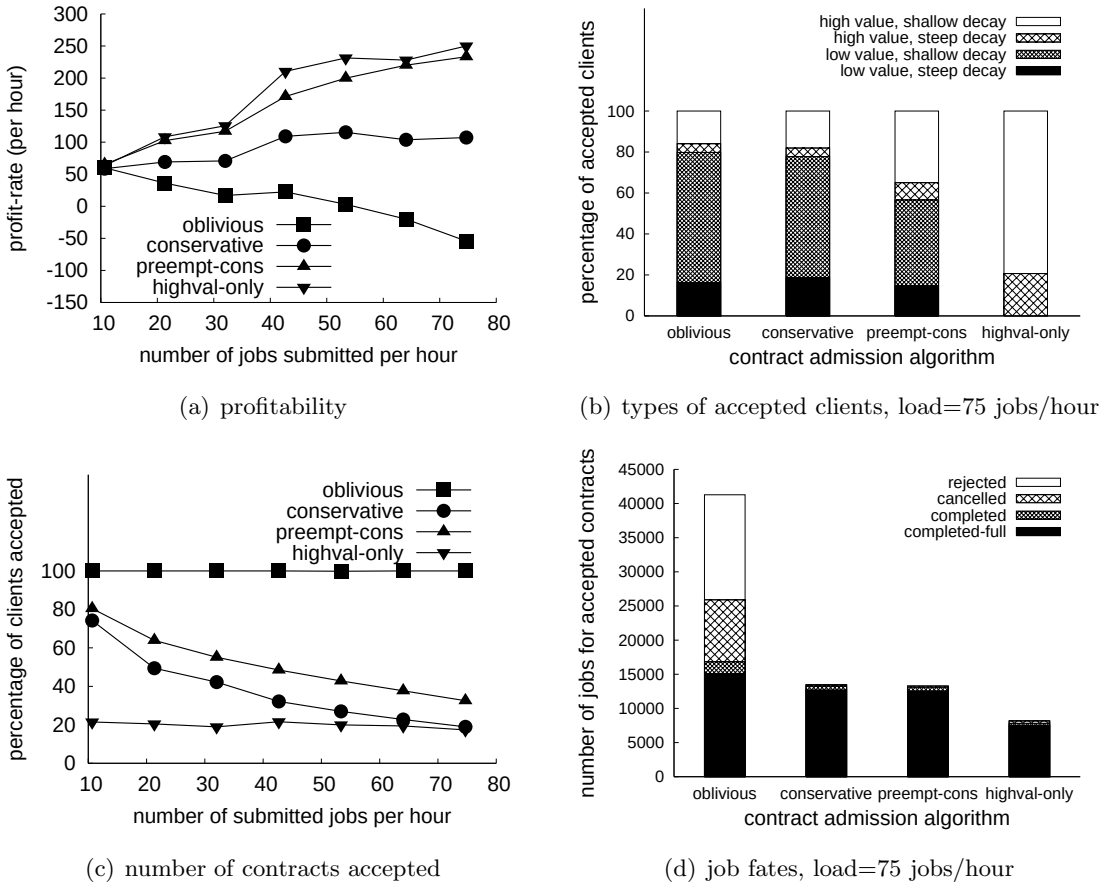
(a) revenue, profit and cost

(b) breakdown of CPU usage by job value-rate (per hour)



(c) breakdown of job fates; "completed-full" jobs are ones that earned their maximum value

Figure 5.7: Indifferent clients and a static resource provider: Client utility rate (revenue), service-provider profit rate, utilization and breakdown of utilization versus offered load for the baseline service provider.



(a) profitability

(b) types of accepted clients, load=75 jobs/hour

(c) number of contracts accepted

(d) job fates, load=75 jobs/hour

Figure 5.8: Sensitive clients: Exploring the effects of different service provider contract-admission policies.

the jobs for high-value contracts are completed. At the same time, a much lower fraction of jobs are completed for low-value contracts. Combined with the penalty incurred for cancelled jobs, the lower fraction causes the overall revenue earned from the low-value contracts to result in negative profit. In fact, for low-value contracts, if only 90% of the jobs complete, the aggregate utility function reduces an average job value of 12 to 10 (which equals cost). When fewer than 90% complete, the *completed* jobs are actually run at a loss. The preempt-conservative policy, by contrast, finishes over 90% of the jobs for most of the low-value contracts that it accepts.

Figure 5.8(c) shows how many fewer contracts are accepted using the policies that monitor and limit load. The unrealistic high-value-only policy accepts the 20% of contracts that are high-value and completes nearly every job at full value, as seen in Figure 5.8(d). However, job arrival is bursty enough that even this policy cannot

complete every high-value job, which is why there are small bands of completed (late) and cancelled jobs at the top of the bar. Furthermore, the much smaller number of jobs completed using the high-value-only policy as compared to the preempt-conservative policy shows that not enough contracts are accepted when using the high-value-only policy to keep the resources utilized.

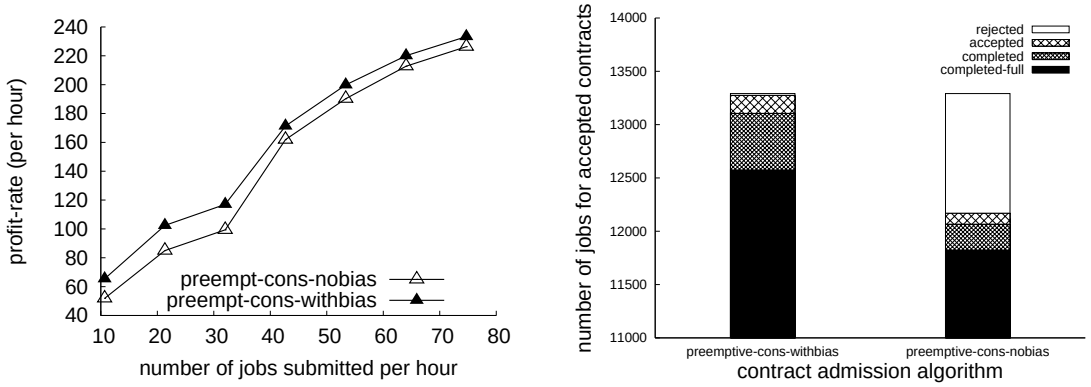
The two policies that consider load when performing contract admission, conservative and preempt-conservative, accept and complete about the same number of jobs. However, Figures 5.8(b) and (c) show that preempt-conservative is able to accept more contracts. By abandoning some contracts in favor of more profitable contracts that arrive later, the preempt-conservative algorithm sees a higher percentage of high-value jobs. These high-value jobs then have a large positive impact on its profit.

Note that profit for these experiments is lower for all policies than in the baseline experiments. By choosing an aggregate utility function where $\alpha = 1$ and $\beta = 2$, the effect of the aggregate utility function is always to diminish revenue. In other experiments where $\alpha = 2$, omitted here for lack of space, we see much higher (nearly double) profit.

All the runs shown in Figure 5.8 use the effective job utility function described in Section 5.4.2 when constructing schedules, both for job admission and job scheduling decisions. The benefit of using the bias calculated by this function is shown in Figure 5.9, which compares the performance of the preempt-conservative policy with and without the bias. Adding the bias improves profit because about 5% more low-value jobs are accepted and run: while these jobs are individually unprofitable and hence rejected without the bias, completing these jobs raises the percentage of jobs completed and hence the aggregate utility bonus enough to more than compensate for their individual losses.

The results shown in Figure 5.8 for $\alpha = 1$ and $\beta = 2$ are similar with other values of $\beta > 1$, as shown in Figure 5.10. For larger values of α and β, adding the bias calculation has a larger impact on the service provider's profitability. This greater profitability is important for clients as well. Client satisfaction is measured in terms of value per accepted contract, and as we can see from Figures 5.8(a) and (c), the algorithms earning a higher profit-rate also deliver a greater level of client satisfaction.

We use these results to establish an operating point for the experiments relating to variability in the resource provider. For the remaining set of experiments, we use the *preempt-conservative* contract admission policy.



(a) profitability (b) job fates

Figure 5.9: Job admission and scheduling: Enabling and disabling the effective job utility function.

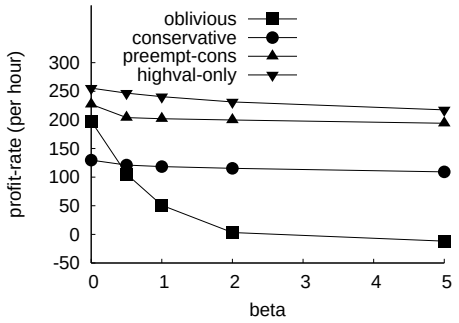


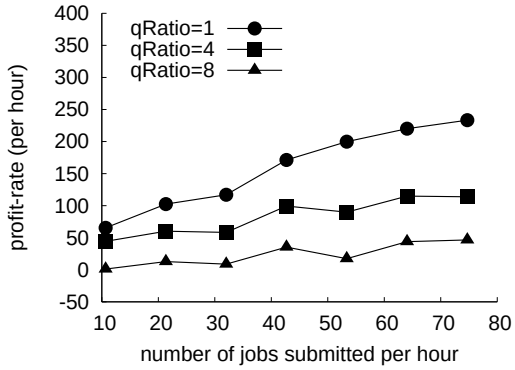
Figure 5.10: Sensitivity of aggregate-aware clients: Exploring the effects of different choices of β ($\alpha = 1$).

5.7.3 Static and Variable Resource Providers

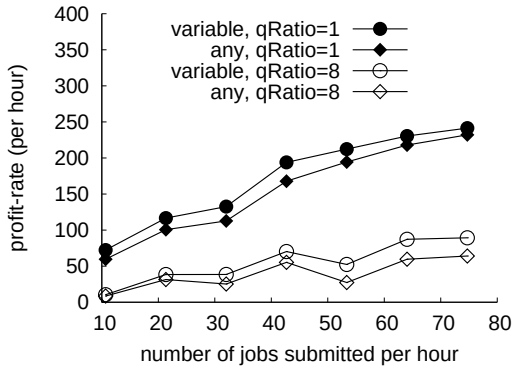
Figure 5.11 shows the effect of varying the resource provider behavior (Section 5.5) for the two contract-admission profitability tests described in section 5.4.1.

Figure 5.11(a) shows the results of fixing the resource costs, but varying the available resource quantity by changing the resource provider's quantity ratio (labeled as *qRatio* in the figures). A quantity ratio of 1 corresponds to the static resource provider used in the previous experiments.

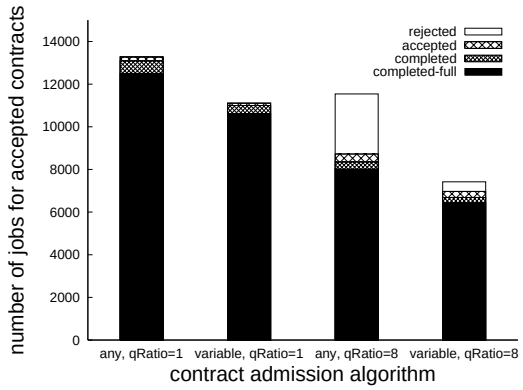
As the quantity ratio increases, the profit rate of the service provider decreases. We hold the average number of resources constant, so when there are more resources in an on period, the number of resources in the off period drops accordingly. For a fixed level of demand, there is an abundance of profitable jobs during the low quantity periods



(a) Cost-ratio = 1; the results are not affected by the choice of profitability test.



(b) Cost-ratio = 1/8; profit rate for two different profitability tests.



(c) Cost-ratio = 1/8; job fates for two different profitability tests, load=75 jobs/hour

Figure 5.11: Aggregate-aware clients and a variable resource provider: The effects of different quantity-ratios and contract-admission profitability algorithms; $\alpha = 1, \beta = 2$.

and not enough jobs during the high quantity periods to use all of the resources. When the quantity ratio is 8, the contract completion rate is so low that most of the jobs from low-value contracts incur a loss, regardless of whether or not the job was actually run. This loss completely offsets the profit from the high-value contracts, which are also completing an average of only 85% of their jobs rather than the 98% they complete with static quantities of resources.

Figure 5.11(b) shows the effect of variable resource costs (by varying the cost-ratio) and also illustrates the benefit of turning on the profitability check from Section 5.4.1. As in Figure 5.11(a), increasing the quantity ratio decreases the profitability of the service provider. However, both with and without variable quantities of resources, the *contract-cost=variable* admission control policy outperformed the *contract-cost=any* policy, especially at higher offered load, since the provider is able to complete a larger fraction of jobs for a given contract. Note that a higher degree of client sensitivity (i.e. larger α or β) would increase the profit gap between the two policies.

5.8 Conclusions

In this chapter, we explore the effects of profit-aware algorithms, and study how load, aggregate utility functions, and the number and cost of resources influence our service provider's profit. We show that our profit-aware approach more thoroughly addresses the problems for both the provider and the clients we outline at the outset of this chapter. Specifically, we demonstrate that a *preempt-conservative* contract admission algorithm with a *variable cost* profitability prediction, and the *effective-utility bias* job scheduling approach is a more effective technique across a wide range of conditions than previously proposed resource allocation algorithms. Together, these techniques represent a good balance between effectiveness, robustness, and ease of implementation and execution.

Based on our results, we also make the following additional high-level observations:

- The idea of a self-interested, profit-aware service provider is a powerful technique for thinking about, generating and selecting algorithms, and avoiding imprecision in defining a "good" outcome.

- The contract-admission control algorithms we developed seem to be quite effective,

and our evaluation highlighted how important careful selection of work is for a service provider.

- Successfully handling aggregate utility functions is a new result. Doing so also increases client utility, which is an important result in its own right.

Our results demonstrate the importance of profit-aware schedulers and admission-control algorithms, and include cases where a decent profit can be obtained in place of losses from less profit-sensitive algorithms. In the context of our previous results, we demonstrate that the notion of maximizing the aggregate utility of a set of clients using a market mechanism can be extended to a two-sided setting where a utility-maximizing service provider and a set of utility-maximizing clients can coordinate using simple market-based mechanisms, such as service contracts, aggregate utility function, and a heuristic auction-based allocation policy.

5.9 Acknowledgements

Chapter 5 is in part a reprint of material that appears in the Proceedings of the IEEE International Symposium of High Performance Distributed Computing, 2006, by Alvin AuYoung, Laura Grit, Janet Wiener and John Wilkes. The dissertation author is the primary investigator in this paper.

Chapter 6

A Market Mechanism for MapReduce

In this chapter, we return to the goal of improving resource allocation but this time, focus on a real *time-sharing system* using market-inspired techniques. Fundamentally, a time-sharing system presents additional resource allocation challenges not faced by a batch-scheduling system. For a user, applications compete with unknown resource demand, and therefore, she may find it difficult to reason about the value of a particular resource share. Also, a resource provider in a time-sharing system must plan capacity for the number of concurrent applications allowed to run on a particular machine; the unknown resource consumption characteristics of the competing applications makes this capacity planning problem difficult.

The primary goal of this chapter is to demonstrate that we can improve the quality of resource allocations for a set of jobs by approximating a utility function for jobs as well as the resources they use. Unlike previous chapters, we do not measure quality of resource allocations using an abstractly-constructed quantity of aggregate utility, but rather, use job *makespan* as a proxy for aggregate utility. Using makespan as a proxy for aggregate utility allows us to focus on this goal, and consideration of additional definitions of aggregate utility (such as the user-provided information we consider in previous chapters) should simply be an extension of these techniques, which we defer to future work. With a goal to minimize makespan (we assume that aggregate utility is a function of makespan), we observe that a job has a preference to complete as quickly as possible, and likewise, a machine has a preference to maximize utilization of its resources,

such as CPU or I/O. Based upon these observations, we define utility functions for each entity, and use this utility information to drive the allocation decisions in our time-sharing system.

To address the resource allocation challenges of a time-sharing system, we implement task profiling support within a Hadoop scheduler to perform informed capacity planning and prioritize applications based upon its resource consumption characteristics. This mechanism allows a resource provider to simplify the problem of capacity planning in the face of unknown resource demand, and for applications to be scheduled based upon resource preferences without requiring a user or application to explicitly provide this information to the scheduler. These techniques infer utility information from both a machine's resources, and the jobs that consume those resources. We demonstrate how these market-inspired techniques improve system throughput on a Hadoop scheduler running real jobs.

6.1 Motivation

The deployment of Bellagio is motivated by the problems stemming from the best-effort, proportional-share resource allocation in PlanetLab. In Chapter 3, we see that the usage costs of Bellagio on users coupled with the existence of other free resources do not justify Bellagio's sustained support in PlanetLab. In this chapter, we simplify our implementation mechanism and investigate its applicability to another time-sharing scheduling framework where we can exercise full control over resource allocation: Hadoop. To begin, we consider the fundamentally unique problems of allocating resources in a time-sharing system relative to the batch-scheduling counterpart we consider in the previous chapter.

From the perspective of a service provider, the primary difficulty of a time-sharing system is dealing with over-subscription of resources. This problem is unique to time-sharing systems because batch-scheduling systems allocate a single task to a machine at a time. In time-sharing systems, applications run on a machine concurrently, and the system scheduler decides how many concurrent application instances to allow at any given time; this choice can impact application performance, resource utilization, and a variety of other relevant metrics.

Consider the example of a type of application which periodically performs a computation and writes its output to disk. Without loss of generality, assume it spends half of its time using the CPU, and the other half, primarily writing the output to disk.

In a batch-scheduling system, both the CPU and I/O subsystem would remain idle for roughly half the time. In a time-sharing system, two applications of the same type can run simultaneously, and theoretically provide better resource utilization. Typically such increased resource utilization manifests itself in increased application performance, such as decreased makespan (time for the entire workflow to complete) or average response time (average time for each workflow component to complete). We will refer to this problem as *capacity planning*.

From the perspective of a user, a time-sharing system may provide more immediate access to resources compared to batch-scheduling systems, but if the system is over-committed, a time-sharing system increases the uncertainty of application performance since the quantity of competing background traffic is typically unknown. In the case of PlanetLab, even the presence of tools to monitor machine resource usage does not encourage applications to use machines with under-utilized resources in lieu of over-subscribed machines [87]. Ideally, a user would like to enjoy the benefits of a time-sharing system, without suffering from the possible performance degradation resulting from poor admission control. Therefore, in a large-scale time-shared system, users must carefully select the machines upon which to try and execute their tasks. We term this problem *task resource assignment*.

PlanetLab pushes both problems to the users, allowing access to each machine to be (theoretically) unbounded, such that users can instantiate an application on a machine at any time [87]. As we mention earlier, PlanetLab users have neither the mechanism or incentive to constrain usage, leading to severe CPU over-subscription on some nodes (Chapter 3, Figure 3.4), which leaves the system poorly utilized [87]. Currently, PlanetLab institutes a limit of 1000 slivers (we can think of these slivers as applications) running on any single machine, which is a heuristic designed to also preserve a minimum amount of disk space and memory for existing tasks. Bellagio uses a market framework to limit the number of simultaneous users on a machine through prices, but is not able to do so since market users share the machine with non-market users.

Instead of continuing to push with PlanetLab, we consider applying market-based allocation ideas to Hadoop, which, as an open-source scheduling framework, we can create our own instance to fully control resource access. The simple job-execution framework of Hadoop allows us to concentrate on the challenges we outline earlier in

the chapter. First, unlike typical job scheduling frameworks, which are batch-scheduled, Hadoop jobs are time-shared, and the number of concurrent jobs, or tasks, that occupy an execution machine (called a *TaskTracker*) is a hand-tuned parameter. Furthermore, the default best-effort scheduling policy used in Hadoop prevents users from controlling when or where their jobs are executed, thereby allowing us to focus on the problem of designing the scheduler to control the environment properly, rather than anticipating how end-users or applications may behave.

In this chapter, we will focus on providing a general approach to capacity planning and resource selection in a time-sharing system, and use Hadoop as our proof-of-concept implementation.

6.1.1 Questions to Address

In this chapter, we focus on the following specific questions:

- How can a time-shared system plan capacity to balance resource utilization and machine load?
- Do real applications exhibit different resource preferences?
- How can a time-shared system use this information to execute jobs that satisfy a task's preferences without explicit fine-grained input from the user?

6.1.2 Summary of Results

We have implemented a market-based mechanism for use with the Hadoop Map-Reduce scheduler. Based upon the implementation, we demonstrate the following results:

- We implement support for task-profiling in Hadoop which allows us to make capacity planning decisions that increase the total task throughput of each machine.
- We show that real applications exhibit distinct resource preferences, primarily across Disk I/O and CPU consumption, and perhaps surprisingly, these differ across comparable task types in jobs.
- Using no input from the user or applications, we infer resource consumption preferences of applications, and increase the throughput of real applications on a fixed resource capacity.

To re-emphasize our point in the opening paragraph of this chapter, unlike previous chapters, we do not rely on exogenous utility or priority information, but rather focus on leveraging the underlying utility preferences of applications to increase their own efficiency.

6.2 Background and Related Work

In this section, we discuss the Hadoop system, the MapReduce paradigm, and research efforts that are related to the work done in this chapter.

6.2.1 Hadoop and MapReduce

Hadoop is an implementation of a job-execution system based upon the Google MapReduce [37] paradigm. This system is used primarily for running large, highly parallel, data-intensive jobs on a cluster (or network) of commodity hardware. The idea behind this paradigm is that cheap, commodity hardware can be used as an alternative to expensive, high-end hardware, if the operations of the job can be partitioned into smaller, more manageable pieces.

Applications written for Hadoop use a specific programming model, which supports two high-level operations: `map` and `reduce`. At a high level, these operations define how to parallelize the job into smaller tasks, and how to aggregate the output of each task. For example, consider a job that must sort a large data set of words. Hadoop first splits the input data into smaller, more manageable data blocks. Hadoop then creates a set of *map tasks* which operate separately on each data block. In this example, the map operation merely describes how to take an arbitrary block of the original input data and transform it into sorted data. Hadoop then creates a set of *reduce tasks*, which take the output data from the map tasks, and aggregates them to create an output of sorted data from all of the blocks. In this example, the logic of the reduce operation is such that the intermediate sorted data produced by the map tasks is logically aggregated into sorted data from the original input data.

The primary distinction between Hadoop and traditional parallel batch clusters is that it is explicitly designed to run on commodity hardware. Since failures may occur more frequently in such an environment, and hardware may have different performance profiles, Hadoop handles task fault-tolerance, task assignment and re-execution seamlessly.

A side-effect of the task partitioning in Hadoop (and MapReduce) is that a single task usually does not require a dedicated machine, so tasks are typically time-shared. This choice leaves many parameters available to tune for any instantiation of the system. In this chapter, the primary parameter we are interested in is the capacity of each task-execution machine in the system. The default parameter in Hadoop is hard-coded to a maximum of 2 concurrent map tasks and 2 concurrent reduce tasks, with this setting based upon intuition from traditional workloads. Production systems can hand-tune this parameters to fit with their workloads. However, this methodology as a whole implicitly assumes that map tasks are similar and reduce tasks are similar, across jobs. In a later section, we will demonstrate that this is not necessarily the case, even for simple workloads.

6.2.2 Related Work

There are two primary research efforts that are related to ours. The first is the Tycoon resource-allocation system [62]. Tycoon is a rate-based market-inspired system for allocating resources for time-shared jobs. In this system, individual users are provided with a finite bank account with which they can "fund" their applications. Since jobs are time-shared, users must fund each resource on the machine: the CPU, network bandwidth, memory capacity and disk storage. Using machine virtualization technology, the Tycoon market (which runs independently on each machine) allocates resources to an application based upon its bid relative to all other applications in the market. Since time-shared applications typically do not have a well-defined execution time, the funding is defined as a rate, which periodically drains currency from the application user's bank account. As a proof-of-concept, Tycoon has recently been applied to a set of Hadoop jobs [98] with the goal to improve resource allocations based upon their preferences. Our work differs from Tycoon because our system does not require fine-grained user input; instead, we rely on the system to infer resource consumption patterns of applications to prioritize jobs and plan TaskTracker capacity. Also using Tycoon with Hadoop relies on the existence of virtualization technology to segregate applications, whereas we do not assume the existence of any such technology on the Hadoop machines.

Another related effort is Spawn, which is a market-based system for applications in a time-shared cluster or distributed system[109]. The primary distinction in our work is that we explicitly consider the resource consumption of multiple bottleneck resources,

such as CPU and I/O — which are relevant to typical Hadoop workloads — whereas Spawn primarily considers the problem of scheduling for primarily CPU-bound jobs, and therefore, resolves only CPU contention.

Tangentially related to our work are efforts to improve the performance of the Hadoop scheduler. In addition to the work by Sandholm and Lai [98], there are multiple efforts by Zaharia *et al.* to increase the performance of the default Hadoop scheduler. One effort considers intelligent re-execution of tasks in heterogeneous environments using the LATE scheduler [119], and another effort adopts heuristics for task placement and ordering to leverage observed patterns in Facebook workloads [118]. We believe that these efforts are complementary to ours, and can be used to improve our techniques further; this chapter merely demonstrates the value of our approach.

Finally, Weinberg *et al.* [111] investigate the ability for users to provide application bottleneck information to improve system throughput in a batch-scheduling domain using *symbiotic space-sharing* (coined and developed by Snavely [102]). Our work differs because our target domain is a time-sharing Hadoop system instead of a batch-scheduling domain, which gives us flexibility in improving system throughput by controlling system load as well as task placement. Furthermore, instead us relying on user-provided input, we infer this information. Finally, we provide an implementation which uses real Hadoop jobs.

6.3 Our Approach

The two primary challenges we wish to address in this chapter concern *capacity planning* and *task resource selection*. We argue that resource utility information from the perspective of both the executing machine and the task is useful to make intelligent decisions in both. We validate this argument by comparing the traditional approaches used by a real Hadoop system with real jobs, to our implementation which gathers utility information to make its decisions. In the next section, we will discuss our basic experimental setup with Hadoop, the Hadoop jobs, and our scheduling mechanisms.

6.3.1 Experimental Setting

Using simulation to characterize the impact of job allocations in a time-sharing system is more difficult when compared to a batch-scheduling system because of the induced interaction between co-scheduled applications (i.e., jobs which reside on the

same host machine). Unlike the previous chapters, where we rely upon simulation, in this chapter, we present results from experimental runs from a real Hadoop system. We use the source code from Hadoop release `0.19.1`, the most current stable release when we began this investigation.

The workloads we consider in this chapter consist of three common jobs in Hadoop applications: `grep`, `sort` and `wordcount`. Each of these jobs themselves are comprised of several map tasks and reduce tasks. More complicated Hadoop jobs can consist of several jobs themselves. We use the implementation of these three jobs that are bundled with the Hadoop source code distribution. We believe that these common operations represent typical building blocks of larger Hadoop jobs, and that with some additional work, these results can generalize somewhat to larger, more complex, Hadoop applications.

A `grep` job takes as input a set of data (a particular file or set of files), and an input string (defined upon the same character set as the input data), and returns the number of instances of that string in the input data. A `sort` job takes as input a set of data (defined over a previously understood alphabet), and returns the same set of data in sorted order (the order of data members is also predefined). A `wordcount` job is similar to the `grep` job, but it counts the number of occurrences of *each* word appearing in the input data. Each of our workloads will consider a combination of these three jobs.

6.4 Capacity Planning

In this section, we investigate the use of resource consumption information to improve capacity planning in Hadoop.

6.4.1 Capacity Planning in Hadoop

In general, time-sharing systems adjust load based upon a domain-specific heuristic: for example, by identifying a maximum saturation level for a particular bottleneck resource (e.g., memory in Xen for the creation of additional VMs [11]), or by setting a limit to the maximum number of concurrent processes based upon past experience (e.g., PlanetLab [87]).

Hadoop also relies on domain-specific knowledge for setting resource capacity limits. Since the granularity of its work is the map and reduce tasks of each job, the system administrator can define two parameters: one each for the maximum number of

concurrent map and reduce tasks to execute. The default Hadoop implementation has a centralized scheduler which acts a single queue for tasks. The scheduler assigns tasks to TaskTrackers with an intent to balance load across the system, while never exceeding an individual machine's pre-defined load limit. Since a map task in any application is viewed as a fundamentally different type of task from a reduce task, their limits are parameterized independently.

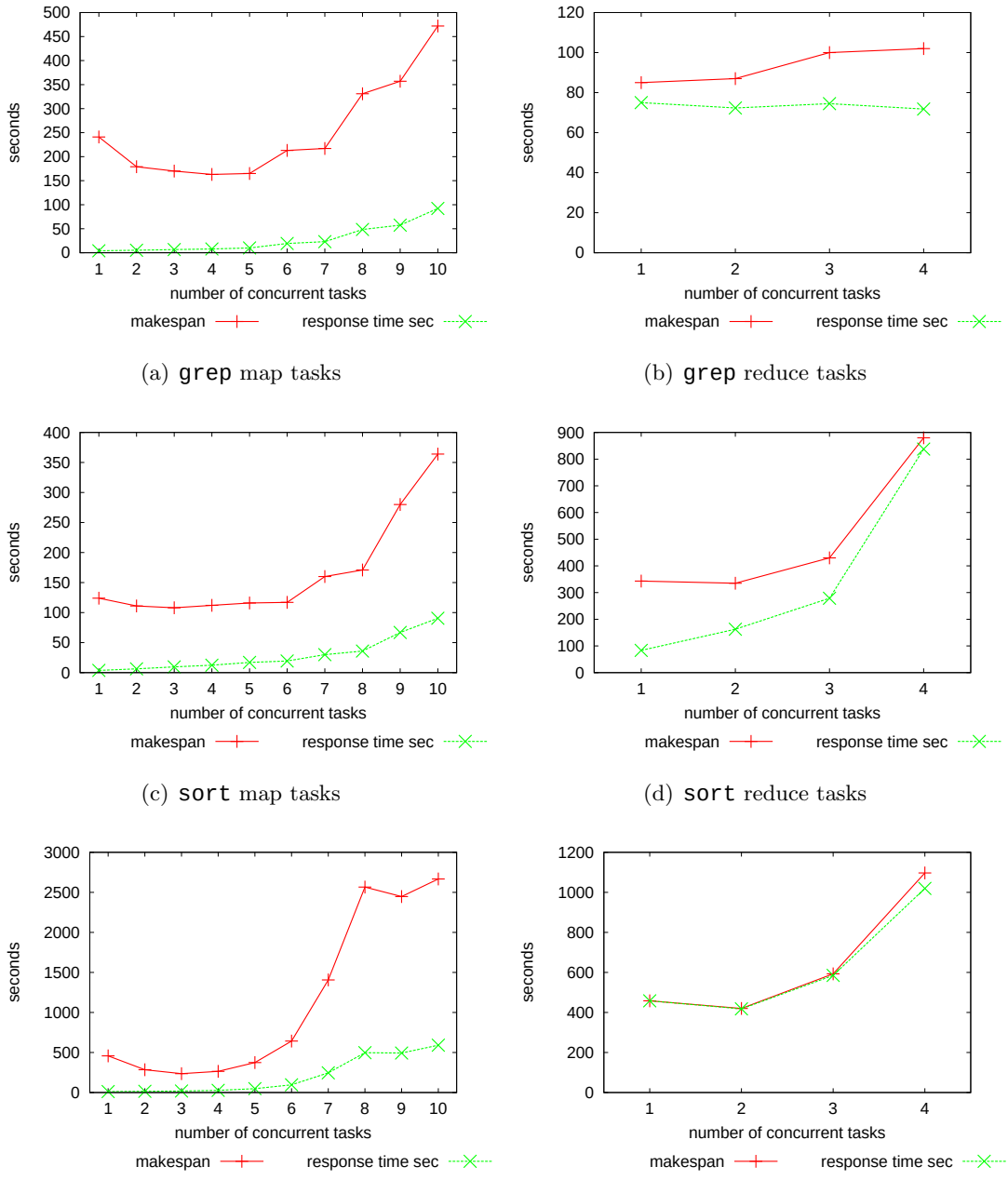
Baseline Measurements

To illustrate the challenge of capacity planning even with domain-specific knowledge of map tasks and reduce tasks, consider the problem of setting the number of map and reduce tasks in a simple 1-node, 1 job-*type* Hadoop system. We expect that increasing the number of concurrent map tasks may slow down the response time of the average map task (since it must now share the machine), but may also reduce the overall makespan of the collection of map tasks (makespan is defined as the total completion time of all tasks, which is calculated as the elapsed time between the start of the earliest task and the completion of the latest task). At some point, we would expect the makespan to *increase* when load is sufficiently high if the resources on our Hadoop TaskTracker become over-provisioned. We expect similar behavior for reduce tasks.

In Figure 6.1, we vary the number of allowed map (or reduce) tasks (we refer to this number as *load*) from 1 to 10 (Hadoop defaults to 2), and compare the average response time for each of the map tasks, and the makespan for the collection of tasks. We plot each job's map tasks and reduce tasks as a separate set on each column of the graphs (we will continue to treat map tasks differently from reduce tasks in our graphs since they are treated as different entities in Hadoop), and plot the results for each job by row.

We first notice that the average response time for tasks in each job generally increases (or stays the same in the case of a `grep` reduce), but the "breaking point" load of each makespan varies, depending on the job. For example, sort map tasks generally benefit from concurrency between a load of 2 and 5, and grep map tasks generally benefit from concurrency between 2 and 5, but a wordcount job performs best (notice the larger y-axis range) at a load of 3. Across jobs, we see that the reduce tasks perform best with a concurrency load of 2.

In general, we notice that the results in these graphs generally corroborate our in-



(a) `grep` map tasks

(b) `grep` reduce tasks

(c) `sort` map tasks

(d) `sort` reduce tasks

(e) `wordcount` map tasks

(f) `wordcount` reduce tasks

Figure 6.1: Makespan and average response time of map and reduce tasks as a function of load.

tuition about load. These results are merely intended to demonstrate that static resource provisioning with a diverse workload (i.e., any combination of our three jobs) is likely to leave resources either under-utilized or over-provisioned, and from the standpoint of maximizing task throughput, there is potential to improve capacity planning.

6.4.2 Waste-Aware Capacity Planning

In order to minimize makespan, we define the utility of a resource to be a function that increases with its utilization. We hypothesize that increasing resource utilization also decreases job makespan. We argue that planning capacity based upon resource availability can direct efficient resource utilization without relying on static resource provisioning. In this section, we present such an algorithm for capacity planning, and demonstrate how its choice of load can lead to increased resource utilization (measured by job throughput), without explicit input from a system administrator in our Hadoop system.

Defining Waste

First, we motivate the design of our algorithm by defining a notion of *resource waste*. We define the waste of a particular resource as a combination of its idle time and its overhead time spend managing load (i.e., which increases as the number of queued tasks increases). The illustration in Figure 6.2 illustrates a typical scenario of a time-shared CPU running multiple concurrent tasks: CPU utilization — measured as the amount of useful work performed by the CPU — increases as idle time decreases, but typically reaches a point of over-saturation for a particular number of concurrent tasks, after which, the CPU begins to thrash, and utilization decreases.

Ideally, we would be able to determine the point of maximum CPU utilization (immediately preceding the thrashing threshold), but this point is dependent on the characteristics of the running tasks, which we do not know a priori. Instead, we try and dynamically adjust the number of concurrent tasks to minimize the idle time and overhead. Intuitively, an individual time-sharing machine would like to minimize idle resource time if there is remaining work to do. At the same time, a time-sharing machine does not want to schedule so many tasks so as to thrash between them, and therefore, would also like to minimize the queuing time that tasks spend on it (one way to think about this is that a waiting task can otherwise be using an available resource on another

machine, or at the very least, not causing congestion for other resources on the machine it is currently blocked on). We do not consider job migration in this chapter since job migration has significant costs in a practical implementation.

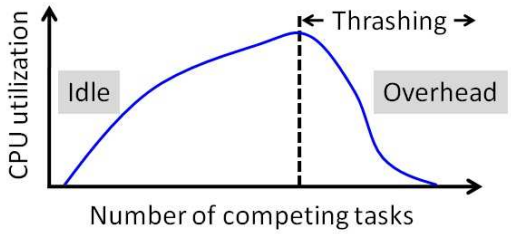


Figure 6.2: CPU utilization as a function of number of tasks.

Also, we acknowledge that different definitions of this waste metric may make sense for other domains. For example, a provider may wish to minimize power consumption of a resource scheduler, or I/O bandwidth. Extensions to these other definitions are a straightforward extension of our algorithm, and simply require supporting mechanisms to dynamically account for each unit of waste with respect to a task.

Waste-Aware algorithm

We implement a scheduler with *waste-aware* capacity planning, which adjusts TaskTracker capacity based upon perceived waste. Based on our definition of waste, the scheduler seeks to adjust load to *minimize perceived waste* on each TaskTracker. At a high-level, our implementation requires an additional mechanism for each Hadoop TaskTracker that records node resource usage information (gathered through the Linux `/proc` and `taskstats` interface in kernel version 2.6.27, with the *task delay accounting* feature enabled). Our scheduler executes tasks with different map/reduce parameter settings, periodically sampling the total waste in the system. In our implementation, we define waste as the sum of idle CPU time, and time spent waiting for any of three resources: memory, I/O and the CPU. The scheduling algorithm operates under the assumption that there is an inflection point, similar to the graphs in Figure 6.1, and tries to compare different load regimes before choosing a particular setting. Since there are possibly many different task states to explore, we start with the default Hadoop settings of 2 map tasks and 2 reduce tasks. The algorithm proceeds to explore different spaces and when sufficient samples are taken, selects the task allocation quantity that

minimizes sampled waste. We present pseudo-code for this algorithm in Figure 6.3.

6.4.3 Experiments

In this section, we examine the ability of our waste-aware scheduler to both minimize waste, and to increase resource utilization compared to the default Hadoop scheduler. First consider a simple workload with only one job: `grep`. We again use a 2-node Hadoop cluster with 1 TaskTracker so as to simplify the problem of inducing resource contention. In Table 6.1, we compare different configurations of the default Hadoop scheduler to our scheduler implementation. We see that our scheduler, which uses waste-aware capacity planning, provides *more throughput* than each of the different default Hadoop settings by better utilization of resources. In Figure 6.4, we plot the behavior of a TaskTracker over time. On the left y-axis is the number of maps or reduces running at any given time, and on the right y-axis is a measure of *waste*, as defined previously. However, in order to make these units more meaningful, we plot to metrics over time: *relative waste*, which is waste as a fraction of average CPU run-time, and *cpu run*, which is the average CPU run time during that time interval. We see that the waste-aware scheduler eventually settles on a capacity for the single `grep` job, with an average waste that is lower than or similar to that of the other Hadoop configurations. Note that the `cpu run` and `rel waste` curves cross near the end of the workload, which indicates the scenario when mostly reduce tasks remain running on the TaskTracker, which utilizes less CPU than I/O.

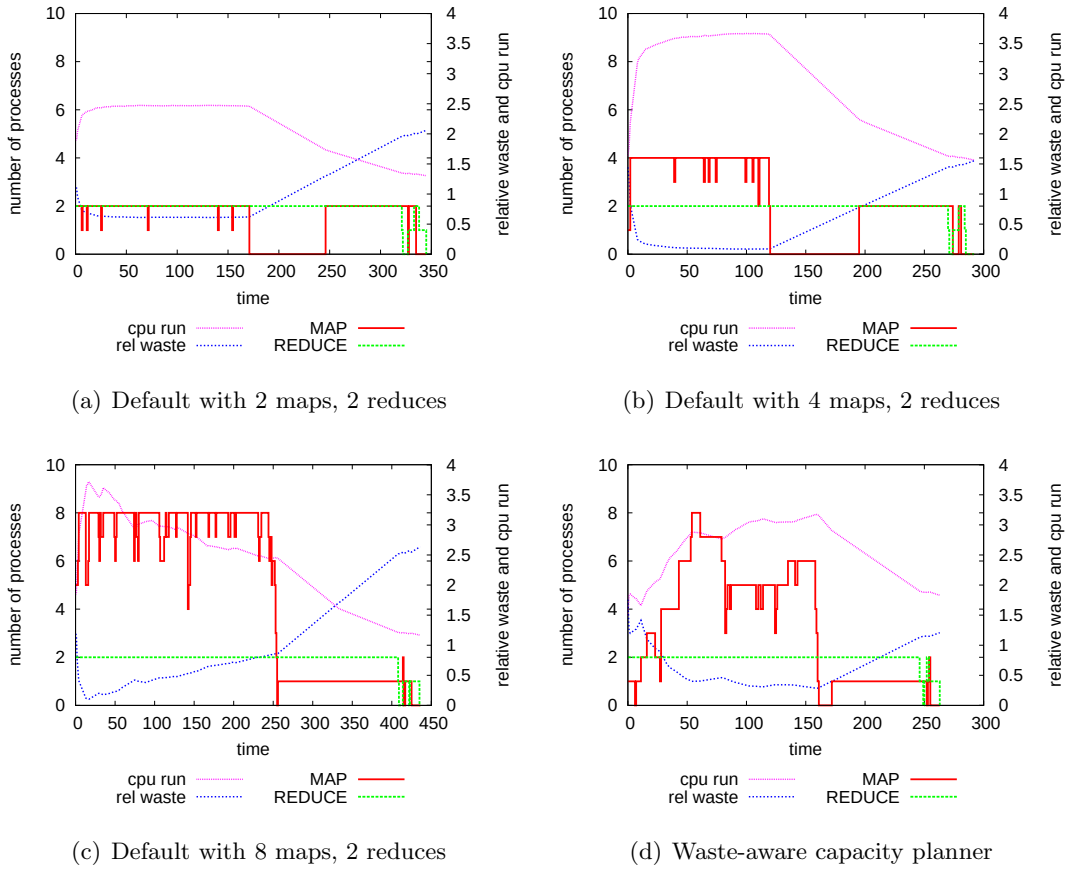
Table 6.1: *Makespan* and *average resource waste* of default Hadoop configurations and waste-aware capacity planning using 2 x `grep` workload.

Scheduler	Maps	Reduces	Makespan	Average Waste
Default	1	2	478	324.165
Default	2	2	347	287.236
Default	3	2	316	271.312
Default	4	2	**297**	**258.994**
Default	5	2	294	260.639
Default	6	2	315	293.891
Default	7	2	308	296.480
Default	8	2	441	425.412
Default	9	2	704	958.795
Default	10	2	890	1288.61
Waste-aware	–	–	**229**	**204.714**

```
desiredNumMaps(
 input:
   currentReds   // current number of reduces running
                 // on TaskTracker
   WasteRecords  // hash table mapping <#maps,#reds>
                 // to data structures recording waste
 output:
   m         // desired number of maps and reduces

  // default <m,r>
  optMaps = 2;
  optReds = currentReds;
  // see if we have any records
  if WasteRecords.hasRecords() == false then:
    return optMaps;
  // see if we have an ideal
  minWaste  = ∞;
  for <m,r> in WasteRecords.keys():
    record = WasteRecords<m,r>;
    if record.enoughSamples() and record.getWaste() < minWaste:
        minWaste  = record.getWaste();
        optMaps = m;
  // try one below optimal if we haven't yet
  // (boundary case of m>0 not shown for brevity)
  if WasteRecords<optMaps-1,optReds>.enoughSamples() == false:
    return optMaps-1;
  // try one above optimal if we haven't yet tried it
  if WasteRecords<optMaps+1,optReds>.enoughSamples() == false:
    return optMaps+1;
  //
  return optMaps
)
```

Figure 6.3: Pseudo-code for waste-aware capacity planning algorithm in Hadoop.



(a) Default with 2 maps, 2 reduces

(b) Default with 4 maps, 2 reduces

(c) Default with 8 maps, 2 reduces

(d) Waste-aware capacity planner

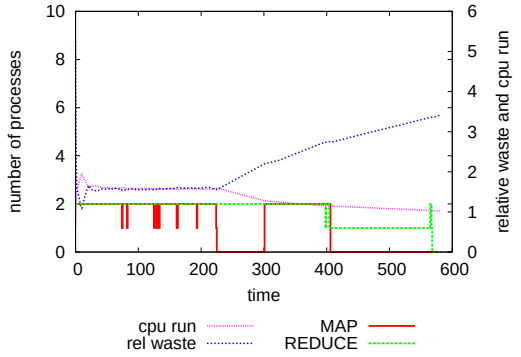
Figure 6.4: Waste of task execution using 2 x `grep` workload.

We now extend this example to a workload with two *different* jobs: a `grep` and `sort`; each job has roughly the same data-input size, but because of the nature of each job, the `grep` requires less time to complete (where time in this case is measured when there is no other resource demand). In Figure 6.5 we again see that the waste-aware scheduling algorithm initially runs with a 3 to 4 maps, but eventually settles down to fewer than 2. From this data and Table 6.2, we see this waste-aware algorithm also results in a less waste and increased throughput (makespan).

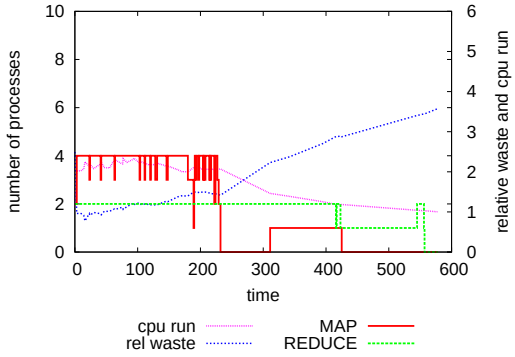
Interestingly, we see that each workload has a different "sweet spot"; in the `grep` workload, the Hadoop scheduler performs best with a configuration of 5 concurrent map tasks, whereas the workload with a `sort` performs best with fewer concurrent map tasks. The waste-aware scheduler simplifies the problem capacity planning to that of dynamic waste minimization.

Table 6.2: *Makespan* and *average resource waste* of default Hadoop configurations and waste-aware capacity planning using `grep` + `sort` workload.

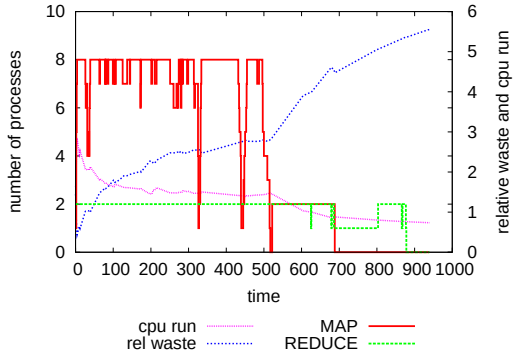
Scheduler	Maps	Reduces	Makespan	Average Waste
Default	1	2	655	400.347
Default	2	2	583	**428.403**
Default	3	2	622	467.357
Default	4	2	**579**	473.891
Default	5	2	637	545.125
Default	6	2	656	537.911
Default	7	2	990	775.603
Default	8	2	942	840.045
Default	9	2	1232	1264.23
Default	10	2	1362	1370.11
Waste-aware	–	–	**535**	**368.566**



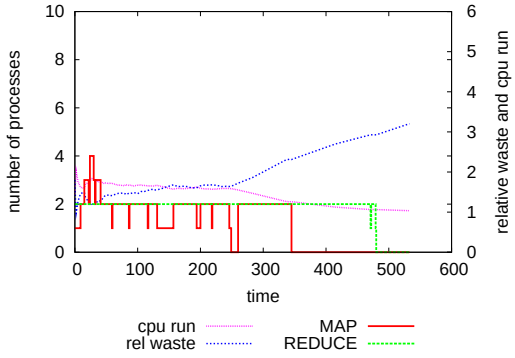
(a) Default with 2 maps, 2 reduces



(b) Default with 4 maps, 2 reduces



(c) Default with 8 maps, 2 reduces



(d) Waste-aware capacity planner

Figure 6.5: Waste of task execution using `grep` + `sort` workload.

Finally, we extend this example to a more complete workload, which involves 2 simultaneous instances of each of the three job types. In the previous example, we see that a setting of either 2 or 4 maps is best for the `grep` and `sort` workload. In Table 6.3 we list the results for these schedulers, and in Figure 6.6 we plot the behavior of each scheduler. Again, we see the waste-aware algorithm lowers waste and increases throughput.

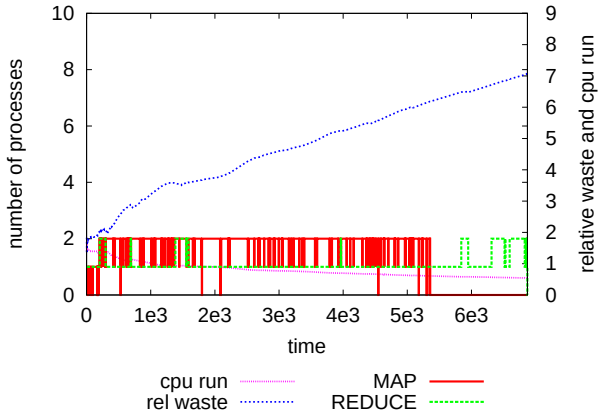
Table 6.3: *Makespan* and *average resource waste* of default Hadoop configurations and waste-aware capacity planning using 2 x `grep` + 2 x `wordcount` + 2 x`sort` workload.

Capacity Planning	Maps	Reduces	Makespan	Average Waste
Default	2	2	**6878**	**6724.93**
Default	4	2	7063	9613.76
Waste-aware	–	–	**4847**	**5484.92**

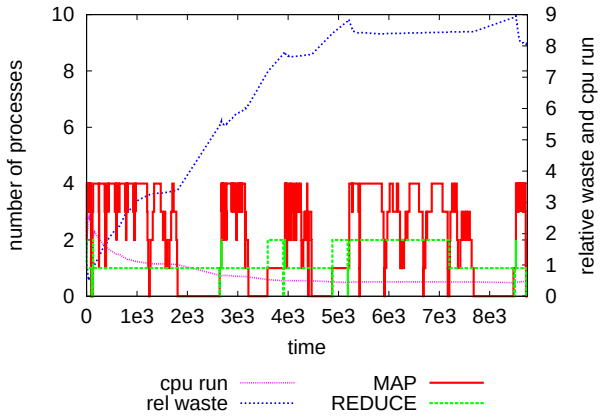
6.5 Task Resource Assignment

In the previous section, we discuss a simple framework for capacity planning based upon resource waste. This planning does not require knowledge about *how* tasks use resources, but merely relies on a heuristic that relates resource under-provisioning and over-provisioning to system throughput. This heuristic provides an effective way to maximize the throughput of a fixed resource capacity in various Hadoop workloads.

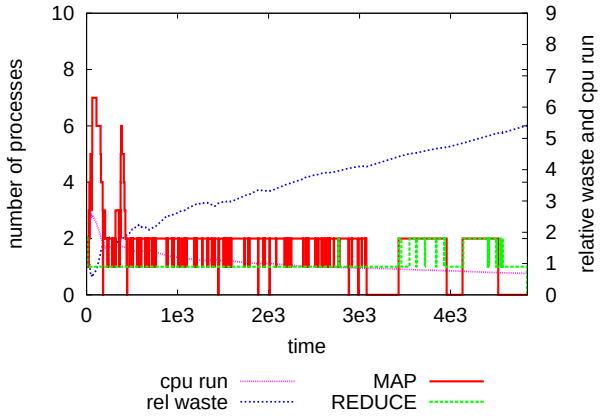
We hypothesize that using task-specific information can also improve allocation decisions for task execution in a way that is complementary to our capacity planning technique. This hypothesis is based upon the premise that applications exhibit different resource preferences, and that this information can be useful to a resource assignments or allocation. Our goal in this section is to show that jobs indeed exhibit different resource preferences, and that we can infer this information — without user input — to make more informed resource assignments for tasks. Note that this situation is similar to the problem in our market-based setting, where we place the onus on users to bid for resources on a particular host which are important to them. In this section, we turn this problem around, and assume that jobs are indifferent about which particular machine they use, and instead place tasks of jobs on nodes where their most desired (or bottleneck) resource is most plentiful.



(a) Default Hadoop scheduler, 2 maps, 2 reduces



(b) Default Hadoop scheduler, 4 maps, 2 reduces



(c) Waste-aware scheduler

Figure 6.6: Waste of task execution for 2 x `grep` + 2 x`wordcount` + 2 x`sort` workload.

6.5.1 Determining Job Resource Preferences

The first question we need to answer is whether or not real applications exhibit different preferences, or utility for resources? We simplify this question to identifying a job's bottleneck resource. We define a bottleneck resources as the one an application depends upon the most to complete execution. In order to identify this information, we measure the resource contention of each of three resources: CPU, I/O and memory for map and reduce tasks for each job we consider in the previous section. In order to measure this contention, we implement a profiling tool.

Profiler Implementation

We again use the `taskstats` interface on the Linux kernel to extract resource consumption and contention information for each task, as well as from the perspective of the node. The `taskstats` interface allows the central Hadoop scheduler to track process accounting information for any task or group of tasks. From this information, we can measure an application's time spent consuming and waiting for different resources. For each task, we create a data structure that periodically samples its resource consumption and waiting time for the CPU, disk I/O and the memory subsystem. For each job, we create a function that can aggregate these samples to generalize the resource profile of each task.

These profiling data structures are similar to the profiler we used in the capacity planning problem, but in the prior case, we are only concerned with aggregate consumption on the node, rather than per-task accounting information.

Measured Job Profiles

We begin by considering our three Hadoop jobs: *grep*, *sort*, and *wordcount*. Intuitively, a `grep` job is CPU-intensive and a `sort` job is relatively more disk I/O intensive since it read and writes more data, and a `wordcount` job exhibits the qualities of both jobs. In order to corroborate the intuition of these jobs, we profile the resource consumption of each of these jobs as load increases.

In Figure 6.7(a), 6.7(c), 6.7(e), we plot the resource consumption of the map tasks of each job as the number of concurrent *tasks* are run. In this graph, we define load (x-axis) as the number of map task slots on a single TaskTracker node. The stacked vertical bars represent the amount of time (in seconds) each task spends in a particular

state: either waiting for a resource (memory, I/O, CPU), or running on the CPU. The height of all bars on the y-axis, represents the average number of seconds accounted to each task for every second; these bars occasionally exceed 1 due to an implementation detail in our profiler. Specifically, each map and reduce task is itself comprised of several concurrent Java processes. Since a task itself can have one process running, and multiple processes waiting, the bar on the y-axis may exceed 1. However, we emphasize that it is more important to consider the bar heights as ratios, rather than as an absolute unit of measure.

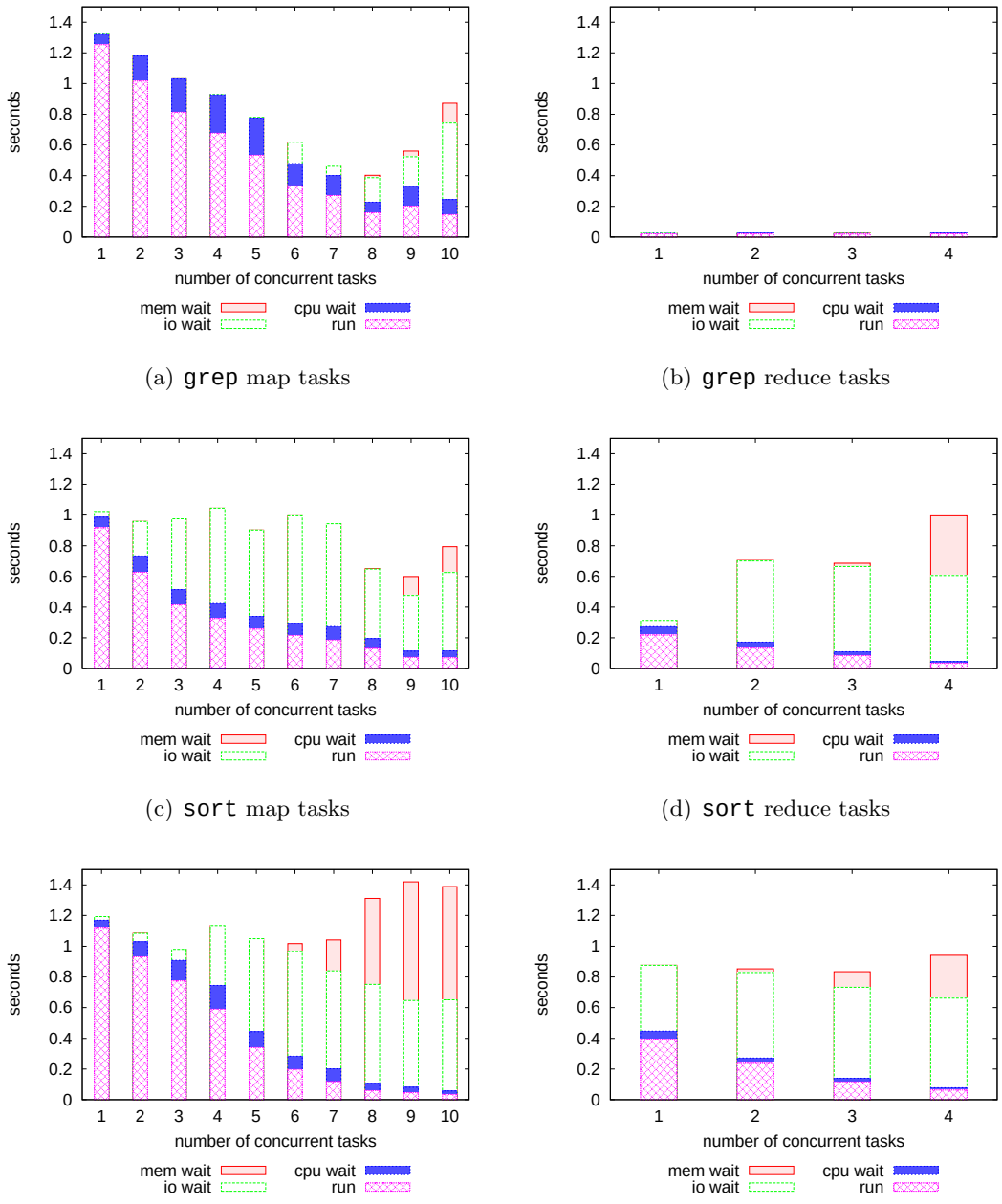
The goal of Figure 6.7 is to illustrate the bottleneck resource of a typical task in each job. As we can see, as the `grep` map load on the TaskTracker increases, each task spends more time waiting for the CPU, with relatively negligible time waiting for I/O or memory. On the other hand, the map tasks belonging to a `sort` job have a significant bottleneck of the I/O subsystem, which, as we mention, is due to its frequent interaction with the data stored in memory and disk.

Finally, the map tasks belonging to a `wordcount` job exhibits a similar bottleneck to the `grep` job for lighter loads, but as the number of concurrent tasks increases beyond 3, the I/O subsystem becomes a bottleneck, until finally, the memory subsystem becomes a primary bottleneck beyond a load of 7 concurrent tasks.

In Hadoop, the reduce tasks of a job are commonly more I/O intensive than its corresponding map tasks because they necessarily involve I/O. On the right column of Figure 6.7, we see that this intuition is correct for reduce tasks in all jobs relative to their map task counterparts, but interestingly, the reduce tasks for the grep require significantly fewer aggregate resources because it is simply shorter.

To relate these resource bottlenecks to the resource utilization on the TaskTracker, we plot *CPU* utilization on the TaskTracker for map tasks in Figure 6.8. We see that these graphs corroborate the high-level findings of which tasks bottleneck with the CPU, but more interestingly, the CPU is never fully saturated when another resource bottleneck exists. In other words, as long as resource contention exists on another resource (in this case, besides the CPU), the CPU will not be fully utilized. Similarly for reduce tasks (Figure 6.8), we see that the CPU remains idle for even longer periods of time.

Taken together, these graphs indicate that map tasks and reduce tasks of a typical Hadoop job can be time-shared effectively, but that is is also important to consider *which*



(a) `grep` map tasks

(b) `grep` reduce tasks

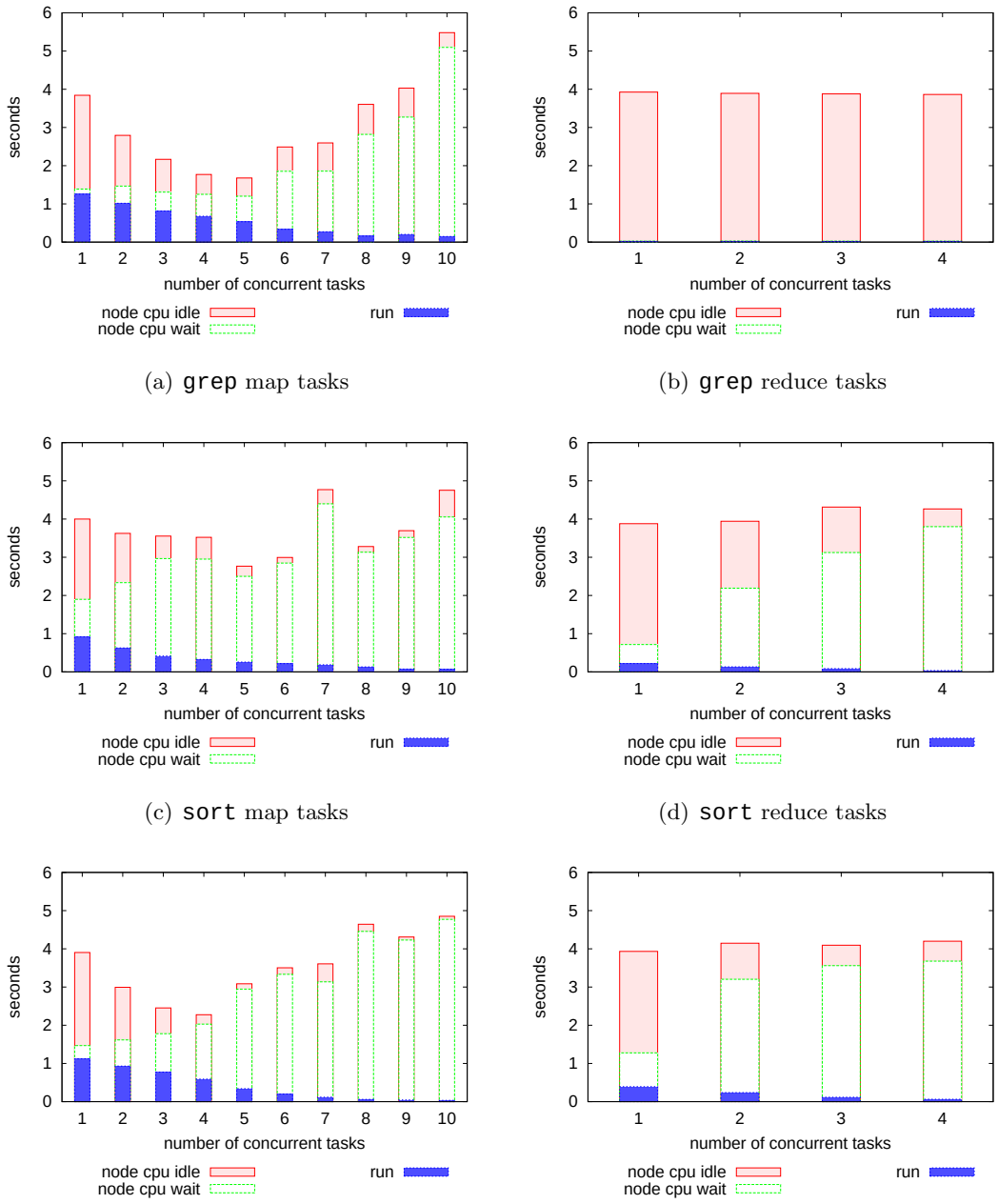
(c) `sort` map tasks

(d) `sort` reduce tasks

(e) `wordcount` map tasks

(f) `wordcount` reduce tasks

Figure 6.7: Resource consumption profiles of map and reduce tasks as a function of load.



(a) `grep` map tasks

(b) `grep` reduce tasks

(c) `sort` map tasks

(d) `sort` reduce tasks

(e) `wordcount` map tasks

(f) `wordcount` reduce tasks

Figure 6.8: Load on TaskTracker from map and reduce tasks for `grep`, `sort` and `wordcount` jobs.

Table 6.4: *Makespan* and *average waste* of waste-aware capacity planning and utility-aware task selection using 4 x `grep` + 4 x`sort` workload.

Capacity Planning	Task Assignment	Alice	Bob	Makespan	Avg Waste
Waste-aware	Default	722	773	773	2433.79
Waste-aware	Utility-aware	**617**	**604**	**617**	**2230.48**

job a map task or reduce task belongs to in executing tasks. Clearly, the map task of a *sort* may contend with the same resources as that of a typical reduce task, and load balancing of tasks should not merely happen at the level of task type. Rather, this partitioning should consider job type as well.

6.5.2 Utility-Aware Task Resource Assignment

We implement a *utility-Aware* task resource scheduler which assigns tasks to TaskTrackers based upon bottleneck resource availability. The algorithm for our scheduler makes several key, simplifying assumptions: (1) the map tasks for any particular job are similar, (2) the reduce tasks for any particular job are similar, and (3) each job has only one significant bottleneck resource. Based upon these assumptions, our scheduler tries at most k tasks with the same bottleneck resource on a machine at a given time. We currently tune k off-line as a proof-of-concept, but eventually hope to derive k in a similar fashion to the waste algorithm from Figure 6.3. We present pseudo-code for our utility-aware algorithm in Figure 6.9.

6.5.3 Experiments

In this section, we consider a workload with 4 `grep` jobs and 4 `sort` jobs. However, we also consider two users, who each have the same workload to run on the Hadoop cluster. Since our task-resource assignment problem is most interesting when there are multiple task trackers, we extend this experiment to run on 4 nodes, with 3 TaskTrackers.

In Table 6.4, we compare the performance of our waste-aware from the previous section, and the utility-aware algorithm in this section. We see that the that the utility-aware can further improve the utilization of resources.

```
assignTask(
 input:
   k             // scheduler parameter
   TaskTracker   // contains state of TaskTracker we are
                 // considering assigning a task to
   TaskProfiles  // data structure for each ready tasks
                 // ready to run
   WasteRecords  // hash table mapping <#maps,#reds>
                 // to data structures recording waste
 output:
   task          // possibly null (for none) task to send
                 // to TaskTracker for execution

  numMemTasks = TaskTracker.numMemTasks;
  numIOTasks  = TaskTracker.numIOTasks;
  numCPUTasks = TaskTracker.numCPUTasks;
  // make list of number of tasks occupying bottleneck
  // resources in ascending order
  sortedBottlenecks = makeSortedList((numMemTasks,MEM),
                                     (numIOTasks,IO),
                                     (numCPUTasks,CPU));
  // extract the numTasks field to see how many
  minBottleneckNum = sortedBottlenecks[0][0];
  // no more than k of the same type of task
  if minBottleneckNum >= k:
    return null;
  else:
    for numTasks, Type in sortedBottlenecks:
      if TaskProfile.hasTask(mybottleneck=Type):
        return TaskProfile.getTask(mybottleneck=Type)
    // if we haven't found a suitable task, return none
    return null;
)
```

Figure 6.9: Pseudo-code for utility-aware task resource allocation algorithm.

6.6 Limitations

Before our scheduler implementation can be used in a production environment, there are a few key implementation limitations to overcome. First, the profiler as currently constructed, introduces overhead to the scheduling system in Hadoop. Secondly, it is not clear how quickly the capacity-planning algorithm or task-assignment algorithm can adapt to larger workloads which may exhibit significantly different characteristics over time.

Furthermore, we have not compared our implementation to a scenario where application resource preferences are known a priori. Since the resource consumption behavior of an application depends upon the consumption characteristics of co-located tasks, we would need to measure every combination of task assignments in order to conduct this comparison. We defer this study to future work.

6.7 Conclusions

In this chapter, we revisit two classic problems in time-shared scheduling: capacity planning and task resource allocation. We frame the capacity planning problem as a waste-minimization problem, and the task-allocation problem as a bottleneck-packing problem in the context of a Hadoop scheduling system. In effect, we define a utility function for each Hadoop TaskTracker machine (maximizing utility = minimizing waste), and a utility function for each task (place tasks on a machine where the bottleneck resource is plentiful), and use this information to drive the decisions in each system. Overall, we implement a system that can infer utility information based upon a few simple rules to demonstrably improve the throughput of real applications. While this implementation does not address the problem of having complete resource control from our Bellagio deployment in PlanetLab, it does provide a framework with which a system like Bellagio can be deployed on a system like PlanetLab with less usage costs imposed upon users.

Chapter 7

Concluding Remarks

In this chapter, we summarize our primary results and discuss a few specific avenues of future research in this space.

7.1 Summary

This dissertation explores the potential of using market-based resource allocation in real, large scale computing systems. We are inspired by the tremendous literature in designing such economically-inspired systems, and offer our implementation and deployment experience from a pragmatic, computer science, systems approach.

Our implementation experience with Bellagio and Mirage illustrates that a practical market-based allocation system itself requires a trade-off between theoretically-desirable pricing and allocation mechanisms, and the tractable heuristics that must be used to implement a scheduling mechanism in large-scale systems. Nonetheless, our Mirage deployment experience demonstrates that a market framework can indeed lead to improved and autonomously-driven allocation decisions based upon user-provided job utility information, when user behavior is properly constrained. However, as we observe in Mirage, the use of heuristic allocation algorithms and virtual currency can lead to undesirable user behavior and render the system vulnerable to strategic manipulation. Although the heuristic allocation algorithms have been adjusted to handle the currently observed behaviors, further analysis is required to understand if the system is vulnerable to other forms of manipulation.

Still, these deployments alone do not necessarily address many of the qualitative

apprehensions many in the computer science community may have toward such market-based systems.

First, a market-based allocation system requires more complex user interaction, and thus is often viewed as too fragile. We explore the impact that user uncertainty and error-prone behavior have on the efficiency of our system. Perhaps unsurprisingly, we find that a market-based system is, indeed, more *sensitive* to these types of errors than traditional scheduling policies, but based upon our forecast of user information fidelity, such systems can still provide a 20%–100% increase in user satisfaction. However, as we mention later, this analysis does not consider the sensitivity of our market to strategic manipulation.

Second, a market-based allocation policy is often qualitatively viewed as less fair than traditional, best-effort allocation policies. We demonstrate that in particular circumstances, this fear *is* grounded, and that users with a significantly larger share of wealth can indeed receive a larger share of resources. However, based upon data of relevant environments, we find that demand characteristics and our proposed (virtual) currency policy leads to increased fairness of allocations. One way to view this result is that the increase in *overall* allocation efficiency allows more room for distribution of utility across users, such that the expected utility received by the average, best and worst-case users are in fact higher than when using traditional allocation policies.

Despite these generally positive results, our deployment experience with a small set of Mirage users, and the relatively short-lived experience with Bellagio also reveal specific challenges to further widespread deployments.

Specifically, we investigate the implementation of a market-based job-execution service and we find that existing market designs are not expressive enough for many real-world clients. For these clients, such a service has little incentive to execute the low-paying portion of their workflows. Adding higher-level constructs, like contracts and aggregate utility functions can provide the incentives to satisfy both a provider's goal for profitability, and the clients' aggregate needs, while having the net effect of increasing the utility of both parties.

A common criticism against market-based allocation mechanism — particularly in a time-sharing system (like Bellagio) — is that they are too burdensome to use: a provider must plan capacity, and users must understand their application's resource preferences. We design and implement support in a Hadoop scheduler that reduces

each of these burdens by automatically inferring this information from an underlying profiler, thereby increasing the efficiency of allocations for real applications, without additional administrator or user support. We are optimistic that this type of system has the potential to apply more generally in other time-sharing systems.

In general, we find that significant additional engineering effort is required in some domains to identify and address the conflicting needs of users and providers, and to make a market mechanism easy enough to use in a production setting. While market-based systems have not yet seen widespread adoption, we believe that our efforts demonstrate that such market-based techniques can improve user satisfaction by improving the quality of resource allocations in real systems.

7.2 Future Directions

Based upon our experience with real users and workloads, we also believe that we can begin targeting practical market-based resource allocation mechanisms more generally to emerging technologies such as cloud-based infrastructures, data centers, and extreme-scale testbeds. Using a market-based system in these contexts enables several broader impacts.

First, such a framework addresses the challenge of establishing an inter-domain allocation policy for the intelligent consolidation of resource infrastructures across different administrative domains [48], thereby decreasing significant and possibly redundant capital investments. Secondly, the fine-grained accounting of resource usage costs and "profitability" can motivate more balanced [73], cost-effective, or energy-efficient designs, thereby impacting larger, social concerns, like the energy footprint of such systems [3]. Finally, such a framework may motivate increased use of volunteer-based computing systems [6, 19, 112]; such systems have had tremendous success in solving large-scale problems using otherwise "idle" resources [48, 112], but have yet to tap the potential of resources belonging to non-participating institutions who currently do not have the incentive to provision their idle resources. These efforts may, in turn, broaden the reach and accessibility of computer power across geographic and socio-economic boundaries. We outline a few specific steps required to take before market-based computing design can fulfill this vision.

7.2.1 Characterizing Strategic Behavior

In Chapter 4, we characterize the sensitivity of a market-based design to traditional problems of user error and inexact system parameter settings. By virtue of its user-driven design, a market creates opportunities for a new class strategic or opportunistic manipulation [79]. In Chapter 3, we discuss evidence of real users manipulating allocation decisions of the initial open-auction allocation in Mirage. While the change to a sealed-bid auction may have ostensibly addressed those particular manipulations, it is not clear how resilient such a framework is in larger or more anonymous settings.

There are efforts in computer science research to design a practical incentive-compatible mechanism in this context [78], but there is currently a lack of a simultaneously efficient, incentive-compatible, and individually rational mechanism in economics theory for our particular setting, which leaves this challenge as an open theoretical problem. However, careful implementation of simulations based upon evolutionary game theory [71] or from empirical studies of real systems can help identify potentially opportunistic strategies. By focusing the analysis on a particular deployment domain, it may be possible to design external system "rules" to help assuage initial apprehensions about the possibility of strategic manipulation by explicitly preventing any such harmful behaviors.

7.2.2 Deployment-Driven Modeling

Our experience with Mirage gives us insight into designing an effective market-based allocation system. This insight is based largely upon user utility information, user workload information, and observed user interaction with market mechanisms. Absent this data, characterizing the expected behavior of such a system in another problem domain is a very difficult problem [74, 101]. Gathering this information in emergent domains such as cloud-based or petascale systems is of increasing importance as applications increase in size and diversity. A critical complement to any pragmatic research agenda is the deployment of experimental testbeds. As opposed to the live deployments of Bellagio and Mirage, the focus of these proposed platforms will be to collect data from real users, and use this data to refine the underlying allocation mechanisms based upon user feedback and empirical analysis.

An added benefit of this type of platform is that it provides a medium for economics researchers to test and refine various allocation mechanisms. We anticipate the

effectiveness of any particular mechanism to be largely dependent upon the particular user types, preferences and job characteristics in each problem domain. Previous approaches to similar problems rely either on pure simulation-based studies [25, 91], abstract game-theoretic models [5, 95], or a combination of both [4, 92]. The alternatives to both the allocation and pricing mechanism used in these systems have not been investigated in the context of real users and real workloads. This platform would also provide a unique vantage point to consider alternative metrics[1], which may be better suited for optimizing over conflicting interests, say for example, balancing profitability and power consumption in a data center, or for balancing load distribution and response time in a network backbone. With a focus on implementation and deployment, this approach can complement theoretical research and promote better design *and implementation* through modeling and measurement.

7.2.3 System Convergence and Worst-Case Performance

We demonstrate how markets can improve the overall value derived from a system in the expected case. Our efforts parallel existing theoretical analysis in this respect, as the prior work in the area also focuses on the characterization of steady-state performance, or equilibrium outcome(s) of these systems. However, the theory that describes these equilibria (if they even exist) says nothing about *how* or even *if* a system can arrive at such an outcome.

We believe that it is important for systems builders to design against potential instability and *worst-case scenarios*, since production systems may be unable to tolerate performance loss and instability from a potentially fluctuating market. For example, emerging petascale clusters engender applications with resource demand that varies significantly across both time (minutes versus months) and space (tens of processors versus tens of thousands). With extreme variability across individual demand, it is unclear whether relying on the existence of equilibrium prices alone is enough to measure the applicability of a mechanism to support these varied applications, since, for example, rapid price *convergence* is also critical to those jobs that operate on short timescales, like minutes. Therefore, we argue that it is important to consider both the convergence and worst-case properties of applying an economics-inspired mechanism to a system. Since these properties likely depend upon domain-specific information like application

[1] As noted before, most applied concepts of user satisfaction involve a utilitarian social welfare function. However, it would be interesting to try alternatives, such as Maximin [75].

demand patterns, an experimental infrastructure such as the one we describe above would be a critical to provide empirical characterization of these domains. In turn, this characterization will allow us to develop or apply the appropriate theory to characterize the worst-case and convergence properties of an economics-inspired design for real systems in these domains.

7.2.4 Automating User Decisions

In many computational systems, user behavior largely governs how quickly, or whether a system can converge to equilibrium. Observation of real users indicates that the primary obstacle to convergence is the fact many users do not behave "rationally", or the way that economic theory predicts that they will play. Information asymmetry [46, 57], lack of sophistication [34], and computational tractability [63] are examples of typical barriers to this rational behavior. We suggest an approach to use the model of economic rationality to *guide* user behavior, rather than as a tool to *predict* how they will behave. There are many instances where we may be able to determine what is best for the user, and help the user refine this information with otherwise minimal burden. We begin investigation of this approach in the multi-user, time-shared computing framework of Chapter 6. The premise of the study is that different jobs have a preference over different computational resources: jobs that are CPU-bound may prefer less disk or network bandwidth than other jobs. We can use existing operating systems profiling techniques to formulate this preference as utility for different resources, and schedule jobs based upon this utility information, all with limited or no interaction with the user. The next interesting question will be to compare the trade-offs between inferring this information and having users provide it.

7.3 Final Thoughts

The intersection between economics and computer science research began almost a decade ago. From the computer science perspective, there have been numerous results in the area of theory, artificial intelligence, and to a more limited extent, system design and networks. However, there have not been as many corresponding experiences in building and deploying systems that leverage these designs. Our work has demonstrated that this approach can deliver significantly more value to all stakeholders in these systems

despite observed inconsistencies in user behavior, uncertainty, or otherwise imperfect operating conditions. We hope to use our experience to initiate a sustained research agenda to push forth the iterated design and deployment of these systems, and introspection of these deployments in order to understand how these systems can best be designed to meet our emerging needs.

Bibliography

[1] San Diego Supercomputing Center (SDSC). `http://www.sdsc.edu`.

[2] TeraGrid. `http://teragrid.org`.

[3] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *Proc. 6th USENIX/ACM Symp. Networked Systems Design & Implementation (NSDI 09)*, April 2009.

[4] A. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J. Martin, and C. Porth. BAR Tolerance for Cooperative Services. In *Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[5] A. Akella, R. Karp, C. H. Papadimitrou, S. Seshan, and S. Shenker. Selfish behavior and stability of the internet: A game-theoretic analysis of TCP. In *Proc. of SIGCOMM*, Aug 2002.

[6] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proc. IEEE/ACM Workshop on Grid Computing*, November 2004.

[7] E. Anderson, D. Beyer, K. Chaudhuri, T. Kelly, N. Salazar, C. Santos, R. Swaminathan, R. Tarjan, J. Wiener, and Y. Zhou. Value-maximizing deadline scheduling and its application to animation rendering. In *Proc. 17th ACM Symp. on Parallelism in Algorithms and Architectures*, July 2005.

[8] A. AuYoung, L. E. Grit, J. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *Proc. 15th IEEE Int'l Symp. High Performance Distributed Computing*, June 2006.

[9] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of 1st Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[10] A. Baldwin, Y. Beres, D. Plaquin, and S. Shiu. Trust record: high-level assurance and compliance. In *Proceedings of iTrust 2005*, volume 3477 of *Lecture Notes in Computer Science*, pages 393–6. Springer Verlag, May 2005.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[12] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the Competitiveness of On-line Real-time Task Scheduling. *Real-Time Systems*, 4(2):125–144, 1992.

[13] J. Beckett. HP Labs goes Hollywood: researchers help bring "Shrek 2" to life, April 2004. Web page.

[14] J. Benhabib and A. Bisin. The distribution of wealth and redistributive policies. Meeting Papers 368, Soc. for Economic Dynamics, Dec. 2006.

[15] A. C. Benjamim, J. Sauvé, W. Cirne, and M. Carelli. Independently auditing service level agreements in the grid. In *Proceedings of 11th HP OpenView University Association Workshop (HPOVUA 2004)*, June 2004.

[16] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware. `http://hadoop.apache.org.`, 2005.

[17] C. Boutilier, T. Sandholm, and R. Shields. Eliciting Bid Taker Non-price Preferences in (Combinatorial) Auctions. In *Proc. 19th Nat'l Conf. Artificial Intelligence (AAAI 04)*, pages 204–211, Cambridge, Massachusetts, USA, July 2004. The MIT Press.

[18] A. Byde, M. Sallé, and C. Bartolini. Market-Based Resource Allocation for Utility Data Centers. Technical Report HPL-2003-188, Hewlett Packard Laboratories, September 2003.

[19] B. Calder, A. A. Chien, J. Wang, and D. Yang. The Entropia Virtual Machine for Desktop Grids. In *2005 ACM/Usenix Int'l Conference on Virtual Execution environments*, June 2005.

[20] J. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proc. 18th ACM Symp. Operating Systems Principles*, October 2001.

[21] K. Chen and P. Muhlethaler. A Scheduling Algorithm for Tasks Described by Time Value Function. *Real-Time Systems*, 10(3):293–312, May 1996.

[22] S.-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon. The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In *Proc. 8th Workshop Job Scheduling Strategies for Parallel Processing*, July 2002.

[23] N. Christin, J. Grossklags, and J. Chuang. Near Rationality and Competitive Equilibria in Networked Systems. In *Proc. ACM SIGCOMM workshop on Practice and Theory of Incentives in Networked Systems (PINS 04)*, September 2004.

[24] B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: A Microeconomic Resource Allocation System for SensorNet Testbeds. In *Proc. IEEE Workshop Embedded Networked Sensors*, May 2005.

[25] B. N. Chun and D. E. Culler. User-centric Performance Analysis of Market-based Cluster Batch Schedulers. In *Proc. 2nd IEEE Int'l Symp. Cluster Computing and the Grid*, May 2002.

[26] B. N. Chun and A. Vahdat. Workload and Failure Characterization on a Large-Scale Federated Testbed. Technical Report IRB-TR-03-040, Intel Research Berkeley, November 2003.

[27] W. Cirne and F. Berman. A comprehensive model of the supercomputing workload. In *Proc. 4th IEEE Workshop Workload Characterization (WWC 01)*, December 2001.

[28] R. K. Clark, E. D. Jensen, and F. D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *Winter USENIX Technical Conference*, April 1993.

[29] Cloudscaling. Amazon's EC2 Generating 220M+ Annually, October 2009. `http://cloudscaling.com/blog/cloud-computing/amazons-ec2-generating-220m-annually.`

[30] K. G. Coffman and A. M. Odlyzko. Internet growth: is there a "Moore's law" for data traffic? pages 47–93, January 2002.

[31] V. Conitzer and T. Sandholm. Complexity of Mechanism Design. In *Proc. 18th Conf. Uncertainty in Artificial Intelligence Conf. (UAI 02)*, pages 103–110, San Francisco, California, USA, August 2002. Morgan Kaufmann Publishers Inc.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition edition, 2001.

[33] D. Cournapeau. em, a python package for Gaussian mixture models. `http://www.ar.media.kyoto-u.ac.jp/members/david/softwares/em/.`

[34] V. P. Crawford and N. Iriberri. Level-k auctions: Can a nonequilibrium model of strategic thinking explain the winner's curse and overbidding in private-value auctions? *Econometrica*, 75(6):1721–1770, 2007.

[35] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke. SNAP: a protocol for negotiating service level agreements and coordinating resource management in distributed systems. *Lecture Notes in Computer Science*, 2537:153–83, January 2002.

[36] A. Dan, C. Dumitrescu, and M. Ripeanu. Connecting client objectives with resource capabilities: an essential component for grid service managent infrastructures. In *Proceedings of 2nd International Conference on Service Oriented Computing (ICSOC'04)*, November 2004.

[37] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th USENIX/ACM Symp. Operating Systems Design & Implementation (OSDI 04)*, December 2004.

[38] T. Duffield, D. Hart, and N. Wolter. Data obtained through personal communication. San Diego Supercomputing Center, 2009.

[39] C. L. Dumitrescu and I. Foster. Gruber: a grid resource usage sla broker. *Lecture Notes in Computer Science*, 3648:465–74, August 2005.

[40] A. Elfatatry and P. Layzell. A negotiation description language. *Software—Practice and Experience*, 35(4):323–43, April 2005.

[41] D. Feitelson. Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/.`

[42] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel Job Scheduling – A Status Report. In *Proc. 10th Workshop Job Scheduling Strategies for Parallel Processing*, June 2004.

[43] D. G. Feitelson. Experimental Analysis of the Root Causes of Performance Evaluation Results: A Backfilling Case Study. *IEEE Trans. Parallel and Distributed Systems*, 16(2):175–182, Feb. 2005.

[44] Flux Research Group, Univ. of Utah, Dept. Computer Science. EmuLab. `http://www.emulab.net.`

[45] I. Foster and C. Kesselman, editors. *The Grid 2: blueprint for a new computing infrastructure*. Morgan Kaufmann, 2nd edition, 2003.

[46] E. Friedman, M. Shor, S. Shenker, and B. Sopher. An experiment on learning with limited information: nonconvergence, experimentation cascades, and the advantage of being slow. *Games and Economic Behavior*, 47(2):325 – 352, 2004.

[47] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. Sharp: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 133–148. ACM Press, 2003.

[48] J. Gray. Distributed computing economics. Technical Report MSR-TR-2003-24, Microsoft Research, March 2003.

[49] D. Grosu. AGORA: an architecture for strategyproof computing in grids. In *Proc. 3rd Int'l Workshop Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, July 2004.

[50] A. Gupta, D. O. Stahl, and A. B. Whinston. The economics of network management. *Comm. ACM*, 42(9):57–63, 1999.

[51] G. Heiser, F. Lam, and S. Russell. Resource management in the Mungi single-address-space operating system. In *Proc. 21st Australian Computer Sci. Conf.*, February 1998.

[52] B. Hudson and T. Sandholm. Effectiveness of query types and policies for preference elicitation in combinatorial auctions. In *Proc. 3rd Int'l Joint Conf. Autonomous Agents and Multiagent Systems (AAMAS 04)*, pages 386–393, August 2004.

[53] D. E. Irwin, J. Chase, L. E. Grit, and A. Yumerefendi. Self-Recharging Virtual Currency. In *Proc. 3rd Workshop Economics of Peer-to-Peer Systems*, August 2005.

[54] D. E. Irwin, L. E. Grit, and J. Chase. Balancing Risk and Reward in a Market-Based Task Service. In *Proc. 13th IEEE Int'l Symp. High Performance Distributed Computing*, pages 160–169, June 2004.

[55] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, March 2007.

[56] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *Proc. 7th Workshop Job Scheduling Strategies for Parallel Processing*, June 2001.

[57] M. Kearns, S. Suri, and N. Montfort. An experimental study of the coloring problem on human subject networks. *Science*, 313(5788):824–827, August 2006.

[58] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: shadow prices, proportional fairness and stability. In *J. Operational Research Soc.*, volume 49, 1998.

[59] T. Kelly. Utility-directed allocation. In *Proc. 1st Workshop Algorithms, Architectures for Self-Managing Systems*, July 2003.

[60] L. Kleinrock. *Queuting Systems, Vol. 2: Computer Applications*. New York, New York, USA, 1976.

[61] V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Proceedings of 2nd International Conference on Autonomic Computing (ICAC'05)*, June 2005.

[62] K. Lai, B. A. Huberman, and L. Fine. Tycoon: A Distributed Market-based Resource Allocation System. Technical Report arXiv:cs.DC/0404013, Hewlett Packard Laboratories, April 2004.

[63] K. Larson and T. Sandholm. Costly Valuation Computation in Auctions. In *Proc. 8th Conf. Theoretical Aspects of Rationality and Knowledge*, pages 169–182, July 2001.

[64] H. C. Lau and M. K. Lim. Multi-Period Multi-Dimensional Knapsack Problem and Its Application to Available-to-Promise. In *Proc. 2nd Int'l Symp. on Scheduling (ISS)*, May 2004.

[65] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource QoS problem. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, December 1999.

[66] C. B. Lee and A. Snavely. On the User-Scheduler Dialogue: Studies of User-Provided Runtime Estimates and Utility Functions. *Int'l J. High Performance Computing Applications*, 20(4):495–506, 2006.

[67] C. B. Lee and A. Snavely. Precise and Realistic Utility Functions for User-centric Performance Analysis of Schedulers. In *Proc. 16th IEEE Int'l Symp. High Performance Distributed Computing*, June 2007.

[68] D. Lehmann, R. Müeller, and T. Sandholm. *The Winner Determination Problem*, chapter 12. MIT Press, Cambridge, Massachusetts, USA, 2006.

[69] L. Levy, L. Blumrosen, and N. Nisan. On Line Markets for Distributed Object Services: the MAJIC System . In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[70] D. A. Lifka. The ANL/IBM SP Scheduling System. In *Proc. 1sd Workshop Job Scheduling Strategies for Parallel Processing*, Apr. 1995.

[71] Z. S. Ma. Towards an Extended Evolutionary Game Theory with Survival Analysis and Agreement Algorithms for Modeling Uncertainty, Vulnerability, and Deception. In *Proc. Int'l Conf. on Artificial Intelligence and Computational Intelligence (AICI 09)*, 2009.

[72] J. K. MacKie-Mason and H. R. Varian. Pricing congestible network resources. *IEEE Journal on Selected Areas in Communications*, 13(7):1141–1149, 1995.

[73] H. Madhyastha, J. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, and A. Vahdat. A model for storage and compute requirements of cluster applications. In preparation, 2010.

[74] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Experiences applying game theory to system design. In *Proceedings of the ACM SIGCOMM workshop on Practice and theory of incentives in networked systems*, New York, NY, USA, 2004. ACM.

[75] A. Mas-Colell, M. D. Whinston, and J. R. Green. *Microeconomic Theory*. Oxford Univ. Press, New York, New York, USA, 1995.

[76] A. W. Muálem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Trans. Parallel and Distributed Systems*, 12(6):529–543, June 2001.

[77] National Science Foundation. GENI: Exploring Networks of the Future. `http://geni.net`.

[78] C. Ng, P. Buonadonna, B. N. Chun, A. C. Snoeren, and A. Vahdat. Addressing Strategic Behavior in a Deployed Microeconomic Resource Allocator. In *Proc. 3rd Workshop Economics of Peer-to-Peer Systems*, August 2005.

[79] C. Ng, D. Parkes, and M. Seltzer. Strategyproof Computing: Systems Infrastructures for Self-Interested Parties. In *Proc. 1st Workshop Economics of Peer-to-Peer Systems*, June 2003.

[80] N. Nisan and A. Ronen. Algorithmic Mechanism Design (extended abstract). In *Proc. 31st ACM Symp. Theory of Computing (STOC 99)*, pages 129–140, New York, New York, USA, May 1999. ACM.

[81] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs for Wide-Area Resource Discovery. In *Proceedings of the Fourteenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[82] D. Oppenheimer, V. Vatkovskiy, and D. A. Patterson. Towards a framework for automated robustness evaluation of distributed services. In *Proc. 2nd Bertinoro Workshop on Future Directions in Distributed Computing*, June 2004.

[83] D. C. Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency.* PhD thesis, University of Pennsylvania, May 2001.

[84] D. C. Parkes, R. Cavallo, N. Elprin, A. Juda, S. Lahaie, B. Lubin, L. Michael, J. Shneidman, and H. Sultan. ICE: An iterative combinatorial exchange. In *Proc. 6th ACM Conf. Electronic Commerce (EC 05)*, pages 249–258, June 2005.

[85] D. C. Parkes, L. H. Ungar, and D. P. Foster. Accounting for Cognitive Costs in On-Line Auction Design. *Lecture Notes in Computer Science*, 1571:25–40, 1998.

[86] P. Patrick. Impact of soa on enterprise information architectures. In *Proceedings of 2005 ACM SIGMOD International Conference on the Management of Data*, June 2005.

[87] L. Peterson, V. Pai, N. Spring, and A. Bavier. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. Technical Report PDN–05–028, PlanetLab Consortium, June 2005.

[88] D. Petrou, G. R. Ganger, and G. A. Gibson. Cluster scheduling for explicitly speculative tasks. In *Proceedings of International Conference on Supercomputing (ICS'04)*, June–July 2004.

[89] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Springer Publishing Company, Incorporated, 2008.

[90] PlanetLab Consortium, Princeton Univ., Dept. Computer Science. PlanetLab. `http://planet-lab.org`.

[91] F. I. Popovici and J. Wilkes. Profitable Services in an Uncertain World. In *Proc. ACM/IEEE Conf. Supercomputing (SC 05)*, November 2005.

[92] L. Qiu, Y. R. Yang, Y. Zhang, and S. Shenker. On Selfish Routing in Internet-like Environments. In *Proc. ACM SIGCOMM '03 Conf. Communications Architectures and Protocols*, pages 151–162, New York, NY, USA, August 2003. ACM Press.

[93] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. Toward loosely coupled programming on petascale systems. In *Proc. ACM/IEEE Conf. Supercomputing (SC 08)*, November 2008.

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

[95] T. Roughgarden and E. Tardos. How bad is selfish routing? In *Foundations of Computer Science, Annual IEEE Symposium on*, volume 0, page 93, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[96] San Diego Supercomputing Center User Services Consulting. Managing your account. `http://www.sdsc.edu/us/consulting/managing.html.`

[97] T. Sandholm. Issues in Computational Vickrey Auctions. *Int'l J. Electronic Commerce*, 4(3):107–129, 2000.

[98] T. Sandholm and K. Lai. Mapreduce optimization using dynamic regulated prioritization. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems*, June 2009.

[99] R. R. Schaller. Moore's law: past, present, and future. *IEEE Spectrum*, 34(6):52–59, June 1997.

[100] S. Shenker, D. Clark, D. Estrin, and S. Herzog. Pricing in computer networks: Reshaping the research agenda. In *Proc. TPRC*, 1995.

[101] J. Shneidman, C. Ng, D. Parkes, A. AuYoung, A. C. Snoeren, A. Vahdat, and B. N. Chun. Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems. In *Proc. 10th USENIX Workshop Hot Topics in Operating Systems*, June 2005.

[102] A. Snavely and L. Carter. Symbiotic jobscheduling on the tera mta. In *In Proceedings of Third Workshop on Multi-Threaded Execution, Architecture, and Compilers*, 2000.

[103] I. Stoica, H. Abdel-Wahab, and A. Pothen. A microeconomic scheduler for parallel computers. In *Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing*, April 1995.

[104] I. E. Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6):449–451, 1968.

[105] D. Talby and D. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In *Proc. 13th Int'l Parallel and Distributed Processing Symp.*, Apr. 1999.

[106] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling User Runtime Estimates. In *Proc. 11th Workshop Job Scheduling Strategies for Parallel Processing*, June 2005.

[107] D. Tsafrir and D. G. Feitelson. The Dynamics of Backfilling: Solving the Mystery of Why Increased Inaccuracy May Help. In *2006 IEEE Int'l Symp. Workload Characterization*, October 2006.

[108] W. Vickrey. Counterspeculation, Auctions, and Competitive Sealed Tenders. 16(1):8–37, 1961.

[109] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kehpart, and S. Stornetta. Spawn: a Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–17, February 1992.

[110] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility Functions in Autonomic Systems. In *Proc. 1st Int'l Conf. Autonomic Computing*, May 2004.

[111] J. Weinberg and A. Snavely. User-Guided Symbiotic Space-Sharing of Real Workloads. In *The 20th ACM International Conference on Supercomputing (ICS'06)*, June 2006.

[112] D. Werthimer, J. Cobb, M. Lebofsky, D. Anderson, and E. Korpela. SETI@HOME — massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, 2001.

[113] A. Wierman and M. Harchol-Balter. Classifying Scheduling Policies with Respect to Unfairness in an M/GI/1. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems*, June 2003.

[114] A. Wierman and M. Harchol-Balter. Classifying Scheduling Policies with Respect to Higher Moments of Conditional Response Time. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems*, June 2005.

[115] A. Wierman and M. Nuyens. Scheduling despite inexact job-size information. In *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems*, June 2008.

[116] R. Wolski, J. S. Plank, T. Bryan, and J. Brevik. G-commerce: Market formulations controlling resource allocation on the computational grid. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, March 2001.

[117] C. S. Yeo and R. Buyya. A Taxonomy of Market-based Resource Management Systems for Utility-driven Cluster Computing. *Software: Practice and Experience*, 36(13):1381–1419, Nov. 2006.

[118] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[119] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. 10th USENIX/ACM Symp. Operating Systems Design & Implementation (OSDI 08)*, December 2008.

[120] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: internet path failure monitoring and characterization in wide-area services. In *Proc. 6th USENIX/ACM Symp. Operating Systems Design & Implementation (OSDI 04)*, 2004.

[121] Q. Zhu and G. Agrawal. Supporting Fault-Tolerance for Time-Critical Events in Distributed Environments. In *Proc. ACM/IEEE Conf. Supercomputing (SC 09)*, November 2009.