UNIVERSITY OF CALIFORNIA, SAN DIEGO

On the User-Scheduler Relationship in High-Performance Computing

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

in

Computer Science

by

Cynthia Bailey Lee

Committee in charge:

Professor Allan Snavely, Chair
Professor Amin Vadhat, Co-Chair
Professor Scott Baden
Professor Roger Bohn
Professor Vincent Crawford

2009

The dissertation of Cynthia Bailey Lee is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____
Co-Chair

_____
Chair

University of California, San Diego

2009

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Allan, Jusok, Rachelle, Jonas, Mom and Dad, this would never have been possible without your patience, guidance and support.

A debt is owed to the committee members, whose feedback and suggestions greatly improved this dissertation. I also wish to thank Jon Weinberg, Mike McCracken, Alvin AuYoung, Beth Simon, Julie Conner, Paul Kube, Jeanne Ferrante, Larry Carter, Dan Tsafrir and Kenneth Yoshimoto.

Parts of Chapter 5 are reprints of the material as it appears in the Proceedings of the 10th Job Scheduling Strategies for Parallel Processing, 2004. Lee, Cynthia B.; Schwartzman, Yael; Hardy, Jennifer; Snavely, Allan, 2004. [46] The dissertation author was the primary investigator and author of this paper.

Parts of Chapter 1 are reprints of the material as it appears in the dissertation author's research exam (Comprehensive Exam) for the Department of Computer Science and Engineering, University of California, San Diego, 2005. [45] The dissertation author was the sole author of this paper.

Parts of Chapters 2 and 5 are reprints of the material as it appears in International Journal of High Performance Computing Applications, 2006. Lee, Cynthia B.; Snavely, Allan E., 2006. [47] The dissertation author was the primary investigator and author of this paper.

Parts of Chapters 3 and 4 are reprints of the material as it appears in the proceedings of International Symposium on High Performance and Distributed Computing (HPDC), 2007. Lee, Cynthia B.; Snavely, Allan E., 2007. [48] The dissertation author was the primary investigator and author of this paper.

VITA

1996-1998	Parallel Systems Intern, NASA Ames Research Center
2001	B.S. University of California, San Diego
2000-2002	Software Engineer and Team Lead, Mohomine, Inc.
2004	M.S. University of California, San Diego
2004	Teaching Assistant, University of California, San Diego
2007	Teaching Assistant, University of California, San Diego
2007	Instructor, University of California, San Diego
2008	Instructor, University of California, San Diego
2002-2009	Research Assistant, University of California, San Diego
2009	Ph.D. University of California, San Diego

PUBLICATIONS

Carrington, Laura, Nicole Wolter, Allan E. Snavely and Cynthia B. Lee. "Applying an Automated Framework to Produce Accurate Blind Performance Predictions of Full-Scale HPC Applications." [12]

Lee, Cynthia B., Yael Schwartzman, Jennifer Hardy, and Allan E. Snavely. "Are user runtime estimates inherently inaccurate?" [46]

Lee, Cynthia B. and Allan E. Snavely. "On the User-Scheduler Dialogue: Studies of User-Provided Runtime Estimates and Utility Functions." [47]

Lee, Cynthia B. and Allan E. Snavely. "Precise and realistic utility functions for user-centric performance analysis of schedulers." HPDC'07: Proceedings of the 16th International Symposium on High Performance Distributed Computing, June 2006. [48]

FIELDS OF STUDY

Major Field: Computer Science
Parallel Computing: Scheduling and Performance Modeling.
Allan Snavely

Major Field: Computer Science
Machine Learning: Document Clustering and Classification

ABSTRACT OF THE DISSERTATION

On the User-Scheduler Relationship in High-Performance Computing

by

Cynthia Bailey Lee

Doctor of Philosophy in Computer Science

University of California, San Diego, 2009

Professor Allan Snavely, Chair

Professor Amin Vadhat, Co-Chair

To effectively manage High-Performance Computing (HPC) resources, it is essential to maximize return on the substantial infrastructure investment they entail. One prerequisite to success is the ability of the scheduler and user to productively interact. This work develops criteria for measuring productivity, analyzes several aspects of the user-scheduler relationship via user studies, and develops solutions to some vexing barriers between users and schedulers. The five main contributions of this work are as follows.

First, this work quantifies the desires of the user population and represents them as a utility function. This contribution is in four parts: a survey-based study collecting utility data from users of a supercomputer system, augmentation of the *Standard Workload Format* to enable scheduler research using utility functions, and a model for synthetically generating utility function-augmented workloads.

Second, a number of the classic scheduling disciplines are evaluated by their

ability to maximize aggregate utility of all users, using the synthetic utility functions. These evaluations show the performance impact of inaccurate runtime estimates, contradicting an oft quoted prior result [55] that inaccuracy of estimates leads to better scheduling.

Third, a scheduler optimizing the aggregate utility of all users, using a genetic algorithm heuristic, is demonstrated. This contribution includes two software artifacts: an implementation of the genetic algorithm (GA) scheduler, and a modular, extensible scheduler simulation framework that simulates several classic scheduling disciplines and is interoperable with the *Standard Workload Format*.

Fourth, the ability of users to productively interact with this scheduler by providing an accurate estimate of their resource (run time) needs is examined. This contribution consists of formalizing a frequent casual assertion from the scheduling literature, that users typically "pad" runtime estimates, into an explicit *Padding Hypothesis*, and then falsifying the hypothesis via a survey-based study of users of a supercomputer system. Specifically, absent an incentive to pad–and including incentives to be accurate–the inaccuracy of runtime estimates only improved from an average of 61% inaccurate to an average of 57% inaccurate. This contribution has implications not only for the proposed genetic algorithm scheduler, but for any scheduler that asks users for an estimate, which currently includes virtually all parallel job schedulers both in production use and proposed in the literature.

Fifth, a survey of users of a supercomputer system and associated simulations explore the feasibility of removing one of the defining constraints of the parallel job scheduling problem–the non-preemptability of running jobs. An investigation of users' current checkpointing habits produced a workload labeled with per-job checkpoint information, enabling simulation of a checkpoint-aware GA scheduler that may preempt running jobs as it optimizes aggregate utility. Lifting the non-preemptability constraint improves performance of the GA scheduler by 16% (and 23% compared to classic EASY algorithm), including overhead penalties for job termination and restart.

Chapter 1

Introduction

1.1 High-Performance Computing

The focus of this dissertation is the scheduling of scientific computing applications on high-performance computing (HPC) systems, also known as supercomputers.

The pace of improvement in performance of computer technology complicates any effort to formulate a constant, meaningful definition of *supercomputer*. Qualitative differences in hardware that used to exist between, for example, the Cray-2 [4] supercomputer and its contemporary Macintosh personal computer [7], are no longer always evident as commodity PC parts are commonly used in supercomputers and vice-versa. However, the "Top 500" [54] semiannual listing of the 500 fastest computers in the world has become a useful *de facto* definition of supercomputer [29, 23, 82, 53, 22].

A single system on the Top 500 list typically costs millions or tens of millions of dollars, and supports a large user population consisting of self-interested parties competing for its use. The significant financial investment that supercomputers entail has motivated a long history of scrutiny and innovation in how these resources should be allocated among users, with a wide variety of approaches.

Cluster and massively parallel processor (MPP) architectures[1], the focus of this

[1]The term *cluster* is generally understood to refer to a system comprised of nodes, "each of which is a system in its own right, capable of independent operation and derived from products developed and

work, constituted the majority of the Top 500 since 1994, and have continued to increase since then, now numbering 498 as of November 2008 [54]. Significantly for this work, both cluster and MPP supercomputers consist of hundreds or thousands of processors connected by a communication network to form a single system [24, 6]. This hardware configuration is well suited to parallel scientific codes that use a message-passing programming paradigm such as MPI [78]. These workloads characteristically rely on frequent, time-sensitive communication between processors. This workload requirement distinguishes cluster and MPP from more federated architectures such as cloud and grid (though there is some overlap in compatability), and motivate various constraints and assumptions about how the problem of scheduling these systems is approached in this and related work.

1.2 Problem Definition Overview

HPC workloads may consist of hundreds of jobs each day, and each job has unique resource requirements. How to best organize the running of these jobs is a mature yet active research area. The problem is to map jobs submitted by users onto blocks of time on subsets of the systems' processors. This domain of scheduling is known as parallel job scheduling, or parallel batch scheduling. Comprehensive reviews of the domain can be found in surveys by Feitelson *et al.* done in 1995 [33], and again in 2004 [31]. The problem is often abstracted as a two-dimensional bin-packing problem. Jobs are modeled in two dimensions, the number of processors required and the amount of time required on those processors. These rectangular-shaped jobs must be arranged on a plane representing the system, with the processors on one axis and time on the other.

marketed for other stand-alone purposes" [24]. By contrast, the term *MPP* usually refers to a system comprised of nodes, which, while they contain the components associated with a stand-alone system (*e.g.*, CPU, memory, network interface), were designed and manufactured with a massively parallel, high-performance deployment in mind. Due to the increasing overlap in design and use of components between low-end and high-end systems, some have argued that there is no meaningful distinction between the terms [6]. In any case, the properties shared by both architectures are conducive to being scheduled according to the constraints and assumptions used in the domain of scheduling under consideration here.

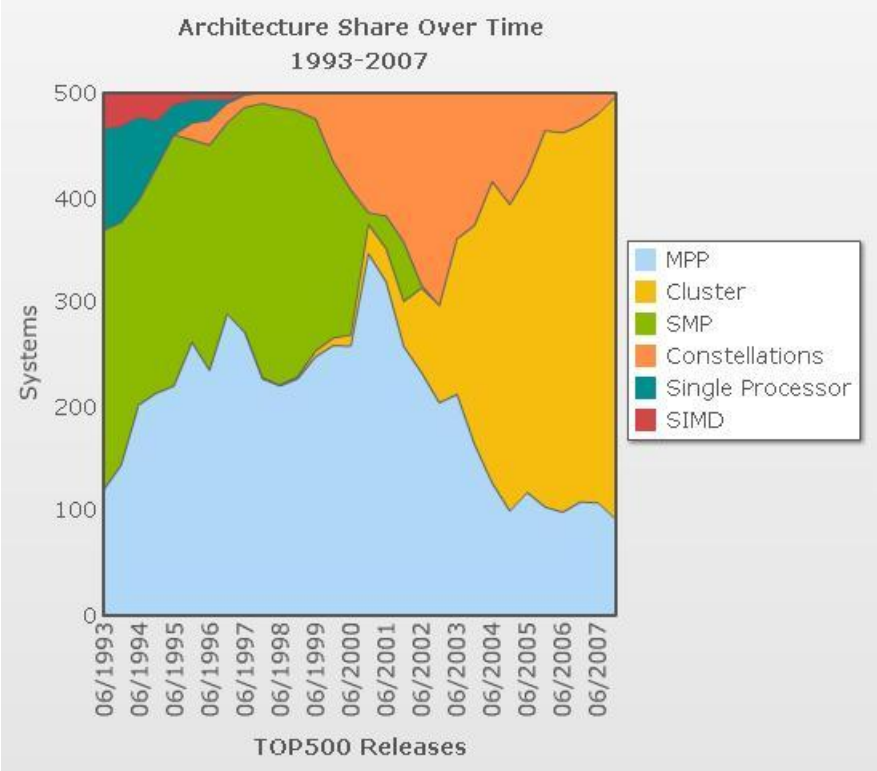

Figure 1.1: Share of Top 500 systems by machine architecture type.

The assumptions these models entail are consistent with parallel scientific codes using a message-passing paradigm.

This task of arranging the jobs belongs to the scheduler. HPC schedulers exist primarily to maximize user satisfaction, on the assumption that system owners' needs are best met when users are most satisfied. Most commonly user satisfaction is considered in terms of a job's *turnaround time*, the elapsed time between the *submit time*, when the user enters a request for the job to run (also called the *release time*), and completion of the job. In addition to speed of job turnaround, there are many other aspects to user satisfaction including predictability of job turnaround times, interface ease-of-use and availability of auxiliary services such as data storage and visualization infrastructure.

Throughout this work, the term *processor* will be used for the smallest unit of compute resource to be scheduled. On some systems, this unit might in fact be a several tightly coupled processors, especially if they share memory or other important resources. To reflect this grouping of processors, the term *node* is sometimes used in the literature, in place of *processor*, as the smallest assignable unit.

Some systems whose machine architecture consists of nodes of several processors each allow the node's processors to be scheduled individually. On others, only whole nodes may be assigned. For example, the Bassi system [57] at the National Energy Research Scientific Computing Center (NERSC) has 111 compute nodes of eight processors and a shared 32GB of memory, and is scheduled at the node level of granularity. NERSC's two-processor-per-node Franklin system [58], and the 32-processor-per-node Cheetah system at Oak Ridge National Laboratory (ORNL) [63], are also scheduled at the node level. The DataStar [71] system at the San Diego Supercomputer Center consists of both 8-processor and 32-processor nodes. The 8-processor nodes are scheduled at the node granularity while the 32-processor nodes are scheduled at the processor granularity.

Where to draw this abstraction will vary based on machine architecture. Jobs running on the same node typically share some resources (e.g., disk), and contention for these shared resources can lead to unpredictable and poorer performance. To pre-

vent these contention problems, scheduling at the node level is the most common. Since neither the term *node* nor the term *processor* is free from potential confusion, for simplicity, this work will in all cases use *processor* to denote the granularity of hardware assignable by the scheduler.

Treatment of parallel job scheduling as a static global optimization problem has been discussed in the literature [35, 25, 40]. Here it will be examined as an online scheduling problem, meaning that jobs arrive over time as a stream of input and the scheduler lacks knowledge of the future (jobs that have not arrived yet). Each time a running job terminates or a new job arrives in the queue, the scheduler is invoked to make a decision about which job(s) to start next, if any.

Optimization entails an objective. Defining the objective that should guide the scheduler's decision-making is not trivial. A wide variety of metrics for assessing success of schedulers have been proposed in the literature. Not all purport to be the ultimate objective in scheduling, but each reveals something about the priorities and beliefs of those who use them. A brief review of common metrics follows in Section 1.4.

1.3 Classic Approaches to Scheduling

To fully define the batch scheduling problem, it will be helpful to briefly review a few of the classic approaches to solving it. The most elementary is First-Come First-Serve (FCFS).

1.3.1 First-Come First-Serve

FCFS works as follows. Let J_0 be the first job in a FCFS queue, and let I be the number of currently available processors. The number of processors required by job J_i is denoted $J_i.p$. If $J_0.p \leq I$, the scheduler signals that J_0 should begin running. If $J_0.p > I$, then J_0 cannot begin running, and furthermore all the other jobs behind J_0 must also wait (because the queue is only accessed at its head), and the I

processors remain idle. Note that in this formulation, a FCFS scheduler does not require the user to specify the requested runtime of the job. Jobs are simply run until their natural completion, and then the next job is started when possible. The idleness caused by this blocking approach can lead to significant wasted resources in the wake of wide (many-processor) jobs.

1.3.2 Backfilling

Growing concern about poor utilization of resources [42] gave rise to backfilling algorithms. Backfilling is a policy of strategically allowing jobs to run out of order. In particular, if a job J_0 is blocked due to a lack of sufficient resources, then another job behind it may run if the available resources are sufficient for that job. Formally, this means allowing a job J_i for some $i \geq 0$, to "skip ahead" if $J_i.p < I$.

Experience has shown that backfilling improves utilization by about 20% on most systems and workloads, and greatly improves the response time for the small jobs, which are most likely to be able to backfill [39]. It is reported that over 90% of short, narrow (few processor) jobs are typically able to backfill; in particular the year-long CHPC workload trace shows over 90% of small jobs backfilling [39]. This is all while often providing moderate improvement for even the largest jobs and others that were not backfilled, because increased utilization of the resource causes the entire workload to be processed more quickly. As a result, backfilling has been described as "something for nothing," a benefit without a tradeoff, made possible because of the previous inefficiency of the system [39].

1.3.3 EASY Backfilling

The first explicitly documented use of backfilling is in the Extensible Argonne Scheduling sYstem (EASY), developed by Lifka for Argonne National Laboratorys 128-node IBM SP system. Eventually integrated with IBMs LoadLeveler administration software, EASY was "used by many MPP sites throughout the world and is known for

its efficient scheduling, simplicity, and robustness." [75]

The EASY scheduler orders jobs by arrival. Scheduling decisions are made when a new job arrives or a currently running job ends. If the job at the head of the queue can run on the available processors, it is started. Otherwise, EASY determines the earliest time at which the first job can begin running (according to the requested runtime for each running job) and scans the queue for a smaller job. The smaller job must fit both width-wise, on the available processors, and length-wise, i.e. its requested runtime is such that it would not delay the scheduled start time of the first job in the queue. This method of backfilling does not delay the first job in the queue, but it may delay subsequent jobs, which are still ahead of the candidate backfilling job.

1.3.4 Conservative Backfilling

Mualem and Feitelson [55] compare EASY to an approach they aptly term conservative backfilling, in which candidate backfill jobs are checked against every job ahead of them in the queue to see if the backfilling action will delay them. This is achieved by giving each job a reservation at the time it is submitted. New jobs may backfill as long as they fit in the cracks between the existing reservations.

One concern would be that following this more restrictive policy would give less efficient scheduling than the more aggressive EASY. But results show that for many workloads, conservative backfilling does not result in a loss in performance compared with the EASY algorithm [55].

Performance being equal, conservative backfilling is preferred because users are able to know their (worst-case) start time at the time the job is submitted. Then, like the EASY creators they corrected, the authors make the claim that conservative backfilling "guarantees that future arrivals do not delay previously queued jobs" [55]. However, the authors of the Maui scheduler [39] in turn proved this claim to be incorrect as well.

Although the reservation system in conservative backfilling guarantees that a job will never start later than its originally reserved time, backfilling jobs ahead of it may

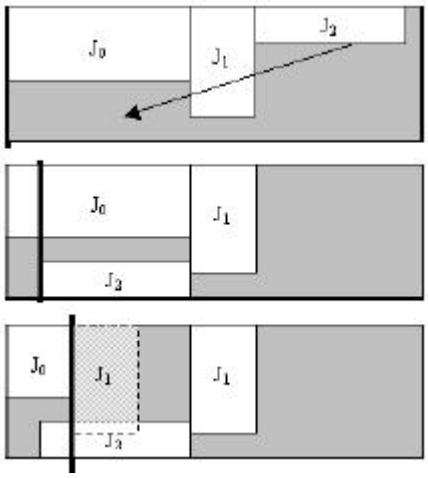

Figure 1.2: Pseudo-delay in Conservative backfilling.

still cause it to start later than it might have otherwise. This delay-by-stolen-opportunity is called pseudo-delay; perhaps not the best name because the effects of pseudo-delay are quite real. This seemingly contradictory situation arises because jobs may not, and most often do not, end up using the full time they request (unreliable user input will be discussed in more detail in Chapter 5). Thus when the scheduler checks that a backfill candidate job will finish no later than other critical-path jobs, it is relying on information that is most often not correct.

Figure 1.2 shows an example scenario, in three steps. In the first step, jobs J_0, J_1, J_2, queued in that order. The y-axis represents the machine's processors, and the x-axis represents time, with the current time marked by a vertical heavy black line. J_0 will begin running immediately, but there are not enough free processors to start J_1. However, there are enough free processors to start J_2, and according to J_2's requested time, J_2 will complete soon enough that backfilling J_2 will not interfere with J_1's planned start time. Then, according to conservative backfilling algorithm, J_2 is started (step 2 of Figure 1.2). But a moment later (step 3), J_0 ends before its full requested runtime has elapsed. Had we not started J_2, then J_1 could have started as soon as J_0 ended. But according to our non-preemption assumption, the decision to start J_2

is irrevocable now that J_2 has begun running. Although J_1 has not been delayed past its scheduled start time, it has still been prevented from running as soon as it could have, by a job that was behind it in the queue.

1.3.5 Priority FIFO

Chun and Culler [17] give the name PrioFIFO to fourth classic scheduling algorithm that approximates systems used in many supercomputer centers. FIFO stands for First-In, First-Out (essentially the same meaning as FCFS). Users assign their job to one of a handful of different priority categories. The job wait queue is partitioned into separate queues for each of these categories. The scheduler first attempts to schedule the job at the head of the highest priority queue that is non-empty. If there are not enough idle processors, other jobs may backfill. For backfilling, the scheduler considers all jobs in a given queue before considering any jobs from a lower priority queue.

1.4 Overview of Common Scheduler Metrics

At a very high level, the scheduler should be a proxy for the supercomputer's owner, and make decisions reflecting the owner's goals and values. Typically the owner's primary concern is satisfying the users, who in turn want their jobs to be completed as soon as possible. Most scheduling metrics reflect this by incorporating turnaround time of jobs in some way (perhaps weighting by various factors, *e.g.*, job size).

The following is a brief overview of several classic scheduling metrics:

Wait Time Number of seconds a job waited in the queue.

Expansion Factor Ratio of turnaround time (wait time plus runtime) to runtime.

Average Bounded Slowdown Defined in [55]. Similar to Expansion Factor, but includes two measures aimed at reducing noise in the result. First, runtimes are bounded at 10 seconds minimum to limit the impact of extremely short runtime

jobs on the average. Second, startup and shutdown effects are avoided by not including all *N* jobs in the workload in the calculation; specifically, the first N divided by 101 and last N modulo 100 are not included.

Makespan Number of seconds between the arrival of the first job and the completion of the last job.

Utilization At a given point in time, utilization is the percent of the machine's processors that are currently assigned a job as opposed to idle. For an entire workload, utilization can be calculated as the sum of the node-hours of each job, divided by the Makespan.

Aggregate Utility Sum, over all jobs, of the utility of the job. Utility is a function specific to each job, and could be a function of many things, but is commonly discussed as a function of turnaround time.

Although Utilization and Makespan are two of the oldest and most frequently cited metrics, [34] argues that they are not useful (and thus should not be reported) for comparing schedulers via workload-based simulation studies. This is because a simulation with a given workload covering weeks, months, or even years of history gives the scheduler comparatively little control over the difference between when the simulation begins and ends (the Makespan). The first job's submit time fixes the beginning, and the simulation can not end any sooner than the submit time of the last job plus its runtime. As an example, if the first job in a trace is submitted on January 1, 2000, and the last job in the trace is submitted on January 1, 2009, and is queued even as long as 3 days, that is just 3 days (or 0.09%) difference from the theoretical minimum Makespan for that workload.

1.5 User–Scheduler Communication

Resource scheduling typically involves a dialogue between a prospective user of a resource and a scheduler to determine when the resource can be used, and for how long. Part of this dialogue may also determine how urgently the user needs the resource, or, in other words, how the value of the resource depends upon when it becomes available. Intuitively, schedules formed in the absence of exact information in each of these categories may be suboptimal.

To understand what is meant by a *dialogue* between the user and scheduler, it will be helpful to examine the first supercomputer scheduler, the scheme of *Tennis Court Scheduling*. This scheme was used on most early supercomputers, for example it was the first scheduler on San Diego Supercomputer Center's Touchstone Delta (as described in a personal interview by an SDSC researcher [66]; for comparison, a contemporary Delta system, belonging to the Concurrent Supercomputing Consortium is documented in [52]). This "scheduler" was in fact a set of human system operators who were responsible for managing phoned-in job requests from users.

While on one level, Tennis Court scheduling is the height of unsophistication, and suffers the fatal flaw that it does not scale well to the large and busy systems of today, it can still provide an important perspective from which to judge software-based schedulers. Human beings possess creativity, flexibility and nuance of analysis that outclass proposed scheduling algorithms, if not in terms of bin-packing algorithmics, at least in terms of the total user interface. One can imagine a lengthy negotiation between user and operator over job parameters and schedule availability to achieve the most satisfying result for all parties concerned. Operators, knowing the habits, personalities and relative importance of their users could assess the urgency of each job and act accordingly: backfilling, rearranging already scheduled jobs and perhaps even stopping already running jobs. It is this complex and nuanced negotiation process that is here considered the gold standard of a dialogue between user and scheduler.

Current (software) schedulers have not re-attained the state-of-the-art in "user-

friendliness" of the Tennis Court scheduler times in terms of dialogue with the user. Modern scheduler communication takes the form of a user-provided job script that typically contains a requested runtime, a priority, the number of processors and other resources needed and essential information for executing the job.

This is crude communication and is also only one-way. Most schedulers provide no feedback, such as suggested modifications to the job to improve its wait time, which could be provided in Tennis Court scheduling. Often the only communication the user will receive is a notification that the job has started running, and that it has ended.

Most systems do allow the user to inquire about the state of the queue, for example how many other jobs are waiting and running, their sizes and their priorities. But the listings returned from these inquiries are often overwhelming and confusing. Even relatively savvy users are unable to use this data to determine an expected wait time for their job, or how they might alter their job to better fit the schedule.

One recent effort to remove the opacity in this process is the QBETS queue wait time prediction system [8, 62, 61]. QBETS and associated tools offer probabilistic queue wait time predictions and probabilistic advance reservations. Users are thus able to query the system regarding a variety of possible job configurations (scaling from longer runtimes on one or few processors, to shorter runtimes on larger numbers of processors) and determine, with real-time-adjusting feedback, which configuration is the most advantageous.

The remainder of this dissertation consists of further examination of the user–scheduler relationship: existing work on the matter, and novel explorations of–and solutions to–some of the most vexing problems therein.

1.6 Organization

The remainder of the dissertation is organized as follows.

In Chapter 2, the desires of the user population are quantified and represented as a utility function. This contribution is in three novel parts: a survey-based study of users

of a supercomputer system, an augmentation of the *Standard Workload Format* to enable scheduler research using utility functions, and a model for synthetically generating said utility function-augmented workloads.

In Chapter 3, a scheduler optimizing the aggregate utility of all users, using a genetic algorithm heuristic, is demonstrated.

In Chapter 5, the ability of users to productively interact with this scheduler by providing an accurate estimate of their resource (run time) needs is examined. This contribution consists of the formalizing a frequent casual assertion from the scheduling literature into an explicit *Padding Hypothesis*, and then falsifying the hypothesis via a survey-based study of users of a supercomputer system. This contribution has implications not only for the proposed genetic algorithm scheduler, but for any scheduler that asks users for such an estimate, which currently includes virtually all parallel job schedulers both in production use and proposed in the literature.

In Chapter 6, the feasibility of removing one of the defining constraints of the parallel job scheduling problem–the non-preemptability of running jobs–is explored via a survey of users of a supercomputer system. This contribution consists of a investigating users' current checkpointing habits, which data informs a simulation of a scheduler that may preempt jobs.

Parts of Chapter 1 are reprints of the material as it appears in the dissertation author's research exam (Comprehensive Exam) for the Department of Computer Science and Engineering, University of California, San Diego, 2005. [45] The dissertation author was the sole author of this paper.

Chapter 2

Job Turnaround Utility Functions

2.1 Introduction

High-performance computing (HPC) batch schedulers exist primarily to maximize user satisfaction, and system owners' needs are best met when users are most satisfied. There are many aspects to satisfaction, including speed of job turnaround, predictability of job turnaround times, interface ease-of-use and even aesthetics, and others. This chapter focuses on the value users associate with their jobs' turnaround time, and quantifying that as a utility function.

2.2 Related Work

2.2.1 Priority

Commonly used metrics such as response time, bounded slowdown and expansion factor (see Section 1.4) are all designed to capture the users' desires to not be kept waiting. Users' desire to not be kept waiting is axiomatic in the scheduling literature, and is also taken as given in this work. This section examines mechanisms by which users may communicate more detail about their desires to not be kept waiting.

HPC users have limited opportunities to communicate information about their

scheduling preferences to schedulers. One mechanism for communication between the user and scheduler is a user-selectable priority describing the relative urgency of a job. This mechanism is available on many past and current production systems. Priorities are typically selected from a short list of discrete options, *e.g.*, *High, Normal, Low*. These may not provide enough granularity for users to adequately express themselves. Using an analogy to mailing a package, postal patrons have a wide range of choices ranging from paying as little as 0.42 USD for slow ground shipping, to 30 USD or more for FedEx to deliver the package early the next morning.

Ironically, users' freedom to express priority has decreased over the years. More fine-grained priority choices were commonplace on SMP architecture supercomputers of the 1980's and 90's. For example, jobs on the Cray XMP were assigned a floating-point priority value between 0 and 2 by their owners. Users could change a job's priority at any time and as often as they pleased, whether the job was being held or running. A system administrator recalls some users becoming absorbed in this activity as one would a computer game. They continuously monitored and fine-tuned their priority values throughout the workday, and even wrote small software tools to assist these efforts [66].

The real-time nature of these vintage systems' priority scheme hints at a richer context to the interplay between users and the completion of their jobs than is captured with any static priority measure. A simple but ubiquitous piece of evidence for this time-dependent context is that virtually all HPC workload traces show diurnal patterns. That is, on average, more and smaller (presumably debugging) jobs are submitted during the day, and very infrequent new job submissions are made at night. If a scheduler sacrifices utilization in favor of very good response time for a particular job at 2 a.m., the effort is most likely wasted–the user will not even check the job until morning–but such a tradeoff could be highly desirable during peak business hours.

Users have both their own daily schedules and preferences (for example, sleeping and eating), and externally imposed schedules and preferences (for example, conference paper submission deadlines and national holidays). Along these lines, Feitelson *et al.* take a significant step in enriching our understanding of the user-scheduler dia-

logue by contextualizing users' scheduling desires in terms of their day-to-day activities. While most previous work studied scheduling at the scope of the digital world within the confines of computer systems' plastic cases, Feitelson *et al.* place the job in context of aspects of the users' lives that aren't even related to computers at all. The following thought experiment scenario exemplifies this:

> Assume that a job...needs approximately 3 hours of computation time. If the user submits the job in the morning (9am) he may expect to receive the results after lunch. It probably does not matter to him whether the job is started immediately or delayed for an hour as long as it is done by 1pm. Any delay beyond 1pm may cause annoyance and thus reduce user satisfaction, *i.e.*, increase costs. This corresponds to tardiness scheduling. However, if the job is not completed before 5pm it may be sufficient if the user gets his results early next morning. Moreover, he may be able to deal with the situation easily if he is informed at the time of submission that execution of the job by 5pm cannot be expected. Also, if the user is charged for the use of system resources, he may be willing to postpone execution of his job until nighttime when the charge is reduced [32].

In a Tennis Court scheduling scheme, the user in the example scenario would be able to convey in full the relevant details of his schedule and preferences to the system operator during the course of their dialogue. But, as noted above, the user-scheduler dialogue on current systems either allows no discussion on this topic, or, most often, allows the user to choose from 2 or 3 priority categories (*e.g.*, *High, Normal, Low*).

2.2.2 An Economics Approach

Although users of production systems have limited opportunity to express priority and preferences, proposed schedulers in the research literature have taken steps to allow more flexibility. Many of these generalize and unify the communication of preferences under an economic scheme. Stoica, Abdel-Wahab and Pothen [81] proposed a *Microeconomic Scheduler* that allows users to create expense account for each job, thereby expressing the job's priority. The system, in turn, can implement incentives for desirable behaviors from users, such as charging users not just for the time a job uses,

but for the idle time created in the job's wake, thus enlisting users in the effort to decrease fragmentation in the schedule. Feitelson and Rudolph [30] have postulated that the diverse sub-research-areas of batch job scheduling (*e.g.*, gang scheduling, dynamic partitioning) can converge under a framework with a flexible economic-based philosophy, and the work of Wolski *et al.* [96] and Buyya [10, 11] also point to an economic model. Singh *et al.* developed a method for adaptive pricing of making a reservation in a batch scheduled system, where the price depends on the cost imposed on other queued jobs by the reservation [74]. There are a number of projects in the Grid realm that are also approaching compute resource management from a microeconomic angle, such as Mirage [16], Tycoon [43] and Spawn [90].

Chun and Culler [17] allow users to express a willingness to pay for different turnaround times by way of a function, $u(t)$, where independent variable t is the turnaround time for the job. The function is expressed in units of some currency. In this dissertation, the currency is usually a system-specific "SU" as discussed in Section 2.5. In Chun and Culler, as well as subsequent related work [17, 2, 15, 38], this willingness-to-pay function is called a *utility function*, and this dissertation retains the terminology.[1]

2.2.3 Aggregate Utility

Utility functions can be used as the basis for a scheduling metric. Classic scheduling metrics such as average wait time, expansion factor or bounded slowdown, and, to a lesser extent, makespan, all strive to represent the notion that a good scheduler maximizes user happiness, *i.e.*, minimizes users' frustration due to being made to wait for results. Average wait time, expansion factor, and bounded slowdown all implicitly assume a plain linear decrease in value to each user over time, with all users and jobs

[1] In Microeconomics, utility is not simply willingness to pay but the difference between the willingness to pay and the charge the user is ultimately required to pay. These are treated interchangeably in Chun and Culler, and the convention is retained here. To the extent that the scope of using the 'utility' functions is to compare the relative efficiency of different schedules (allocations of the resource), the distinction does not significantly affect the analysis.

having the same start value and rate of decrease. As noted in Chun and Culler [17] (also [2, 15, 38]), when armed with workloads bearing job-specific utility functions, we can directly compute the total value delivered to users as the sum of each user's utility function, evaluated at their job's turnaround time:

$$Aggregate\_utility = \sum_{j \epsilon Jobs} u_j(turnaround\_time_j)[17] \tag{2.1}$$

In microeconomics terms, the aggregate utility metric is essentially a *purely utilitarian* approach to *social welfare*. That is, it encodes an assumption that the supercomputer's society's greater good is best served when total utility is maximized, without regard to the outcomes for individual users (apart from their contribution to the sum). If we assume that utility equates to revenue for the supercomputer center, it is clear why the maximizing revenue by maximizing aggregate utility is a reasonable representation of the goals of the center's managers. But is the satisfaction of individual users also maximized? This is a more nuanced question.

The aggregate utility metric necessarily gives more weight to the outcome for some jobs, namely those with higher associated willingness to pay, over other jobs. These jobs may tend to come from users whose initial endowments were the greatest. However, relying on our assumption that the allocation of initial endowments was done correctly, this cannot be considered an undesirable preference. The goal is simply to reach toward Pareto efficient outcomes, given the initial endowments. Again, these efficient outcomes are also conditional on users' truthfulness in their specification of their utility functions, which is in turn conditional on an assumption that truthfulness is incentivized and enforced by a price charged against their finite budget (an incentive compatible price mechanism).

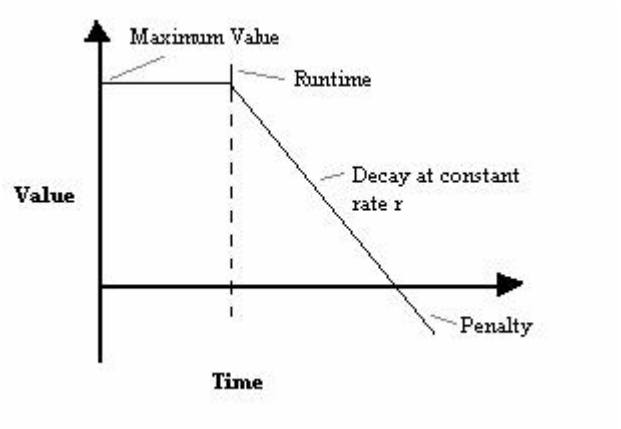

Figure 2.1: Utility model used in prior work [38].

2.2.4 Expressiveness of Utility Function Model

In Chun and Culler, the utility function is modeled as a simple linear utility functions with customizable maximum (starting) value and slope, as shown in Figure 2.1. Users may set the maximum value to an arbitrary positive number, and the slope to be an arbitrary negative number, greatly increasing expressiveness from the fixed priority category scheme. The form used in [38] also allows for a penalty for "late" completion of jobs, with the decay in value continuing on the same trajectory (even after it passes $u(t)$=0).

In this formulation, the utility function is essentially not defined during the time the job is running–although a value nominally exists here (the starting value), the *loss* of value that occurs during this time is not represented. This distinction is meaningless when narrowly considering scheduling on a single fixed system with consistent runtimes because the passage of time due to running the job is unchangeable and unavoidable.

However, users experience loss in value during the running time. Representing this loss in value due to running time would enable inquiries such as quantifying the value delivered to users of, for example, processor upgrades that reduce running time (even if queue time were to remain constant due to increased workload). Upgrades of HPC systems can cost millions of dollars, so calculating this utility impact for users may

be quite important. Another application would be quantifying the tradeoff in user utility of scheduling jobs to share processors or other resources, thus decreasing wait time but increasing runtime (such a scheduler has been proposed in [91]).

Considering again Feitelson *et al.*'s thought experiment scenario, the utility function $u(t)$, implied by the scenario is not a simple linear function. There are discontinuities (see at 1 p.m.) and periods where the slope is zero (see between 5 p.m. and the following morning). Furthermore, it is likely that every *(user, job)* pair will have a function with a different pattern.

This leads to the question of whether actual users could and would express complex job valuations available with an unconstrained utility function formulation, or whether the simple two or three category priority system or simple linear utility function is adequate to capture as much as they are willing to provide. This question is addressed Section 2.3 in the form of a survey experiment of users on the San Diego Supercomputer Center's IBM Power3 system, Blue Horizon [70].

2.3 Survey Experiment Design

Blue Horizon was a 1,152-processor IBM SP2 system installed at the San Diego Supercomputer Center (SDSC) [70]. Resource management was handled by IBM's LoadLeveler software [37], augmented with a scheduling program called *Catalina* [97] that was developed in-house at SDSC. Users of the Blue Horizon system submit jobs by using the command *llsubmit*, passing as an argument the name of a file called the job script. The script contains vital job information such as the location and name of the executable, the number of nodes and processors required, a requested runtime and a priority.

During the survey experiment period, the llsubmit program was modified so that before performing its usual functions, it administered a brief survey. The survey was only invoked in one of every five job submissions, selected randomly. Users were notified of the study, by email and newsletter, a week prior to the start of the survey period,

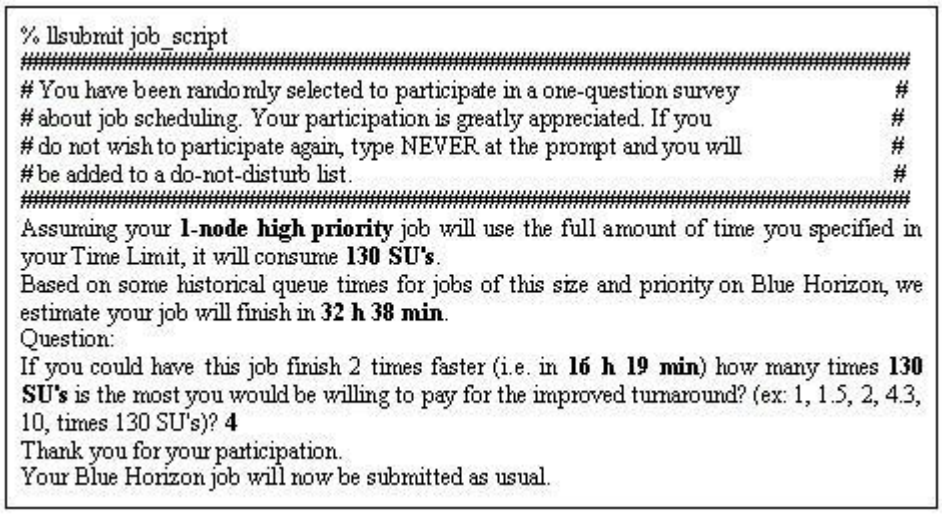

Figure 2.2: Sample user utility curve survey and response.

and could opt out ahead of time, or at any time when presented with the survey.

The moment of job submission was chosen because it is the most timely, and therefore most realistic, moment to measure the user's true valuations of their job. Specifically, in a scenario where users would be asked to participate in a market-based scheme of job scheduling, it would be at this moment that information from their utility function would need to be summoned and revealed.

The text of the survey is as follows. First, the user is reminded of requested number of nodes, time and priority queue. An estimate of the total turnaround time for the job (queue time and run time) is generated using historical data for similar jobs. The total cost for the job is also calculated (this is a function of nodes, runtime and priority). The user is presented with the turnaround time and cost, and also a hypothetical scenario in which the turnaround time is a factor of n times faster or slower, where n is varied according to a set pattern dictated by how many times the user has previously responded to the survey. The sequence is: 2, 1/2, 3, 1/3, 4, 1/4.... The questions are posed in that order for each user, with a log tracking which questions each individual user has already answered. A sample of the survey output is shown in Figure 2.2.

2.3.1 Truth of Users' Survey Responses

The stated preference methodology of this survey, while suited to collecting data for simulation-based research, is not suited to use in a production environment.

Here, the reported willingness to pay information (utility functions) are connected neither to how much users will ultimately be charged for their jobs, nor how those jobs will be prioritized and scheduled by the system. Consequently, users have no direct incentive to understate nor overstate, respectively, their willingness to pay. There may indirect causes of untruthfulness at work (habitual tendency to understate willingness, recklessness with hypothetical scenarios that promotes overstating, etc.), but this lack of direct incentives constitutes an adequate assurance of truth for the purposes of the survey.

The problem of fostering truthful revelation of utility functions (willingness to pay) in a production environment where the information will affect scheduling decisions will be addressed in Section 3.4.

2.4 Survey Results

It is commonplace among HPC users to have a pattern of work that consists of running the same code many times, using slightly varying input parameters each time. The nature of numerical simulation of physical phenomenon, from weather to fluid dynamics, is that the pattern of conducting experiments, or workflow, involves many repeated simulations with only slightly varying initial conditions. Since the purpose of the survey is to examine the complexity of valuation expressiveness users could provide, only the results for users who responded to the survey more than once per job are analyzed. For the purposes of analysis, a re-run of the "same job" is defined as any any job with the same user, processor count and requested runtime as a previous job. This definition potentially results in both false positives and false negatives in identifying a given set of jobs as being the same.

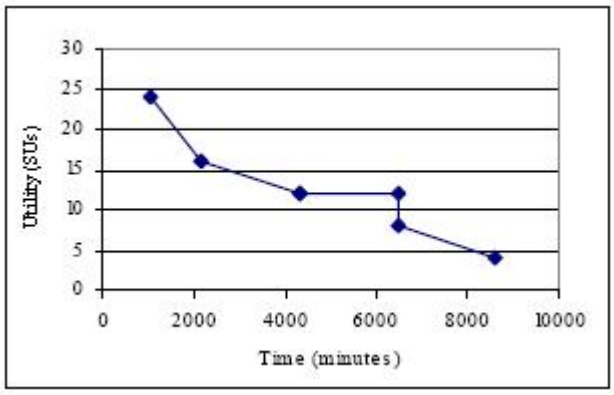

Figure 2.3: User utility curve.

Obviously many users do not have a workflow that entails repeatedly running the same job with the same processor/time configuration many times, and thus a majority of the survey responses were unfortunately discarded because they lacked additional survey data points. One way to avoid this would have been to collect several data points from the user at one moment in time. However, in order to secure permission to collect this information, the amount of time and effort an instance of the survey require of the user was constrained.

All the $(t, u(t))$ pairs for a given job are joined together into a single utility function. This is admittedly an imprecise interpretation of the data. Indeed, it is expected that users will have different needs and desires at different times, even for the same job. Thus, because the responses were elicited on different occasions, possibly spanning days or weeks during the month-long duration of the survey, it is possible to have different values of $u(t)$ for the same value of t. It is also possible to have points where the value of the job seems to have increased over time. Some exemplary utility functions are shown in Figures 2.3 through 2.9, one user per graph (some users have more than one job).

The first significant finding from looking at these utility functions is that none of them is linear. Further, many of the complex traits listed above in connection with the example narrative scenario are indeed observed in these actual user-provided functions. There are periods of time where the slope is zero (Figures 2.3, 2.5, 2.6, 2.7 and 2.9), a

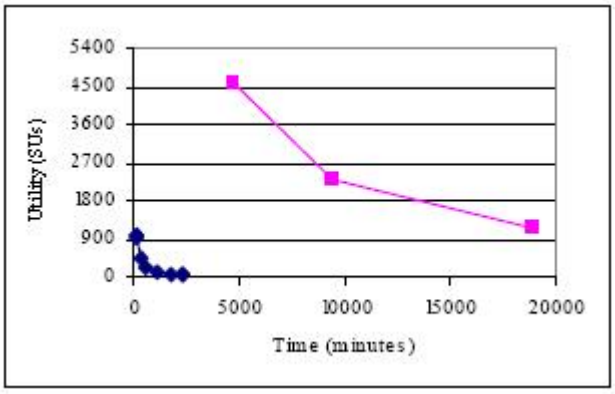

Figure 2.4: User utility curve.

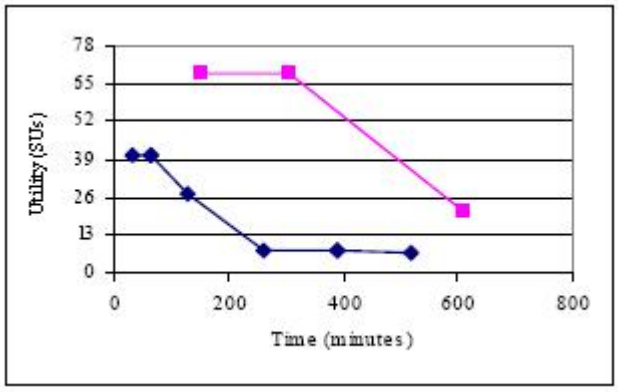

Figure 2.5: User utility curve.

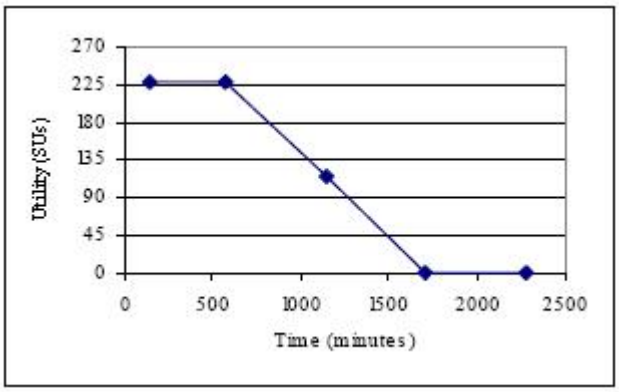

Figure 2.6: User utility curve.

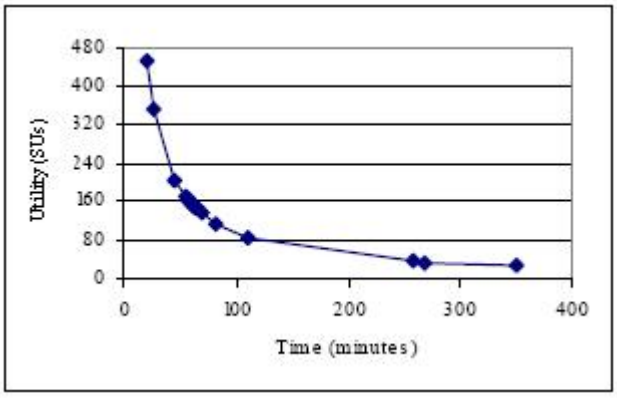

Figure 2.7: User utility curve.

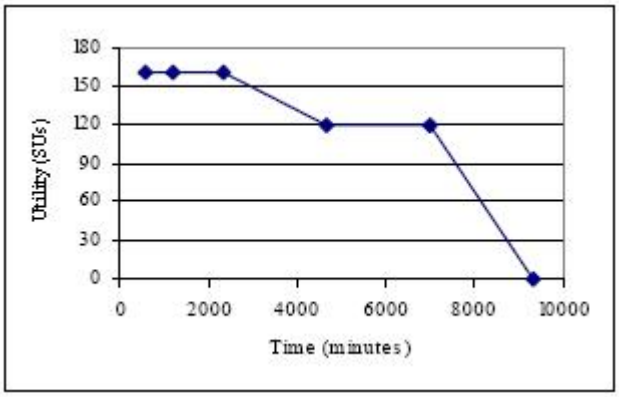

Figure 2.8: User utility curve.

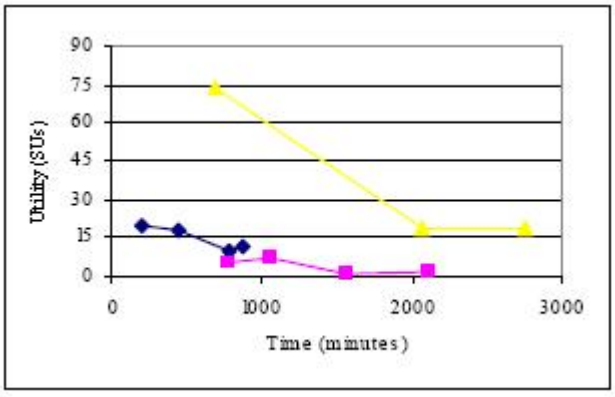

Figure 2.9: User utility curve.

very steep drop in value in the moments after job submission with a leveling off later (Figures 2.4, 2.5 and 2.8), and a deadline-like utility function where the value is constant until the deadline nears, causing a sharp decline in utility (Figures 2.5, 2.6 and 2.7).

While these results represent just a small sampling of users and jobs, they provide clear evidence that users not only have complex needs and desires regarding the scheduling of their jobs, but also that they are able to express themselves when given the opportunity. It is significant that these functions were elicited on a purely voluntary basis, from users who had little to gain by participating, and whose jobs were unaffected by their responses. Perhaps users would be even more willing to provide thoughtful detail if they were engaged in an actual dialogue with the scheduler and their input would have an impact on the scheduling of their job.

2.5 A New Utility Function Representation

Given this evidence that users are able to express their desires regarding the scheduling of their jobs, might a new and more complex utility function representation be justified?

There is a legitimate concern that users may not be willing to provide this level of detail in a real-life job submission setting. They already struggle to provide the information currently asked of them, such as job runtime estimates [46]. Colloquially, one might say that people may not be good at talking about their jobs, but if there is one topic everyone loves talking about, it is themselves. A utility function is not a property of the software; it is a property of the user. More concretely, previous work shows that such information is obtainable. Our future work involves detailed human factors studies to fine-tune the details of an agreeable interface for eliciting the utility function in a real job submission setting.

It is therefore proposed to use a continuous piecewise linear utility function representation, that allows users to specify the location of each data point, and even the number of points to provide. This format is both rich in descriptive power, and simple

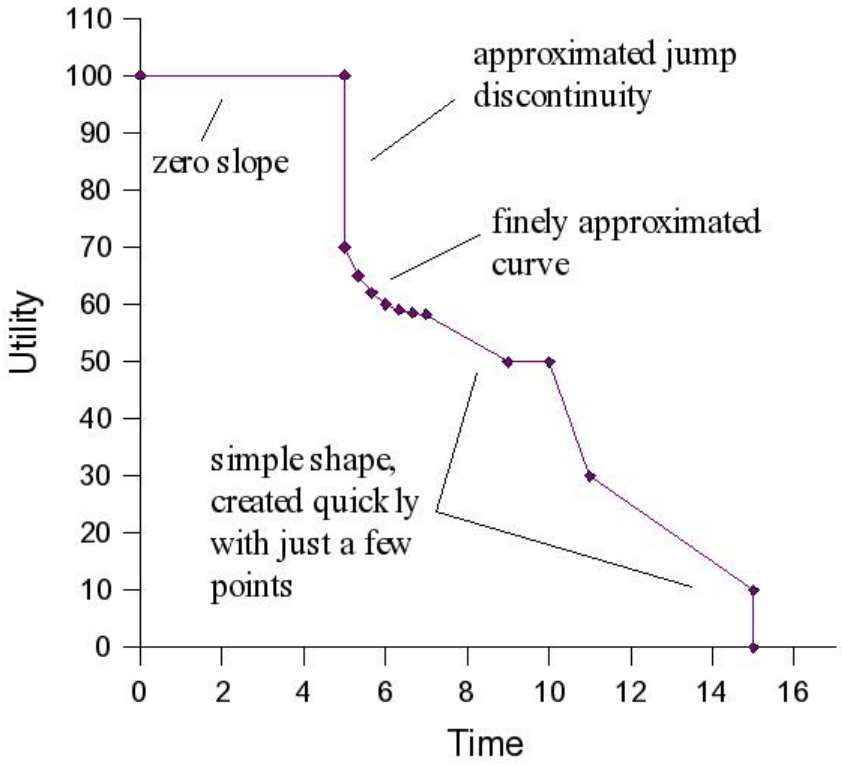

Figure 2.10: Demonstration of flexibility of proposed formulation.

to understand. Some of the possibilities for user self-expression afforded by this format are demonstrated in Figure 2.10.

Notice, in Figure 2.10, some of the features this formulation allows to be expressed: periods of zero slope ($0 \leq t \leq 5$ and $9 \leq t \leq 10$), approximated jump discontinuities ($5.0 \leq t \leq 5.001$ and $15.0 \leq t \leq 15.001$), a finely approximated curve ($5 \leq t \leq 7$) and a pattern perhaps more crudely drawn ($7 \leq t \leq 15$). The placement of points is user-determined both in terms of the $x-$dimension and the $y-$dimension. The number of points to include is also user-determined (minimum two).

A main goal of this work was to allow a high level of detail in specifying the utility function. This format achieves that; in particular, it is much more powerful than the purely linear (height and slope adjustable) formulation used in some previous work (shown in Figure 2.1). On the other hand, one must be mindful that obtaining a job's

utility function places a time and energy burden on users, who must stop to reflect on their desires and then enter the information.

Although many HPC users are trained in the sciences and would be familiar with various standard function shapes (linear, exponential, hyperbolic, etc.), it would be unwise to rely on this knowledge by requiring users to input formula parameters for these shapes. First, submitting a job should not require a mathematics glossary or formula reference. Second, the format should not impose a predetermined function shape on the user. Thus, this formulation meets the two essential requirements: it is easy for any user to input and flexible.

Piecewise linear formulation also has an intuitive mapping to the type of typical daily routine-linked changes in value seen in Feitelson *et al.*'s example scenario and in the survey of real users.

Finally, this formulation has the benefit that users can make as much or as little effort as they please. From a single line segment of two points, to a finely approximated curve composed of many segments, users can tailor their level of effort to the benefit they perceive can be extracted from the scheduler.

The function is stored as a series of (time, value) tuples. The time in the first tuple must always be set to zero, and this represents the time at which the job was submitted. The value in the first tuple is the initial (maximum) value for the job if it were completed instantaneously. The units of value depend upon the system. Allocation units or service units (SU) are already in use at many HPC center, but in commercial and other settings, it may be units of real currency (*e.g.*, US Dollars). The domain for both times and values is the set of non-negative real numbers. Reading the series from start to finish, times must be strictly increasing, while values are decreasing (non-strictly, *i.e.*, periods where the slope is zero are allowed). Jump discontinuities are approximated by using two different times t and $t + \varepsilon$, with some very small ε.

The value at any time greater than the last time listed in the sequence is defined to be zero (the sequence may also explicitly contain time(s) at which the value is zero).

Using simple linear interpolation between the user-provided data points, the se-

quences of tuples are interpreted as continuous piecewise-linear functions.

Thus all well-formed function specifications represent functions that are defined over the domain of all times $t, 0 \leq t \leq \infty$. (Note that since $t = 0$ is the submit time of the job in question, times must be normalized to some absolute clock for comparison amongst different jobs.)

2.6 Synthetically Generating Utility Functions

The following is a proposed model of generating synthetic utility functions that, to the extent possible, incorporates knowledge and research about real user preferences. A software implementation of the utility function generation methods described here is available; please direct inquiries to the authors.

Tsafrir [86, 85] and others [83] have elucidated the many subtle attributes of real workloads and how they can have unexpectedly significant impact on performance. Tsafrir's work underscores the need to rely on actual data as much as possible. Anticipating a time when workload traces including users' own utility functions are available, it is necessary to propose this statistical model in the interim, in order to perform the simulations necessary to justify asking them of an entire user population.

2.6.1 Input Data for Synthetic Functions

Workloads in the Standard Workload Format (SWF) form the basis of the synthetic workload generation. Many logs of actual HPC workloads are available in this format from the Parallel Workloads Archive [13, 28]. Each line of the file represents one job. Standard job description data such as number of processors, submit time, runtime, priority, and so on, are provided. The Standard Workload Format was extended by appending the list of tuples that define ourthe utility function to the end of each line.

There are three sources of information to draw on when forming a model for generating utility functions: 1) each job's actual user-assigned priority and completion

time, 2) data on utility function shapes from the previous study [47], and 3) the distribution of actual wait times from the log, which gives guidance about user expectations.

The priority found in the workload data is the main clue as to how the user values a specific job. However, this is at best a loose guide to the initial value of the utility function and none at all to its overall shape. The user has the equivalent of a utility function in mind simply by virtue of being a human with needs and desires. In selecting a priority for the job, the user was forced to project that utility function curve onto a single dimension, one with a severely limited domain at that (for example, just 2 or 3 choices). It is not clear exactly how this projection of the function was or should be carried out. If a user's job has a high maximum value, indicating importance, but retains the value for a long duration, indicating lack of urgency, would that user have assigned it a *Low* or *High* priority? There is no obvious answer, and probably different users have different methods of doing this mapping. Some may effectively have no method at all–they may not see an appropriate choice for expressing their needs and wants, and just arbitrarily select one of the options. Details of how this data is used are in the next section, but it should be emphasized here that this method does not claim to perform the reverse of this projection in a way that is historically accurate in any individual case. This would be impossible. The goal is merely to generate utility functions that do not contradict the available data about the general shape of utility functions from other real users.

The model also incorporates data about utility function shapes from the study previously mentioned. Again, this information is a loose guide. Each system, user and job are unique, and data only exists for a certain group at one site. But the synthetic utility function model attempts to incorporate commonly seen traits from the collected curves.

Each workload has a different priority scheme: number of different priorities, restrictions on eligibility of jobs for different priorities (commonly, a limit on number of processors or time), etc. These reflect policy choices by the administrators of the various systems. In the SWF format, priority is encoded as an integer, with header com-

ments giving meaning to the values. Some human interpretation of this will be needed for each different workload. This model currently incorporates a generalized version of commonly seen priority systems where the different priorities have a strict lowest-to-highest ordering that makes sense according to the semantics given in the header comments of the SWF file. It is also assumed that the step between each increasing priority is of equal distance in terms of value to the user. The representation of the priority choices is normalized to be natural numbers $0, 1, , N_{priorities} - 1$, where 0 signifies the highest priority and $N_{priorities} - 1$ the lowest ($N_{priorities}$ is the number of different priorities in the system). For example, *Low, Medium, High* would be normalized to $Low = 2, Medium = 1, High = 0$.

2.6.2 Generating the Starting (Maximum) Value for Jobs

First, the maximum (starting) value for the job is determined. This is based on the priority for the job and its size (number of processors times requested time).

The space between zero and *globmax* (an arbitrary global maximum value) is divided into $N_{priorities}$ bins of equal size, where $N_{priorities}$ is the number of priority choices available on the system in the input workload. The model should reflect that, in general, higher priority jobs should have higher start values than lower priority ones. So the starting value for the job, $startvalue_{unscaled}$, is randomly selected using a Normal distribution centered over the bin corresponding to the job's priority (as shown below in the formula for mean). This causes most, but not all, jobs to have a start value proportional to their priority.

$$mean = \frac{priority + 0.5}{N_{priorities}} * globmax \tag{2.2}$$

This randomly selected starting value for the job is the value per processor-minute requested. So to compute the value for the entire job, the start value is scaled to reflect the job size:

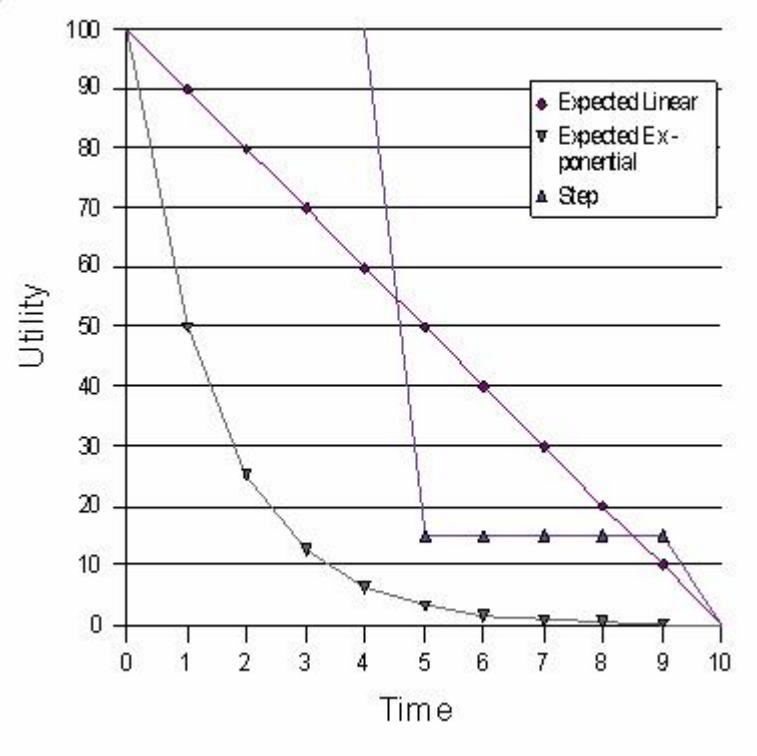

Figure 2.11: Three models of decay in utility.

$$startvalue = startvalue_{unscaled} * processors * time \tag{2.3}$$

2.6.3 Modeling Decay Patterns

The first tuple in the utility function is $(0, startvalue)$, as described above. To generate the remaining tuples, three different models of decay in value are used: expected linear, expected exponential, and step (see Figure 2.11). These represent roughly the three categories of patterns observed in the survey of actual user utility functions.

Expected linear represents users whose utility value decays mostly linearly over time. Expected exponential represents users whose jobs have high initial urgency (a much more precipitous initial decline); however, after some time has passed, additional

delay causes a diminishing marginal decay in utility. In both cases, expected refers to the possibility of some "bounce" or random out-of-pattern behavior. Step represents users whose jobs remain at their peak value for some amount of time, then experience a step-function-like immediate drop in value, then after another flat period, lose all their value.

The first step in generating these decay patterns is to select the time at which the job will finally lose all of its value, called the *deadline*. The deadline should reflect, as closely as possible, real user expectations for wait times for their jobs. Wait time expectations are dependent on the priority chosen, but also heavily dependent the job's size (number of processors and duration). To capture the nature of this dependency and real expectations of distribution of wait times, we base the deadline on the actual time, from the log, that the job originally waited, setting deadline to be twice this time (or 10 seconds, whichever is greater).

Now, with the first and last times and values of the sequence selected, the intermediate times and values are selected. For expected linear and expected exponential, randomly select a predetermined number of unique times between zero and deadline. For step, randomly select one intermediate deadline between zero and deadline. The times are then sorted in increasing order. The values corresponding to this sequence of times are selected as follows. For expected linear, values are selected randomly between zero and *startvalue*, inclusive. For expected exponential, values are iteratively selected randomly between the last value selected (or *startvalue*) and zero, inclusive. Values do not need to be unique. Values are sorted in decreasing order and matched to the sequence of times to complete the tuples.

2.7 Conclusion

Utility functions are widely used in economics as a means of expressing possibly complex changes in the value of a completed job over time. To discover if users of HPC systems had more complex feelings about their jobs than can be expressed in the simple

and discrete *High, Normal and Low* priority choices available on most systems, a survey was conducted. At the time of job submission, a software program asked how much more or less they would be willing to pay for earlier or later job completion, respectively. In the survey, users did indeed express a wide variety of valuations and function shapes. Even for the same user, valuations vary from project to project, and from day to day on the same project. This is evidence that overly simplistic formulations cannot capture the nuances of real users' preferences. This dissertation exhibited a method that can capture those nuances without being overly complicated, and provides a tool that scheduling researchers can use to generate realistic utility functions for the purposes of augmenting job logs. Whole user happiness is no doubt a function of more variables than just turnaround time of jobs. It could include for example, predictability of wait times, friendliness of staff, ease of use of storage systems, availability of debuggers and other auxiliary tools, and so on. Utility functions are a tool that can be extended to cover these variables as well.

Parts of Chapters 2 and 5 are reprints of the material as it appears in International Journal of High Performance Computing Applications, 2006. Lee, Cynthia B.; Snavely, Allan E., 2006. [47] The dissertation author was the primary investigator and author of this paper.

Chapter 3

Genetic Algorithm Scheduler

3.1 Introduction

This chapter introduces a novel approach to scheduling that uses a genetic algorithm (GA) heuristic to evolve a priority queue ordering of the jobs seeking to maximize aggregate utility. One purpose in developing this algorithm was simply as a way to explore the question, "How well is it possible to do?" In other words, what is the opportunity-space to maximize aggregate utility under realistic assumptions about users' utility functions? Testing of a prototype implementation of the algorithm suggests it is also suited to use in real-time, production supercomputer scheduling.

Classic approaches to scheduling, exemplified by EASY and Conservative backfilling (described in Section 1.3), consist of two components: an ordered list of jobs, and a defined procedure for transforming that list into a schedule in two dimensions (processors and time). In the case of EASY and Conservative backfilling, the procedure consists of ordering the jobs by arrival (traditional queue) and of waiting for enough processors to become idle so that the first job in the queue may be started. EASY and Conservative backfilling differ in their criteria for allowing jobs further back in the queue to fill in gaps.

While EASY and Conservative backfilling seek to refine the procedures for

transforming the ordered list (queue) into a schedule, other approaches to scheduling have instead (or additionally) addressed the queue itself, through the mechanism of a priority calculation. For instance, the Maui scheduler [39] uses a priority queue sorted according to a single priority number. That number is calculated from a weighted combination of a complex set of interconnected factors, configurable for each system, including: what auxiliary resources were requested by the job, how long the job has already been queued ("aging" factor), how many jobs have bypassed the job due to backfilling, how much of the fiscal year's total allocation the user (or group) has already consumed, quality of service (QoS, *i.e.*, priority) request from the user and the difference between the job's current wait time and a target wait time set for that user by the system administrators.

The Genetic Algorithm (GA) scheduler also takes the approach of addressing the list itself, by using evolutionary refinement to optimize the ordering of the (priority) queue such that, when provided as input to a classic scheduling discipline like EASY, it produces a schedule that optimizes the desired metric.

3.2 Algorithm

In each scheduling iteration, the GA scheduler produces a planned mapping of jobs to processors from now into the foreseeable future, and suggests which job(s) to start immediately. *Foreseeable future* in this case means that all jobs that are currently in the queue are scheduled. It does this by speculatively testing out many possible schedules. These schedules are selected in a process that has a stochastic element, but is guided by the goal dictated by a given objective function. However, rather than trying to evolve a schedule directly, the GA algorithm actually evolves an ordering of the jobs.

One might ask, why not directly optimize the schedule itself? To do this, a fundamental requirement is that one have a process for enumerating possible schedules, from which one could select the best, according to a given metric. In the case of two-dimensional HPC system schedules, there does not seem to be a straightforward way

to do this, given the constraints on jobs not overlapping, and having arbitrary widths (processors) and lengths (runtime).

Fortunately, the analogy with biology provides a useful abstraction for separating the hard-to-enumerate two-dimensional schedule forms from an easy-to-enumerate underlying specification that can be transformed into a schedule. In biology, a distinction is made between an organism's present manifestation in terms of form, development and behavior, or its *phenotype*, and the organism's DNA that gave rise to that manifestation, the *genotype*. Observing and describing, much less enumerating, all possible aspects of an organism's phenotype is generally intractable. However, the DNA code constitutes a concrete and manageable piece of data. Similarly, the GA scheduler distinguishes between the schedule (phenotype) and the priority queue ordering (genotype) in order to sidestep the enumeration problem.

3.2.1 Individual Genotype

This priority ordering can be thought of as the genotype of the individual. Then the phenotype, in this analogy, is the schedule derived from presenting the jobs, in the genotype order, to some scheduling algorithm.

After the genotype (queue ordering) is rendered as a phenotype (a schedule), the measure of the schedule's fitness can be taken. The fitness function could be any scheduling metric. In the experiments provided here, the fitness function is the aggregate utility of the schedule (see Equation 2.1 in Section 2.2.3).

Thus, the individual in the population's genotype is modeled as an ordered list of the job identifiers for all jobs currently in the queue (*i.e.*, currently available for scheduling). The model of an individual is diagrammed in Figure 3.1. A population is seeded with random permutations of the job identifiers. In experimental results given here, the population also includes a few special seed individuals whose jobs are sorted by, variously: arrival time, priority (secondarily by arrival time), current utility per node-hour, and absolute current utility.

Figure 3.1: Modeling an individual in the population.

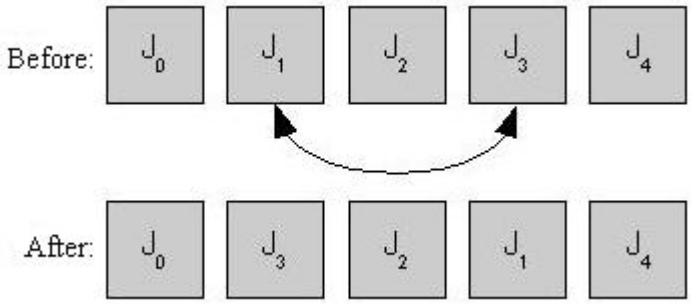

Figure 3.2: Modeling point mutation.

3.2.2 Point Mutation

Point mutations are simulated by selecting two jobs in the list at random, and swapping their places, as shown in Figure 3.2. Point mutations occur on each individual with a customizable probability ϕ.

Thus far, the GA scheduling algorithm consists of a straightforward application of the genetic algorithm optimization method widely used in a variety of application areas of computer science. However, the GA scheduling algorithm deviates somewhat from the traditional model in how it simulates reproduction.

3.2.3 Reproduction

Traditional genetic algorithms model sexual reproduction by randomly selecting a location on the "DNA" string, and copying everything up to and including that location from one parent, and everything after that location from the other parent. This presents a problem for the way the genotype is represented in the model described above. Namely,

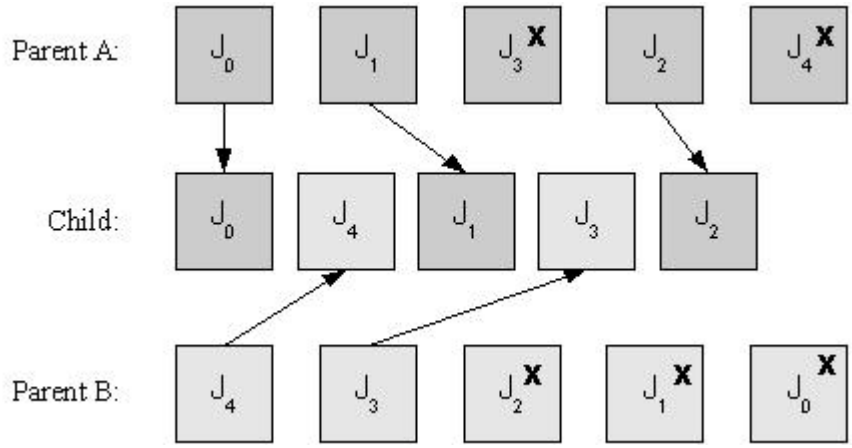

Figure 3.3: Modeling sexual reproduction.

the resulting child will not have a permutation of the job queue, but will have a list that is missing some jobs and contains duplicates of other jobs. Rather than change the model of the individual so that it functions correctly when using the traditional method of reproduction, a novel method of reproduction will be introduced.

Sexual reproduction is modeled in the GA scheduler by combining the two parent queue orderings in a way analogous to a clothing zipper. Specifically, one of the two parents is chosen at random. The first job in this parent's queue is popped and becomes the first job in the child queue. Then a parent is again chosen at random, the first job of that parent is popped, and becomes the second job in the child queue, and so on. If the popped job to be added to the child's queue is already present in the child's queue, one must continue popping jobs from that parent's queue until a non-duplicating job is found. This ensures that the child's list includes each job identifier exactly once.

Reproduction is illustrated in Figure 3.3. Jobs are popped from the parent queues into the child queue, alternating between Parent A and Parent B. When popping J_3 from Parent A, it is discovered that J_3 is already present in the child queue and the next job from Parent A, J_2, is used instead. A black 'X' denotes a job duplicate job that is skipped (*e.g.*, Parent A's J_3).

Reproduction happens in the context of one generation of a population, con-

sisting of individuals with different values according to the fitness function. The GA scheduler follows the traditional genetic algorithm approach to selecting which individuals' genes will propagate to the next generation. That is, individuals with higher fitness are more likely to mate and produce offspring than individuals with lower fitness. A predetermined number of individuals exist in each generation, and this number is constant from generation to generation. To generate each individual of the next generation, two parents are selected at random.

If the population set is P, then probability of selecting an individual x as the first parent of a child is given by:

$$Pr[select\ x\ as\ parent] = \frac{fitness(x)}{\sum_{i \in P} fitness(i)} \tag{3.1}$$

Selection of the second parent is done *without* replacement, in other words, a child must have two distinct parents. Selection of parents for different children is done *with* replacement; in other words, an individual may have more than one offspring (with the same partner or a different partner).

3.3 Prototype Implementation Details

This section describes a particular sample implementation of the GA scheduler algorithm described above in Section 3.2.

The GA scheduler was written in the Python programming language [88, 89], adapting Python code provided in conjunction with *Artificial Intelligence: A Modern Approach*, by Russell and Norvig [60, 69].

All the results presented in this dissertation, for the GA scheduler and comparison algorithms were generated on using Python implementations within a common framework that was also written in Python.

The framework runs a discrete event simulation to process the stream of incoming jobs, manages the simulated compute resources, and performs scheduling metric

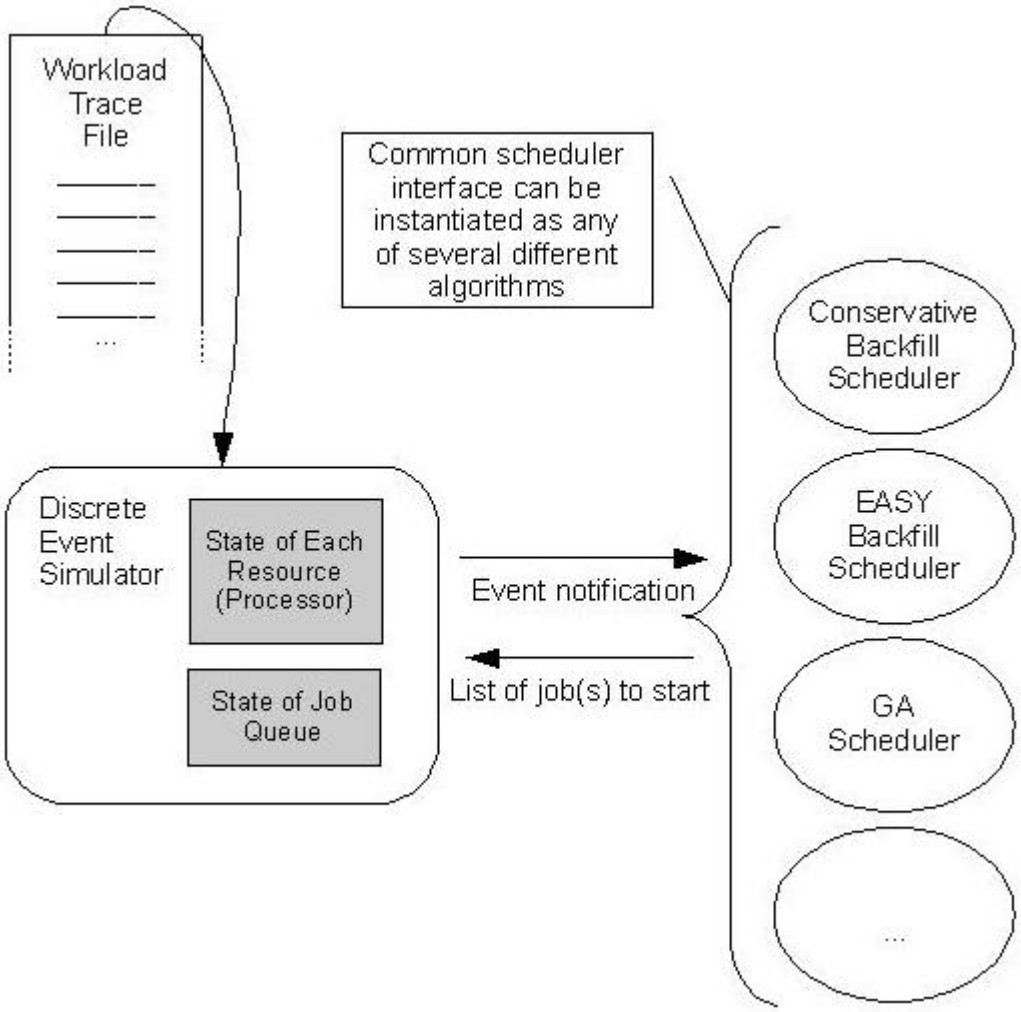

Figure 3.4: Design of the scheduler simulator framework.

bookkeeping. The scheduler component is abstracted so the various algorithms to be tested may be substituted within the framework. The abstract scheduler takes as input the entire set of queued jobs, and a list of compute resources, each noted as either available or with the time when the currently running job's requested time will expire. This design is similar to that of, *e.g.*, the Portable Batch Scheduler (PBS) [36].

One drawback to using an optimization approach like a genetic algorithm is that it is not deterministic. However, results show that the performance of GA was highly consistent across runs. For example, in 10 repeated trials using the same workload,

aggregate utility ranged from 1.32×10^9 to 1.38×10^9, with a standard deviation of 1.72×10^7, or 1.27%. Because variation is possible, results for GA presented hereafter are the average of two runs of GA per synthetic utility function augmentation of the workload in question (also done twice), for a total of 4 trials per test or configuration.

Of course, the major drawback to using a heuristic optimization instead of an efficient deterministic algorithm is the (potentially) much longer execution time. The GA scheduler can be run for any length of time, with the fitness score of the schedule tending to increase the longer it is run. (It separately stores the best individual seen thus far in any generation, so it is not possible for the fitness of the returned schedule to decrease over time, even if all individuals in later generations do manage to become inferior to one in a previous generation.) In a production environment, the scheduler is required to reach a decision in a small, relatively fixed amount of time. Thus the number of individuals in each generation is fixed, as is the number of generations to calculate.

Unless indicated otherwise, the number of generations of reproduction was limited to 100, with 20 individuals in the population of each generation. With this configuration, GA took an average of 8,900 seconds (measured on a desktop PC) to process an entire workload of 5,000 jobs, representing three weeks of operation of the supercomputer. Comparing to the following classic scheduling algorithms, it was just 110 seconds for conservative backfilling (Section 1.3.4), 35 seconds for Prio-FIFO (Section 1.3.5) and 30 seconds for EASY (Section 1.3.3).

Although slower than other algorithms, performance of the GA scheduler is well within a the time requirements of the task of real-time scheduling of a contemporary HPC system. In that environment, the scheduler must select which job(s) to start whenever a new job arrives or a running job terminates. The GA scheduler takes less than two seconds to do this calculation (8900s/5000 decisions = 1.8s/decision). According to SDSC system logs, approximately 90 seconds elapses between jobs as the system "cleans up" and does other tasks. Thus the two-second latency of GA's decision calculation is insignificant. Furthermore, the calculation could be performed much faster than two seconds if even just one processor of the HPC system were to be used for this

purpose, as opposed to the desktop PC used to produce these timings.

3.4 GA Scheduler as a Market

To this point, we have assumed that the user utility functions (willingness to pay) associated with each job were obtained by the GA scheduler essentially via omniscience. This section examines how the utility functions could be *truthfully* elicited, including how users should be charged for their usage of the system.

3.4.1 Auction Model

The problem of allocating processors to jobs in exchange for a fee can be framed as a combinatorial auction problem. Combinatorial auctions are auctions in which many goods are simultaneously available for sale. Finding Pareto-optimal allocations in combinatorial auctions is complicated by the presence of complementarity and substitutability in users' preferences. Complementarity means that a customer would prefer to consume one good with (an)other good(s). For example, a user bidding on a left shoe is most interested in winning the auction if he also wins the auction for the complementary right shoe. Substitutability means that more than one good may fill a customer's desire, and the user would be happy to bid on any or all of them, but the customer only wants to ultimately win the auction for one (or some number less than all) of them.

Complementarity is seen in parallel job scheduling, because the user (typically) needs all of their requested processors at the same time in order for the job to run at all. Complementarity also exists on the time dimension, in that the user would like to occupy the processors continuously until the job completes[1]. Substitutability is seen in parallel job scheduling because the processors are homogenous and the user doesn't have reason to prefer any one subset of them to another, given equal size and start time.

The "gold standard" of combinatorial auction design is the Generalized Vick-

[1]There are some technical workarounds for this constraint, which will be discussed in Chapter 6.

rey Auction (GVA) [49], using the Vickrey-Clark-Groves (VCG) pricing scheme, where each winner pays the opportunity cost imposed on all other participants by their presence as a winner. This is an incentive-compatible generalization of the standard *Vickrey auction* (or *sealed-bid, second-price auction*) for a single good, where the 1st-place bidder, the winner, pays the amount of the 2nd-place bid.

Lehmann et al. [49] analyze parallel job scheduling as a combinatorial auction and show that we cannot rely on the guarantees of VCG in the case where the allocations are only approximately optimized. This is a critical issue because the parallel job scheduling optimization problem is known to be intractable [35], and inexact optimization algorithms must be employed. They propose a system for clearing combinatorial auctions for parallel job scheduling markets where truth revelation is a dominant strategy, but this property only holds for a "severely" restricted class of users.

Mirage [16] is an actual deployment of auctions for compute cycle scheduling, on a grid environment as opposed to supercomputer environment. Mirage uses an iterative sealed-bid, first-price auction. The mechanism is not strategy-proof, and although one might hope that users would be too naive or unsophisticated to realize this, Ng et al. [59] observed strategic user behavior on the Mirage system. Related work by Munk *et al.* [56] uses an Expected Externality Mechanism for parallel batch queue systems that only accepts single-processor jobs, in which truthfulness is a *Nash-optimal strategy* (optimal if one assumes other users are also following the strategy). However, the variable number of processors (parallel) job case remains an open problem, and this is a key requirement for supercomputer environments.

The GA scheduler can be viewed as an algorithm for clearing combinatorial auctions for processor time. Because it relies on heuristic optimization, the GA scheduler does not guarantee to find the schedule that perfectly optimizes Aggregate Utility (total willingness to pay). It also doesn't readily provide an opportunity cost figure for each successful bid, a barrier to using VCG pricing. This is because the variety of job sizes, in processors and time, doesn't allow for a straightforward analysis of what would have happened to the schedule in the absence of a particular job. The effects could cascade

across the whole schedule in unexpected ways. The lack of guaranteed optimization and direct calculation of shadow price are challenges in the design and analysis of a price mechanism for the GA scheduler.

Section 3.4.3 will further discuss the issues around possible price mechanisms and propose a solution. First, an exploration of the analogy with electric grid markets, in Section 3.4.2, will suggest the feasibility of a similarly structured solution.

3.4.2 Power Grid Market Analogy

Using the GA scheduler to calculate a schedule, taking into consideration competing user desires and associated willingness to pay data, forms a hybrid between a centralized, planned economy and a pure decentralized auction between individual buyers and sellers (processors, anthropomorphized). Such a scheme does not fit neatly into an analytical category that provides desirable properties like guaranteed Pareto-optimality. However, it is not without precedent in the real world. The electric grid market in the United States, as documented by Wilson [95], also functions with elements of auctions combined with central planners responsible for ensuring feasibility of allocations.

Electric power grid market clearing and processor scheduling are problems that share several key traits. For example, the high cost of storing power is analogous to the impossibility (infinite cost) of storing computer cycles. Because both electric and compute cycle demands vary over time, capacity planning becomes an issue. Power market supply and demand must be balanced at all times, necessitating ample reserve capacity to avoid extreme responses to spikes in demand, for example rolling blackouts. The drawback is that they suffer from low average load factors. In parallel processor scheduling, queues are a ubiquitously employed mechanism for managing variation in demand.

Both power markets and processor scheduling feature users of varying degrees of sophistication. Unsophisticated users have less willingness and ability to respond to rapid price signals, less ability to forecast demand, and less information about their own

demand even ex post. Other users may be experts in the very technology and algorithms used to schedule the processors and thus able to devise and deploy gaming techniques [59], if the system is vulnerable to any. Similarly, some power market users, notably many households, put little thought into their patterns of consumption of electricity, while others are highly sophisticated, even generating, and selling the excess of, their own power.

3.4.3 A Price Mechanism

Chapter 5 details the challenge of eliciting accurate runtime requests from users. As will be seen in Section 4.6, inaccurate runtime requests can have a detrimental effect on scheduling results. There is a second piece of possibly-inaccurate data furnished to the GA scheduler by users–the utility function. Inaccurate utility functions can likewise be detrimental to the performance of the scheduler. Recent work by AuYoung et al. [3] empirically analyzes effects on utility-based scheduling systems when inaccurate utility information is provided due to user uncertainty, and finds that performance is tolerant to some inaccuracy. But malicious gaming of utility function reporting must be addressed. Careful design of a *price mechanism*, or system for setting how much users pay for their use of the system, can incentivize users to provide truthful utility functions.

GA within a Single-Price Framework

Under the current scheduler on, for example, an SDSC system, users are charged a fixed price regardless of eventual turnaround time of the job. This standard price is $1SU$ per processor-hour.

What would happen if the GA scheduler were to be dropped into such a system, requiring users to provide a utility function, determining the schedule based on the utility functions, then charging everyone a fixed $1SU$ per processor-hour price? Chaos would quickly ensue as self-interested users claim higher and higher utility functions in order to bypass others in the schedule. Users would have no reason to limit their reported utility

functions because they are not penalized for over-reporting. The utility value required for a job to run would be limited only by the maximum integer value. Even assuming that users' innate sense of morality prevents dishonest utility function over-reporting in every case, this system still suffers from another problem. Users who might be interested in contributing to the smooth operation of the system by allowing their jobs to run only in very low-load times–in exchange for a discount from the 1*SU* fee–are unable to do so because the fee is fixed. They will truthfully indicate their flexibility, only to unfairly pay the same price as everyone else.

This rigid price scenario is obviously pathological, but in fact even traditional SDSC systems operate under a more flexible system.

GA within a Multiple-Queue, Multiple-Price Framework

On SDSC systems such as Blue Horizon [70] and Datastar [71], users may have the option to submit to a *High* priority queue, which purports to offer shorter turnaround time, and for which they will be charged 2*SUs* per processor-hour. At various times, SDSC has also offered a *Low* and *Express* queues at 0.5 and 1.7 *SUs* per processor-hour, respectively.

This system has drawbacks: the price is not directly connected to outcome, and users have no way of knowing if the High priority queue offers better service for any given job–even after the fact. But it contains the rudiments of a well-designed price mechanism, namely, users' desire for priority service is tempered by their limited budgets and the surcharge for the High priority queue. Would this also help temper the pathology seen in the previous scenario using the GA scheduler?

Such a system would require a way to map utility functions to the set of queue choices {*Low, Normal, Express* and *High*}. The effectiveness of the price in controlling users' behavior depends in part on the accuracy of this mapping, but there is an inherent loss of fidelity. A first-order approximation would be to bin users based on the starting (maximum) utility value, but this ignores other urgency signals such as the deadline.

Another problem is that users who are willing to pay the rate for the *High* category have no disincentive to wildly overstate their utility function, because the charge will never be greater than the *High* rate no matter how high they bid. Similarly, users in other categories would bid up to the maximum for their bin. Each category would then degenerate into a competition decided by the timing of their submissions. AuYoung [3] and Chun [15] studied scheduling schemes with binning elements and noted similar problems.

This system, though simple and vulnerable to problems, may be feasible in some settings. For example, environments where a degree of collegiality and extra-price-motivated "good behavior" can be assumed.

GA as an Approximate First-Price Auction

A further refinement would be to charge users based on their individual utility functions and turnaround times. The simplest of this type of system would be to charge users their willingness to pay, in other words, to evaluate their utility function at the turnaround time actually achieved in the schedule. The advantages are that it is very simple to calculate, parallels the value the GA scheduler uses internally, and provides direct pressure on users not to overstate their willingness to pay. It establishes a quasi-first-price auction. First price auctions are known to suffer from a truthful-revelation problem, which is that users are more "gun-shy" about revealing their true maximum willingness to pay.

GA as an Approximate VCG Auction

To mimic a VCG auction, we need to calculate the opportunity cost the running of each job imposed on the system. As noted above, directly calculating this value for parallel job scheduling is an open problem. However, the opportunity cost of the decision to schedule a given job J_i can be approximated after the fact by re-using the GA schedule optimization engine to simulate schedules with J_i removed. The procedure

is as follows:

1. After J_i completes, collect the workload history for the set of all jobs in the queue at the time J_i was submitted, union the set of jobs that arrived while J_i was running. This set of jobs is denoted $\{J_0, ..., J_i, ..., J_n\}$.

2. Calculate the Aggregate Utility for $\{J_0, ..., J_i, ..., J_n\}$, evaluating each job's utility function at the turnaround time that actually occurred in the workload history. Call this $AU_{true,all}$.

3. Subtract the utility of job J_i from $AU_{true,all}$ and call this AU_{true}.

4. Simulate the GA scheduler on jobs $\{J_0, ..., J_{i-1}, J_{i+1}, ..., J_n\}$, to produce an (approximately) optimized schedule with just those jobs (job J_i is not present).

5. Evaluate the Aggregate Utility of the schedule in (4) and call it AU_{alt}.

6. The difference $AU_{true} - AU_{alt}$ is the approximate opportunity cost of the decision to schedule J_i.

This procedure needs to be done separately for each job, and is computationally expensive (relative to first-price auction prices). Fortunately, this calculation is not time-sensitive like the schedule calculation, and can be done offline as part of a periodic billing cycle. Note that, assuming one calculates the same number of generations of evolution as is done in the actual scheduling decisions, this calculation takes the same amount of time. Namely, about two seconds per job for which to calculate a price, or about 3 hours to do the billing for a whole month's worth of jobs.

For a more accurate approximation, the window of the workload history examined could be extended further into the future by letting J_n be the last job to arrive in the queue within a 24 hours, a week, or two weeks after J_i finished running. Setting J_n to be the last job submitted while J_i was running is a first-order approximation including only those jobs that could be directly impacted by J_i's run. Even if the time horizon

were to extend infinitely far into the future, the opportunity cost calculation would still only be approximate, because GA is not guaranteed to find the optimal schedule.

It is possible that the approximate opportunity cost calculation would result in a negative number. This could indicate that the GA scheduler happened to find a better schedule during its price calculation simulations than it had managed to find during its scheduling optimization, or that uncertainty about the future (arrival of new jobs and inaccurate runtime requests) impaired its ability to find the most efficient schedule originally. In this case, the price should revert to a default value, say zero.

3.5 Conclusion

The GA scheduler evolves an ordering of the jobs and runs the EASY scheduling algorithm on the resulting priority queue in order to obtain a decision about which job(s) to start in each round of online scheduling.

Each individual in the population to which "evolution" is applied represents a possible way of scheduling the currently available jobs. The individual's genes consist of an ordering of the available jobs. Evaluating the fitness of each individual, and thereby how many offspring it will produce, is done by estimating the aggregate utility of the resulting schedule (including all jobs scheduled out indefinitely into the future).

As a heuristic optimization approach to scheduling, the GA scheduler is non-deterministic and can be run for any length of time, with the estimated quality of the schedule increasing over time. With caps on the number of different schedules to explore as currently configured, the GA scheduler runs in an amount of time that is well suited to a production HPC environment.

Several pricing schemes of varying complexity and effectiveness in promoting truthful utility function revelation were proposed for the GA scheduler.

Though it has been shown that the GA scheduler has acceptable performance in terms of runtime, it remains to be shown how it compares to other scheduling algorithms in terms of the quality of the resulting schedule. Simulation experiment results

addressing this question are presented in Chapter 4.

Parts of Chapters 3 and 4 are reprints of the material as it appears in the proceedings of International Symposium on High Performance and Distributed Computing (HPDC), 2007. Lee, Cynthia B.; Snavely, Allan E., 2007. [48] The dissertation author was the primary investigator and author of this paper.

Chapter 4

Scheduler Evaluation Results

4.1 Introduction

The following experiments were conducted using a workload with utility functions generated using the synthetic utility function generation methods described in Section 2.6. The base workload is SDSC-Blue [26], available from the Parallel Workloads Archive [13, 28]. SDSC Blue has 1,152 processors and six main priority categories. Only jobs 5,000 through 10,000 are used, thus avoiding the atypical patterns at the beginning of the trace when the machine was first being opened to users and utilization was very low. The real-time duration of the time period (jobs 5,000 through 10,000) being simulated is three weeks. It therefore includes cyclical changes in workload and job arrival rates that occur on both diurnal and weekly bases.

The schedulers to be compared are several of the classic scheduling algorithms: EASY backfilling, Conservative backfilling, and Priority-FIFO (simple supercomputer scheduler policy where all jobs from a higher priority queue are considered before any in a lower priority queue and EASY backfilling within these constraints). These are compared against the utility-function-aware GA scheduler.

Generation of synthetic utility functions, as described above, relies in part on pseudorandom selection of parameters; even for the same base input workload, result-

ing utility functions will differ each time they are generated. Thus all results reported in this paper are the average of the result on two different workloads (same base workload, augmented with utility functions two different times). Recall that GA has its own variability, and is run four times, twice on each of these two workloads.

The metrics used in the following were defined in Section 1.4. Examining the impact of the scheduler on the full population of jobs, not just an average, gives much richer insight. Therefore in this work, results are presented for several of the metrics as percentile ranges (for example, 75% of jobs had a Wait Time of 73,881s or less using the Conservative backfilling algorithm).

4.2 Comparison of Schedulers by Various Metrics

In addition to metrics defined in Section 1.4, a metric used here is *Percent of Job's Start Value Earned*, which is simply the ratio (percent) of the job's earned utility and the maximum (starting) value for the job (or, $u(T_{complete})$ where $T_{complete}$ is the complete time of the job).

Of the classic algorithms (all but GA), Priority-FIFO performed the best on all workloads, and on all metrics except expansion factor. Priority-FIFO and GA have made an explicit choice to favor some jobs over others, using the priority information associated with the job. Priority-FIFO and GA significantly outperform Conservative and EASY for aggregate utility. This is not surprising since they are the only algorithms that can take priority or utility into account. GA outperforms Conservative, EASY, and Priority-FIFO on aggregate utility by 53%, 14%, and 6% respectively.

According to several of the metrics shown in Table 4.1, the GA scheduler not only delivered better performance for the highest-priority, "elite" jobs, but also spread the improved performance out to a broader set of jobs. For example, GA scored 96.0% of start value earned at the 25th percentile, compared to 93.3% for Priority-FIFO. At the same time, the very worst-off jobs, at the 98th percentile, GA scored almost 2.84% of start value earned, with just 1.04% for Priority-FIFO. A small improvement in per-

Table 4.1: Comparison of Schedulers by Various Metrics

	Wait Time (s) 25%ile 50%ile 75%ile 98%ile 100%ile	Expansion Factor 25%ile 50%ile 75%ile 98%ile 100%ile	Average Bounded Slowdown	% of Job's Start Value Earned 25%ile 50%ile 75%ile 98%ile 100%ile	Aggregate Utility
Cons	0 1,375 16,746 73,881 112,103	1.0 1.30 2.25 34 146	131	78.9% 11.5% 0% 0% 0%	8.77e08
EASY	0 0 5,372 59,067 116,552	1.0 1.0 1.55 28 131	76	90.4% 40.8% 0.055% 0% 0%	1.16e09
Prio	0 0 2,733 72,403 319,822	1.0 1.0 1.30 22 353	73	93.3% 53.1% 1.04% 0% 0%	1.27e09
GA	0 0 822 62,758 1,355,885	1.0 1.0 1.12 34 2261	148	96.0% 59.1% 2.84% 0% 0%	1.35e09

centage points, but it shows that GA is not delivering better "elite" performance at the expense of the majority of jobs.

GA does result in severe starvation of less than 2% of jobs (see the 100 percentile wait time in Table 4.1). Supercomputer centers would have to carefully consider, at a policy level, whether or not this is acceptable. If the schedule, turnaround times of all jobs, and ultimately overall user satisfaction, are significantly negatively impacted by just a small number of jobs, it may make sense to consider whether they should be, for example, asked to move to a different and more suitable venue.

4.3 Comparison of Schedulers by Utility Decay Type

The default behavior of the synthetic utility function generator is to generate approximately one-third of the workload using each of the three decay patterns: expected linear, expected exponential, and step. If one has reason to believe that a particular workload of interest would be more heavily populated by one of these types, it would be useful to know if certain schedulers perform better on that type. Figure 4.1 shows this comparison using the aggregate utility metric.

Of the three types of decay models, the expected exponential decay model resulted in the lowest score for each algorithm (this is to be expected). Jobs simply lose value too quickly to complete all of them before some lose much (or all) of their value. This is also the decay model on which the GA scheduler's performance is highest relative to the other algorithms, an improvement of 40% over the average of the others and 20% over Priority-FIFO.

4.4 Comparison of Schedulers by System Load

The most difficult test of a scheduler may be its performance in heavy load conditions. When it simply isn't possible to run all jobs in a reasonably timely manner, what should be done? Schedulers perform a balancing act–triage of the importance of

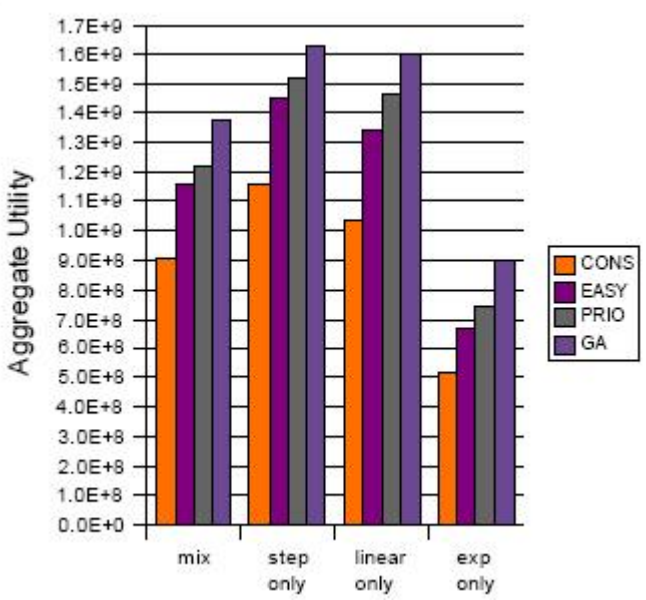

Figure 4.1: Scheduler performance by decay type.

Figure 4.2: SDSC Blue's load variation by time of day.

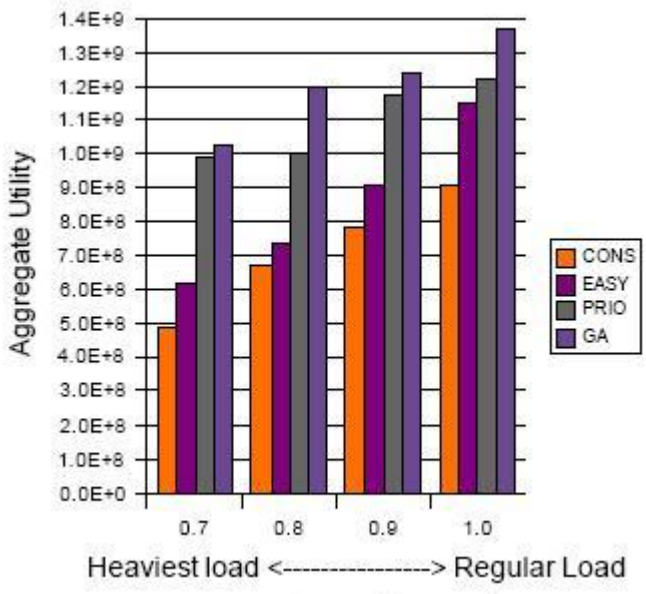

Figure 4.3: Scheduler performance by job load.

each job and how long it has already waited, while keeping an eye on how each action might impact the global good. Perhaps an important job's dimensions (processors and duration) are such that it cannot be fit in without doing an amount of damage to the scheduling of the rest of the jobs that outweighs the benefits.

The SDSC Blue workload already has variation in load, sometimes reaching quite heavy loads, according to local time of day and day of the week (load by time of day shown in Figure 4.2, graph from the SDSC-Blue workload [26] documentation in the Parallel Workloads Archive [28], used with permission.). This load is exacerbated by scaling the inter-arrival times of jobs in the log by varying factors, as shown in Figure 4.3.

Priority-FIFO and GA perform extremely well in high load conditions. The greater the load, the greater the difference in performance between these algorithms and those that do not consider priority or utility. Priority, expressed through a queue choice or a utility function, is a powerful tool to help schedulers manage heavy load.

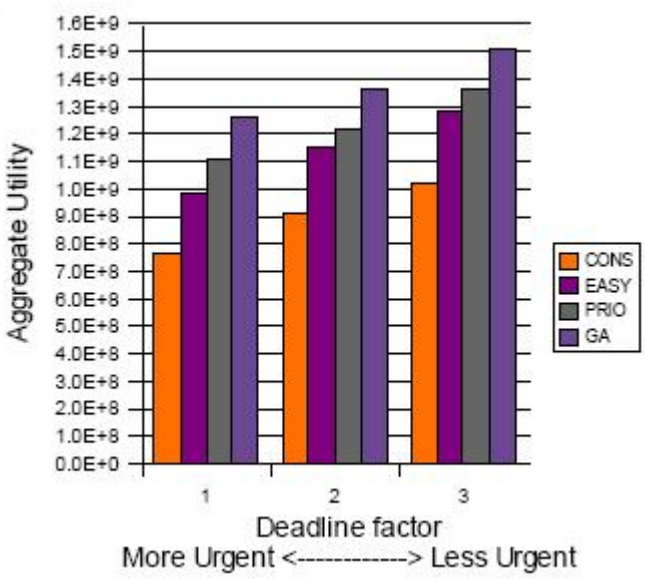

Figure 4.4: Scheduler performance by deadline urgency.

4.5 Comparison of Schedulers by Job Deadline Urgency

Another way to stress a scheduler is to scale the deadlines of the utility functions down, so that jobs are more urgent. Recall that deadline is set to be twice the actual wait time for that job recorded in the workload trace. New workloads are generated where this factor is one or three, as shown in Figure 4.4.

Naturally, all schedulers perform better when they have more time to schedule jobs (deadline factor = 3). Under deadline factor = 1, EASY proves more resilient with this method of stressing a scheduler than it was when load was increased by scaling inter-arrival times (Section 4.4, Figure 4.3). This is probably because one thing EASY does well is aggressively backfilling small jobs, resulting in a large percentage (even the majority) of jobs having no wait time whatsoever (see Table 4.1). Thus no matter how soon the deadline of these jobs is moved, provided it is nonzero, EASY will still earn their full utility.

4.6 Comparison of Schedulers by Sensitivity to Runtime Accuracy

For the algorithms EASY and Conservative backfilling, there is a well known and yet surprising result that they perform as well or better with inaccurate requested runtimes (relative to actual runtimes) than with completely accurate ones [55]. This result is surprising in that it goes against the "garbage in, garbage out" mantra of computer programming; typically, poor quality input results in poor quality results. In this case, the claim is that a noisy input signal is beneficial to scheduler performance.

Extending this idea, [98] showed that with more and more inaccuracy (larger and larger multipliers on the real runtime), EASY and Conservative degenerate into approximating the Shortest Job First algorithm, which is the optimal algorithm for minimizing average wait in uniprocessor settings. So, according to [98], inaccurate requested runtimes serve to transform a less-than-ideal algorithm into a nearly ideal one. This raises the question of whether a algorithms that do not suffer from this issue initially would still follow the "garbage in, garbage out" law.

As we explore the aggregate utility metric, and a new algorithm, GA, it is prudent to validate or invalidate the inaccuracy result of [55] as it might apply to these.

Figure 4.5 compares each scheduler's performance on the aggregate utility metric when using inaccurate requested runtimes (the actual user-provided requested runtimes), versus using completely accurate requested runtimes. All schedulers except EASY had marked improvement in performance, according to this metric, when using the accurate requested runtimes. EASY's slight decline in performance is consistent with the prior result of [55] when using the average bounded slowdown metric (though it contradicts the result for this workload, as will be seen in Figure 4.6).

Table 4.2 shows the comparison of schedulers using a variety of metrics, all run using completely accurate requested runtimes (*i.e.*, requested runtime = actual runtime). These results should be compared to results with real–inaccurate–runtime requests, in

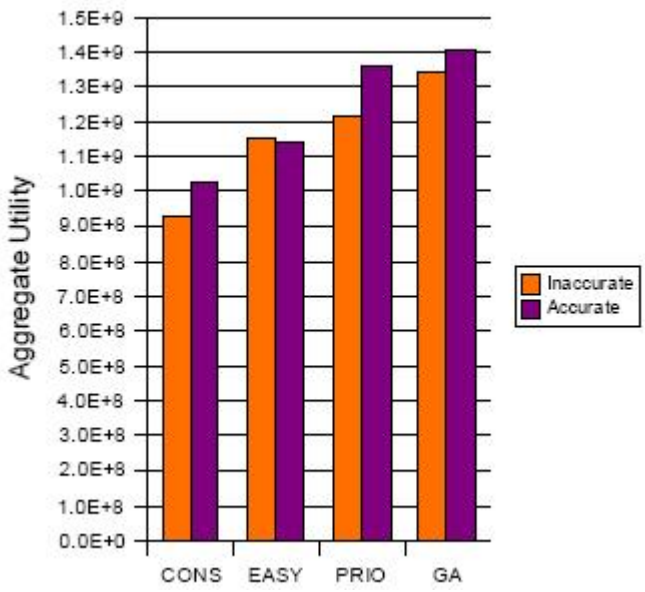

Figure 4.5: Scheduler performance with real user runtime estimates and fully accurate requests.

Table 4.1). The key comparisons between inaccurate and accurate results (combining columns of Tables 4.1 and 4.2) are shown in Figure 4.6.

Figure 4.6 can be summarized in two main points. According to the Aggregate Utility (AU) metric, all schedulers–except EASY backfilling–improve when provided with accurate runtime requests. On the other hand, according to the Average Bounded Slowdown (ABS) metric, all schedulers–except Conservative backfilling–get worse when provided with accurate runtime requests.

There are several interesting aspects to these results. First, that EASY gets worse with accurate runtime requests, according to to the Average Bounded Slowdown metric, differs from the prior result of [55]. This can be attributed to the experiments using different workloads, but shows that EASY's improvement in the ABS metric with inaccurate runtime requests is not a universal property of EASY. It is also notable that, although ABS and AU agree in their ranking of algorithms in virtually all experiments reported here, they produce opposite movement when comparing accurate and inaccu-

Table 4.2: Comparison of Schedulers by Various Metrics With Fully Accurate Runtimes

	Wait Time (s) 25%ile 50%ile 75%ile 98%ile 100%ile	Expansion Factor 25%ile 50%ile 75%ile 98%ile 100%ile	Average Bounded Slowdown	% of Job's Start Value Earned 25%ile 50%ile 75%ile 98%ile 100%ile	Aggregate Utility
Cons	0 3,621 27,009 246,923 307,491	1.0 1.76 5.28 78.2 393	116	73.9% 0.02% 0% 0% 0%	1.03e09
EASY	11 7,666 35,657 198,790 252,240	1.0 2.0 6.62 99.2 353	164	68.2% 0% 0% 0% 0%	1.14e09
Prio	0 383 6,592 159,417 775,694	1.0 1.1 1.98 42 861	97	89.5% 30.4% 0% 0% 0%	1.36e09
GA	0 140 3,483 381,516 1,153,529	1.0 1.02 1.52 105 1,923	224	91.2% 40.6% 0.003% 0% 0%	1.41e09

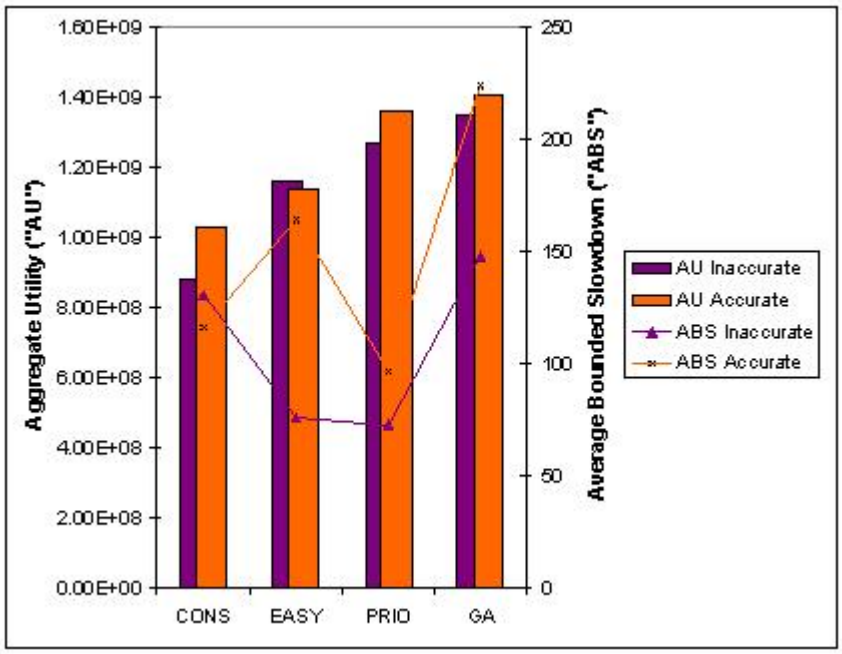

Figure 4.6: Scheduler performance by runtime accuracy, comparison of two metrics.

rate runtime requests.

The problem of inaccurate user runtimes will be further investigated in Chapter 5.

4.7 Conclusion

The results of these simulations illuminate several important points. First, the aggregate utility metric, in combination with synthetically-generated utility functions to augment the workload trace, are useful tools for evaluating and comparing scheduling algorithms of any variety.

Second, the notion that inaccurate requested runtimes are beneficial to scheduling is shown to be false in most cases. These experimental results are bolstered by the methodological criticism by Tsafrir and Feitelson [86]. The original result was in part an artifact of an overly “clean” model of user inaccuracy, which consisted of multiply-

ing all the actual runtimes by the same factor f to generate inaccurate ones. Of course with real user estimates, f will vary with each job. This methodological problem undermines the original result. Further discussion of this debate, and other related work on the problem of inaccurate user runtimes is found in Section 5.1.1.

Third, the GA scheduler was shown to provide not only more "attentive" preferential treatment to the highest-priority jobs, but did this while also improving the scheduling results for the majority of jobs. In other words, the GA scheduler did not serve the few at the expense of the many.

Parts of Chapters 3 and 4 are reprints of the material as it appears in the proceedings of International Symposium on High Performance and Distributed Computing (HPDC), 2007. Lee, Cynthia B.; Snavely, Allan E., 2007. [48] The dissertation author was the primary investigator and author of this paper.

Chapter 5

User-Provided Runtime Estimates

5.1 Introduction

5.1.1 Inaccuracy: Characterization, Effects, and Causes

As noted in Section 4.6, is well-documented that user-provided runtime estimates are inaccurate. Characterizations of this error in various real workload traces can be found in several classic and recent papers. Cirne and Berman [19] showed that in four different traces, 50 to 60% of jobs use less than 20% of their requested time. Ward, Mahood and West [94] report that jobs on a Cray T3E used on average only 29% of their requested time. Chiang, Arpaci-Dusseau and Vernon [14] studied the workload of a system where there is a 1-hour grace period before jobs are killed, but found that users still grossly overestimate their jobs' runtime, with 35% of jobs using less than 10% of their requested time (includes only jobs requesting more than one minute). Similar patterns are seen in other workload analyses [55, 79, 44]. Figure 5.1 shows the discrepancy between requested time and actual runtime jobs on SDSC's Blue Horizon system (where the survey described in the following sections was conducted (N=2,870)).

One might ask what impact user inaccuracy has on scheduler performance–why worry if user estimates are inaccurate? Indeed, Mu'alem and Feitelson have shown the surprising result that if workloads are modified by setting the requested times to

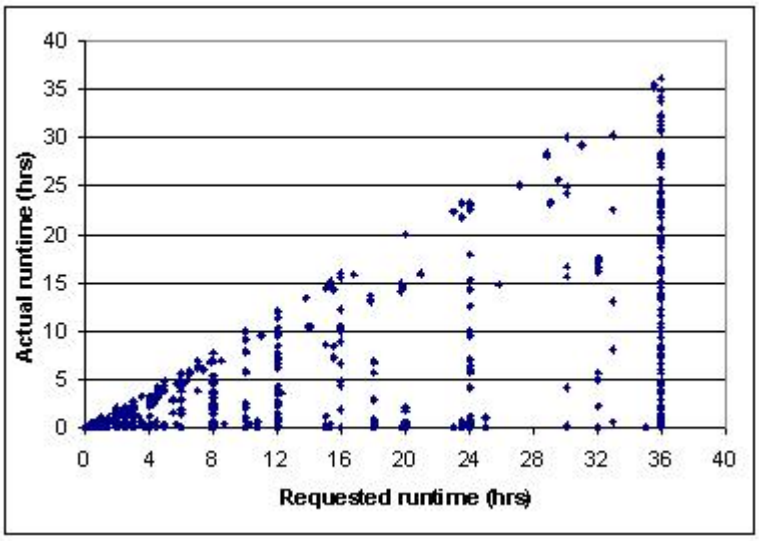

Figure 5.1: Comparison of actual runtime and requested runtime for all jobs on Blue Horizon during the survey period.

$R * actualruntime$, average slowdown for the EASY and conservative backfilling algorithms actually improves when $R = 2$ or $R = 4$, compared to $R = 1$ (total accuracy) [79, 98]. Similar results have been shown when R is a random number with uniform distribution between 1 and 2, or 1 and 4, etc. [55, 98].

But simply taking the accurate time and multiplying it by a factor does not mimic the "full badness of real user estimates" [55], leading some–including a co-author of the original paper–[86] to call this result into question. Using real user-provided times [55, 79], some scheduling algorithms did still perform equivalently or slightly better, compared to the same workload with completely accurate times. However, other algorithms experienced significant performance degradation as a result of user inaccuracy [14, 44]. The key point here is that Mu'alem and Feitelson's result only applies to the specific algorithms they studied, and it is necessary to re-prove (or disprove) their result for each new algorithm individually. Also, even for an algorithm such as conservative backfilling, which shows some mild improvement of average slowdown with inaccurate estimates, it is at the cost of less useful wait time guarantees at the time of job submis-

sion, and causing an increased tendency to favor small jobs over large jobs (which may or may not be desirable) [14, 98].

Many factors contribute to the inaccuracy of user estimates. All workloads show a significant portion of jobs that crash immediately upon loading. This is likely more indicative of users' difficulties with configuring their job to run correctly, than difficulties with providing accurate runtime estimate [55]. However, a job's runtime may also vary from run to run due to load conditions on the system. In an extreme example, Nitzberg and Jones [41] found that on an Origin system where different jobs on the same node share memory resources, job runtime varied 30% on a lightly loaded system, to 300% on a heavily loaded system.

Mu'alem and Feitelson [55] note that because many systems kill jobs after the estimated time has elapsed, users may be influenced to "pad" their estimates, to avoid any possibility of having their job killed. Users may also be insufficiently motivated to provide accurate runtime estimates. Many users are likely unaware of the potential benefits of providing an accurate request, such as higher probability of receiving quicker turnaround (because of an increased likelihood of backfilling), or this motivation may not be strong enough to elicit maximum accuracy.

5.1.2 Requested Runtimes vs. Estimated Runtimes

With regard to this padding, it is important to be precise about what users are typically asked to provide, which is a time after which they would be willing to have their jobs killed, and to distinguish this from the abstract notion of an estimate of their jobs' runtime. This leads us to prefer the term, requested runtime for the former, reserving the term estimated runtime for a best guess the user can make without any kill penalty (and possibly even with an incentive for accuracy).

5.1.3 The Padding Hypothesis

This experiment seeks to affirm or falsify the following hypothesis, which is implicitly or explicitly assumed in many of the papers in the literature:

The Padding Hypothesis: Users know the runtime of their jobs; the error observed in requested runtimes is a result of users adding padding to an accurate runtime estimate they have in mind.

This study tests this hypothesis by asking users of the Blue Horizon system [70] at the San Diego Supercomputer Center (SDSC) for a non-kill-time estimate of their jobs' runtime, and offering rewards for accuracy.

5.2 Survey Experiment Design

Blue Horizon was a 1,152-processor IBM SP2 system installed at the San Diego Supercomputer Center (SDSC) [70]. Resource management was handled by IBM's LoadLeveler software [37], augmented with a scheduling program called *Catalina* [97] that was developed in-house at SDSC. Users of the Blue Horizon system submit jobs by using the command *llsubmit*, passing as an argument the name of a file called the job script. The script contains vital job information such as the location and name of the executable, the number of nodes and processors required, and a requested runtime.

During the survey period, users were prompted for a non-kill-time estimate of their jobs' runtime by the llsubmit program, randomly one of every five times they submit. The question was asked at the moment of job submission because this is the most timely and realistic moment to measure the user's forecasting abilities. The traditional requested runtime is not modified in the job script, but merely reflect that value back to the user and the user is asked to reconsider it, with the assurance that their response in no way affects this job.

Users were notified of the study, by email and newsletter, a week prior to the start of the survey period. The notification included information about prizes to reward

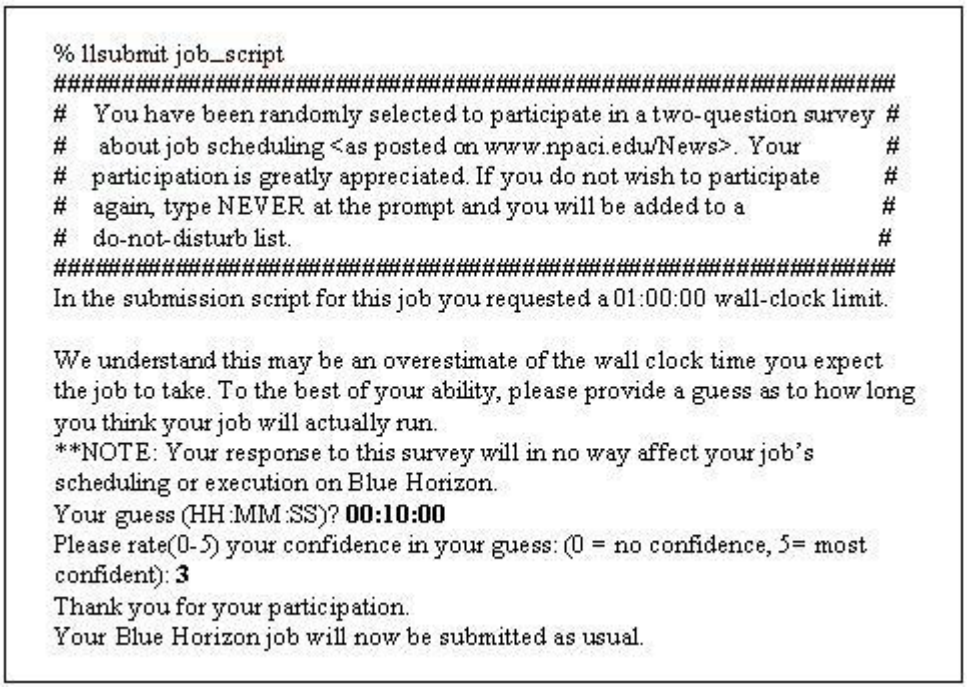

Figure 5.2: Sample runtime estimate survey and response

the most accurate users (with consideration given also to frequency of participation), specifically one MP3 player and several USB pen drives. The prizes were intended to provide a tangible motivation for accuracy and thus to elicit the most accurate estimates users are capable of providing.

The text of the survey is as follows. First, the user is reminded of the requested runtime (kill time) provided in their script. The user is then queried for a better estimate. Finally, the user is asked to rate their confidence in the new estimate they provided, on a scale from 0 to 5 (5 being the highest). This question was designed to test if users could self-identify as good or poor estimators. The survey does not provide default values. A sample of the survey output is shown in Figure 5.2.

5.3 Results

5.3.1 User Accuracy

Over the 9-week period of the survey, 2,870 jobs ran on the system (note that there were more job submissions than this, since many jobs are withdrawn while still waiting in the queue). Only one in five job submissions (selected pseudorandomly) triggered an attempt to administer the survey. Automated submissions (81) were not surveyed due to the lack of human respondent. At the request of system administrators who wished to minimize disturbance of users in the debugging process, jobs that requested less than 20 minutes of runtime (172) were excluded. There were 21 timeouts, where there was no response to the survey for more than 90 seconds; and 59 jobs submitted by the 11 users who decided not to take part in the survey. The number of completed surveys was 143.

Of these 143 surveys, 20 had actual runtimes that were equal to the requested runtime. This situation could possibly indicate that the user was precisely accurate or, more likely, that the scheduler killed the job once it reached its requested runtime. These survey entries were discarded since it was not possible to determine whether the job was completed or killed from the information collected. In 16 other responses, the estimated runtime given in response to the survey was higher than the original requested runtime in the script. Taken at face value, this means that upon further reflection, the user thought the job would need more time than they had requested for it, in which case the job is certain to be killed before completing. Some of these responses appeared to be garbage (*e.g.*, "99:99:99") from users who perhaps did not really want to participate in the study or just hoped a random response had some chance of winning a prize. In the analysis that follows, all of these higher responses were discarded, as well as a survey response indicating an expected runtime of 0 seconds. Henceforth, discussion of the survey results refers only to these 107 complete and valid surveys.

The 107 responses are divided into two categories. First, those where the esti-

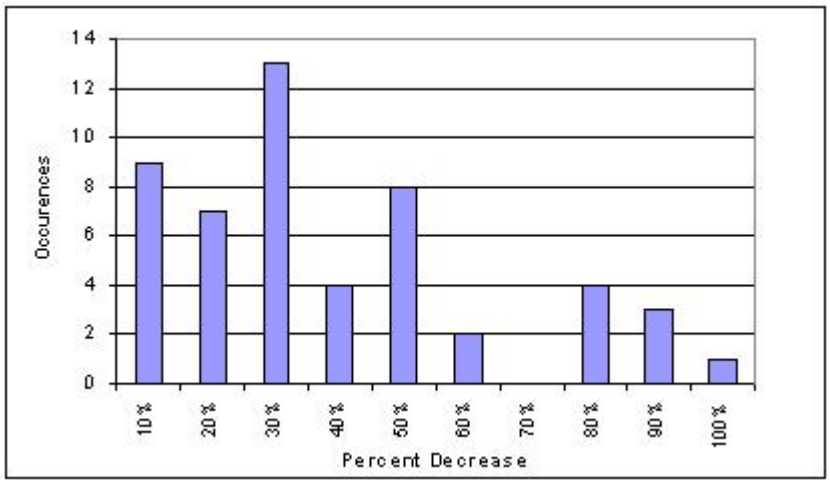

Figure 5.3: Histogram of percent decrease from the requested time to the estimate provided in response to the survey.

mated time (the survey response) was the same as the requested runtime (from the job script), numbering 56, and second, those where the user provided a reduced estimate in response to the survey, numbering 51. (See Figure 5.3. Note that Figure 5.3 includes only responses that were different from the requested time–56 responses had a 0% decrease. Categories represent a number of respondents up to the label, *e.g.*, 20% represents 7 responses that were between 10% (exclusive) and 20% (inclusive) decreased from the requested time in the script.) Of the 51 responses where users provided a tighter estimated runtime, users cut substantially–an average of 35%–from the requested time. The average *inaccuracy* in this group decreased from 68% to 60%. Inaccuracy means the percent of requested or estimated time that was unused, as given in the following formulas:

$$inaccuracy\ of\ requested\ time\ = \frac{|runtime_{requested} - runtime_{actual}|}{runtime_{requested}} \tag{5.1}$$

$$inaccuracy\ of\ estimated\ time\ = \frac{|runtime_{estimated} - runtime_{actual}|}{runtime_{estimated}} \tag{5.2}$$

Again, the requested time is from the job script and the estimated time is the survey response. So, for example, an estimated time inaccuracy of 68% means either that 32% of the estimated runtime was actually consumed by the running job, or that 168% of the estimated time was consumed. Note that it is possible for the actual runtime to exceed the estimated time, though that was very unusual in the survey. A requested time inaccuracy of 68% means that 32% of the estimated runtime was used (168% inaccuracy doesn't apply to requested times because overruns are not possible).

Including both categories of responses (same as requested time, and different), the overall the inaccuracy decreased from an average of 61% to 57%. Those users who did not tighten their estimate were notably more on target than those who did revise it; their initial inaccuracy was 55%. To fully understand the two metrics it is helpful to understand an example: A not atypical user requested their job to run for 120 minutes, revised (estimated) the runtime at 60 minutes in response to the survey, and the job actually ran for 50 seconds (!). In this example the user tightened their estimate by 50%. But the inaccuracy of the request is 99%, and the inaccuracy of the estimate is improved only 1% down to 98%. Intuitively, many users are substantially improving extreme overestimates, still without making the bounds very tight.

Figure 5.1 shows the comparison between the requested runtime in the script, and the actual runtime for *all* jobs on Blue Horizon during the weeks when the survey was being conducted. The results are similar to those seen in Figure 5.4, where we see the same information, but for only the jobs for which survey responses exist. The results in Figure 5.1 are also similar to those seen in the literature, in particular see [55]. Figure 5.5 shows the results if the estimate provided in the survey is used, instead of the requested runtime in the script. Note that no job's actual runtime can exceed the requested runtime, but because the survey responses were unconstrained in terms of being a kill time, the actual runtime can be either more or less than this estimate. This can be seen in the presence of points above the diagonal line. The great majority of survey responses were still overestimates of the actual runtime (below the diagonal).

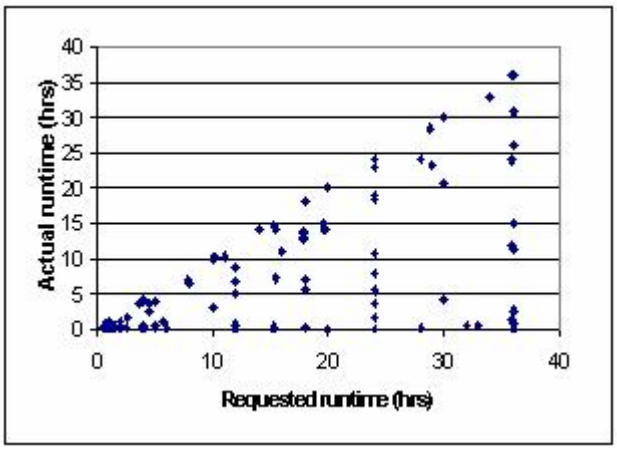

Figure 5.4: Comparison of actual runtime and requested runtime jobs in survey sample.

One could speculate that this may be a lingering tendency due to users having been conditioned to overestimate by system kill-time policies.

Little improvement can be seen in the pattern of error from Figure 5.4 to Figure 5.5. Notably, users still tend to round their times to 12, 24 and 36 hours in the survey, but not quite as heavily.

5.3.2 User Confidence

It is likely that even the most motivated of users will not always be able to provide an accurate runtime request or estimate. But it may be useful if users can at least self-identify when they are unsure of their forecast. In this study, users were asked to rate their confidence in the runtime estimate they provided in response to the survey on a scale from 0 (least confident) to 5 (most confident). Figure 5.6 shows the distribution of responses. In a majority (70%) of the responses, users rated themselves as most confident or very confident (5 or 4 rating) in the estimate. This is in spite of the fact that, overall, the accuracy of the requested runtimes and runtime estimates was poor (though typical, as observed in other workloads). It may be that users did not significantly adjust their forecasts of their jobs' runtime to account for possible crashes and other problems

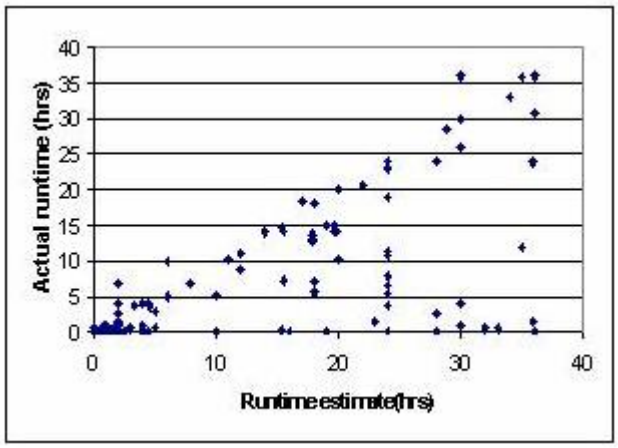

Figure 5.5: Comparison of actual runtime and requested runtime jobs in survey sample.

[51, 9].

The responses are divided by category–those users who provided a revised estimate in response to the survey, and those who reiterated the requested runtime in their script. Figure 5.7 shows that in 60% of responses that were the same as the requested runtime, users rated themselves as most confident (5), with another 22% rated very confident (4). No users rated themselves as low or very low confidence (1 or 0). In contrast, of those responses that were a different estimate (right graph of Figure 5.8), most users rated themselves somewhere in the middle (4 or 3).

Psychologists Kruger and Dunning have observed that people who are least knowledgeable or skilled in a subject area are more likely to overestimate their own abilities than those who are most knowledgeable or skilled. This result was shown in studies of students enrolled in university psychology classes, who were given tests in logic, grammar and humor [51]. Initially, the confidence score results in this experiment might seem to be an instance of the same phenomenon. Specifically, perhaps users reiterated the same requested runtime from the job script out of ignorance, and were then very self-confident, as predicted by Kruger and Dunning.

However, it appears that users who did not change in response to the survey, and had high confidence, did on average have more accurate estimates (as seen in Fig-

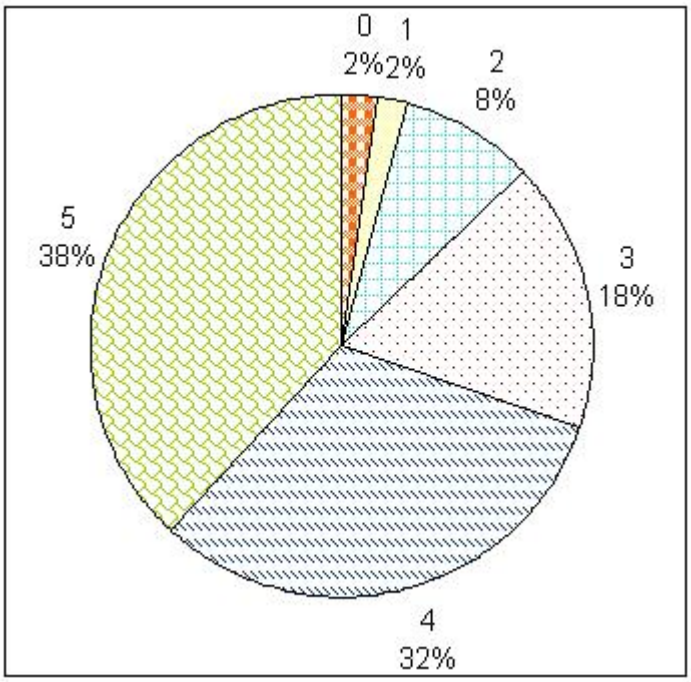

Figure 5.6: Distribution of user confidence scores.

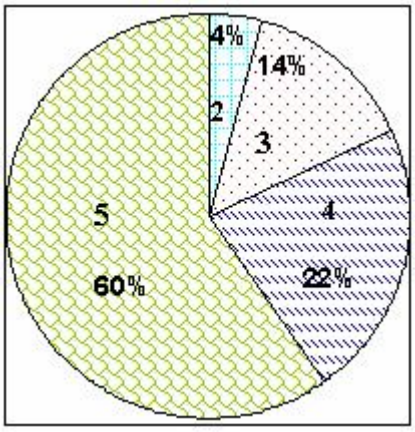

Figure 5.7: Distribution of user confidence scores, users who revised their estimate.

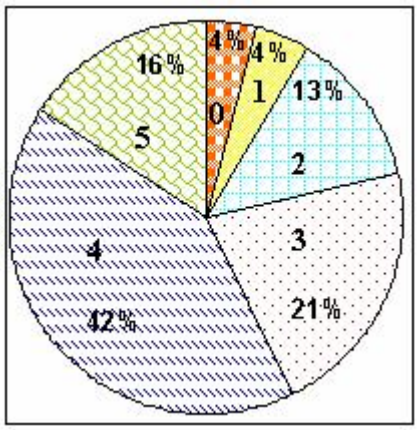

Figure 5.8: Distribution of user confidence scores, users who revised their estimate.

ure 5.9). For both changed and unchanged responses, there is a pattern of decreasing average inaccuracy as the confidence increases. There seems to be a strong correlation between these users' confidence and the accuracy of the estimates they gave in the survey, indicating that users are largely able to self-identify as accurate or inaccurate.

5.4 Other Approaches to Estimate Improvement

Asking the user for a more accurate time, as was done in this study, is not the only approach to mitigating inaccuracy. One suggestion is to weed out some inaccurate jobs through speculative runs, in order to detect jobs that immediately crash [44, 65]. Or, the system could generate its own estimates [84]. For jobs with a regular loop structure, this could be done via extrapolation from timings of the first few iterations [14]. Another proposal [81] is to charge users for the entire time they requested, not only the time they actually used. This idea, meant to discourage users from "padding" their estimates, may seem unfair to users because the fact that runtime may vary from run to run due to system load conditions necessitates a certain amount of padding.

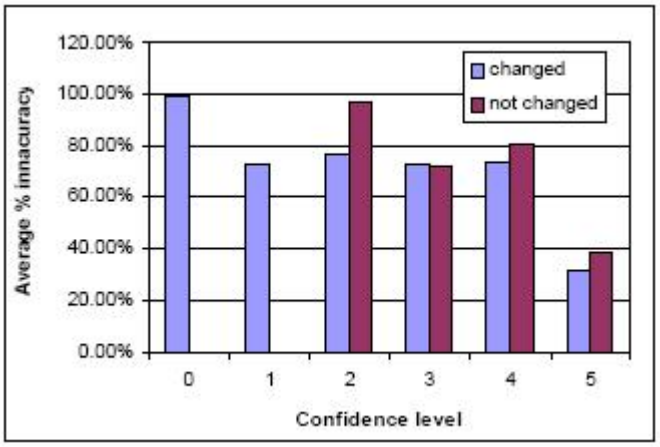

Figure 5.9: Average percent inaccuracy of survey responses.

5.5 Conclusion

Sound design of any software–including schedulers–must not ignore the human-computer interaction (HCI) component of the system. In the case of schedulers, a successful submit-time dialogue between the user and the scheduler lays the foundation for optimal scheduling outcomes. Asking one party in the dialogue to provide information that he or she can not provide, or does not wish to furnish, is a substantial barrier to successful dialogue.

Are users are capable of providing more accurate runtime estimates? To answer this question, users were surveyed at the time the job is submitted, asking them to provide the best estimate they can of their job's runtime, with the assurance that their job will not be killed after that amount of time has elapsed. The results of the survey demonstrate that some users will provide a substantially revised estimate but that the accuracy of their new estimates was only slightly better than their original requested runtime–from an average of 61% inaccurate to 57% inaccurate.

Thus the Padding Hypothesis is false–even absent a reason to pad, users do not demonstrate knowledge of a highly accurate estimate, so padding cannot be the sole cause of estimate inaccuracy. However, it is clearly one of several major causes, as

nearly half of users showed an awareness that they do pad by reducing their estimates, by an average of 35%. Another useful outcome of the survey was the observation that many users were able to correctly identify themselves as more or less accurate in their estimating than other users.

Parts of Chapter 5 are reprints of the material as it appears in the Proceedings of the 10th Job Scheduling Strategies for Parallel Processing, 2004. Lee, Cynthia B.; Schwartzman, Yael; Hardy, Jennifer; Snavely, Allan, 2004. [46] The dissertation author was the primary investigator and author of this paper.

Parts of Chapters 2 and 5 are reprints of the material as it appears in International Journal of High Performance Computing Applications, 2006. Lee, Cynthia B.; Snavely, Allan E., 2006. [47] The dissertation author was the primary investigator and author of this paper.

Chapter 6

Increasing Schedule Flexibility Using Checkpointing

6.1 Introduction

Decades of work by numerous researchers, including work presented here, has addressed the challenge of how to squeeze more utilization and shorter job wait times from supercomputers, all while operating within a rigid set of assumptions and constraints. Among these assumptions and constraints are that jobs have fixed width (number of processors) and length (in time), that we know the job's length when it is submitted, that jobs should not share processors, and that the scheduling decision is irrevocable.

Each of these was adopted by the research community with good reason and each remains useful. Nevertheless, valuable innovations have been brought about by challenging many of these assumptions. Questioning the fixed length and width of jobs yielded moldable job scheduling [80, 21, 18, 20, 50]. Questioning that jobs should not share processors yielded symbiotic scheduling [76, 77, 91, 93, 92]. Work presented in Chapter 5 examines the problem of not knowing the length of the job when it is submitted.

In this chapter, the assumption that scheduling decisions are irrevocable is ques-

tioned.

6.2 Checkpointing

Checkpointing is a means of saving interim results of a computation so that not all work is lost in the case of job failure. Typically, checkpointing is done at static intervals throughout the duration of job execution. In the event of an unexpected failure, such as hardware or system crash, the job may be restarted from the information contained in the most recent checkpoint. Only the work between that checkpoint and the failure is lost, thus saving computation time compared to starting from the beginning. This savings must be weighted against the overhead of performing the checkpoints. In the case of parallel jobs, the checkpoint must be performed on each process in a coordinated fashion [5, 27, 73, 64].

There are two main approaches to checkpointing: system-level checkpointing and user-level checkpointing.

System-level checkpointing saves the state of the entire process, including all memory and processor state. It is done at the operating system level and can be performed on any process at any time [67, 72]. Its advantages are that no customization of the checkpointing procedure to the application is required, and no cooperation with or modification of the application is required. In fact, the operating system could unilaterally decide to checkpoint any application, and restart it.

Precisely because it is not coordinated with the application, this type of checkpointing must err on the side of safety and save the entire state, which may be wasteful in terms of the amount of the data saved and, consequently, the amount of time needed to perform the checkpoint.

User-level checkpointing is potentially less wasteful because it may be customized to the application [1]. This customization could include only saving those data structures or parts of data structures that are of interest. Often, checkpointing is performed at times when the computation has reached a natural break where less data

needs to be saved, such as iterations of an outer loop of a computation. This results in lower overhead associated with performing the checkpoint, both in the amount of data to save and time spent checkpointing. Loop-based checkpoints may be only approximately periodic but for simplicity are described with a static interval length, as in [64].

The primary disadvantage of user-level checkpointing is that it requires effort to add this functionality to applications. Users are seldom the authors of their software, and may lack the necessary expertise to edit the code. In the case of software the user has licensed from a vendor, it may not be possible to make modifications. In any case, doing so would require significant expense of engineering labor.

6.3 Using Checkpoints in Scheduling

As discussed above, checkpointing is typically used as insurance against unexpected failures such as hardware failures. If jobs are already incurring the overheard of performing periodic checkpoints for this purpose, it may make sense for the scheduler to leverage this in order to improve system-wide scheduler performance. Schedulers could rely on checkpoints in order to strategically relax the classic scheduling constraint that a decision to start running a job is irrevocable. By opening up the possibility of intentionally causing jobs to fail (terminating them), the scheduler greatly increases its flexibility in decision-making.

A scenario where this might be a beneficial strategy would be the following. All jobs in the queue and currently running have low urgency (alternatively, have utility functions with near-zero slope), and all processors on the system are being utilized. A very high urgency job arrives. Under the standard assumptions, the scheduler is not permitted to consider terminating a currently running low urgency job in order to accommodate the high urgency one. Although the delay would result in some loss in value for the terminated job (it would have to be restarted later), the decision may be result in higher aggregate utility because it would be offset by the much greater utility earned by running the high urgency job immediately.

There are many factors to consider in making such a decision. What is the difference in value and urgency between the affected jobs? How much time would they need to complete if not interrupted? How long would they be delayed if interrupted? Checkpoints are relevant in this situation because the penalty of terminating a job will be mitigated if it would lose a only small amount of work and not have to be restarted from the beginning.

A scheduler could initiate a system-level checkpoint of a job it wants to terminate, just prior to terminating it. This would result in almost no work lost. However, the overhead of performing a system-level checkpoint is high. If the scheduler knows that the job performs its own system-level or user-level checkpoints at regular intervals, and knows little time has passed since the most recent checkpoint, it could terminate the job and plan on relying on that checkpoint information. Even knowledge of just the checkpointing interval would allow the scheduler to determine the probabilities of various amounts of work lost to the job.

In order to evaluate the effectiveness of schedulers employing such strategies, it is necessary to have a workload for simulation which includes information about users' checkpointing habits. This chapter describes a survey of supercomputer users designed to collect information about their habits and annotate the workload trace including their jobs with this information. This will enable investigations into the checkpointing question that are grounded in real-world data.

6.4 Survey Experiment Design

This survey follows a similar structure as those described in Section 5.2 and Section 2.3. The survey was conducted over approximately three weeks in Spring of 2008 on the Blue Horizon system installed at the San Diego Supercomputer Center (SDSC) [70]. Users of the Blue Horizon system submit jobs by using the command *llsubmit*, passing as an argument the name of a file called the job script. The script contains vital job information such as the location and name of the executable, the number of nodes

and processors required, and a requested runtime.

During the survey period, users were prompted to respond to a survey question by the llsubmit program. Unlike the two previous surveys, where users were surveyed only one of every five times they submit (chosen at random), this survey was administered on every submission from every user. The only exceptions were submissions from users on the opt-out list. Users were notified of the study, by email and newsletter, a week prior to the start of the survey period. They could opt out at that time by email, or at any time when the survey was administered.

A sample of the survey output is shown in Figure 6.1. Following a brief consent message (not shown in Figure 6.1), users are asked to respond to a single multiple-choice question about whether or not the job being submitted will perform checkpointing, including restart capability. The "no" response (checkpointing will not be done) is divided into several categories according to various explanations for why checkpointing might not be done. An "other" response choice was provided for users who felt none of the provided explanations were applicable. Users were constrained to providing exactly one response; thus users who felt that more than one explanation was applicable would need to select only the most applicable explanation. In the case of a "yes" response (checkpointing will be done), users were then asked to provide the frequency interval of the checkpointing, in minutes. For both the initial question, and follow-up, if any, an empty or invalid response would result in the user being asked to try again. Users always had the option to decline to participate by selecting that choice from the provided options. This results not only in exiting the current instance of the survey, but in being added to a permanent "do-not-disturb" list.

```
SURVEY QUESTION:
   Will the job you are submitting perform checkpointing and have
restart capability?

   0) YES  [you'll be prompted to enter how often]
   1) NO,  because it's a debug/trial run
   2) NO,  because it will probably end successfully
   3) NO,  because checkpointing isn't available for this program
   4) NO,  because checkpointing incurs a slowdown
   5) Other [you'll be prompted to enter an explanation]
   6) Decline to participate in this survey
   Please enter your choice [0-6]
3
```

Figure 6.1: Sample checkpoint survey and response

6.5 Results

6.5.1 Prevalence of Checkpointing

There were 1,356 responses to the survey, which ran from April 11, 2008 to April 29, 2008. Table 6.1 summarizes the responses. Only 16% of jobs were reported to be configured to perform checkpointing, while 80% were reported to be without checkpointing, and 4% of survey responses were "opt-out." Note that data is lacking for substantially more than 4% of the total number of jobs submitted during the time the survey was being conducted, due to the "opt-out" being permanent for that user. The fact of only one user having selected "other" (choice 5) provides some assurance that the selection of explanations for why checkpointing was not being performed provided adequate coverage of users' circumstances.

Among only the "no" responses, half (50%) indicate that the job was a debug/trial run (choice 1). Another 39% believed that job failure was unlikely (choice 2), and the rest cited logistical and performance problems (choices 4 and 5). Debug/trial run jobs are smaller on average than other jobs in terms of processors and runtime. Thus the large percentage they represent here, in terms of number of jobs surveyed, greatly overstates their contribution to the overall consumption of resources on the HPC system. In fact, although 40% of jobs were debug/trial run jobs, only 3% of the total node-hours

Table 6.1: Summary of checkpoint survey results

Choice	Number of Responses	Percent of Responses	Choice Meaning
0	219	16%	*YES: performs checkpointing*
1	540	40%	*NO: debug/trial run*
2	426	31%	*NO: will probably end successfully*
3	29	2%	*NO: isn't available for this program*
4	94	7%	*NO: incurs a slowdown*
5	1	0%	*Other*
6	56	4%	*Decline to participate*
Total	1365	100%	

of all jobs surveyed were consumed by debug/trial run jobs.

Importantly for the purposes of this work, although only 16% of the jobs reported using checkpointing, these jobs comprise a majority of the surveyed workload, in terms of node-hours (57%).

Table 6.2 shows the percent of the total node-hours of surveyed jobs that is represented by each response category. Node-hours reported in Table 6.2 use the actual runtime consumed by the job, not the requested time.

6.5.2 Frequency of Checkpointing

Of the 16% of jobs (57% of the workload by node-hour) reported by users to use checkpointing, we would like to know how often the checkpoints are taken. Users who reported using checkpointing received a follow-up question asking them to report this interval, in minutes. Figure 6.2 shows the cumulative distribution function (CDF) of reported checkpoint intervals.

The median checkpoint interval was 15 minutes. The most commonly reported intervals were 5 minutes (36 responses), 10 minutes (38 responses) and 30 minutes (34 responses), collectively accounting for half of the 219 survey responses. Another way to consider the checkpoint interval is as a fraction of the runtime of the job, indicating how many checkpoints are taken per run. Figure 6.3 shows a CDF of the ratios of each

Table 6.2: Breakdown of workload by checkpoint survey results

Choice	Number of Responses	Node-Hours of Jobs	Percent of Total Node-Hours	Choice Meaning
0	219	37,397.65	57%	*YES: performs checkpointing*
1	540	2173.71	3%	*NO: debug/ trial run*
2	426	18,668.29	28%	*NO: will probably end successfully*
3	29	2,086.38	3%	*NO: isn't available for this program*
4	94	255.9	<1%	*NO: incurs a slowdown*
5	1	35.89	0%	*Other*
6	56	5335.9	8%	*Decline to participate*
Total	1365	65953.72	100%	

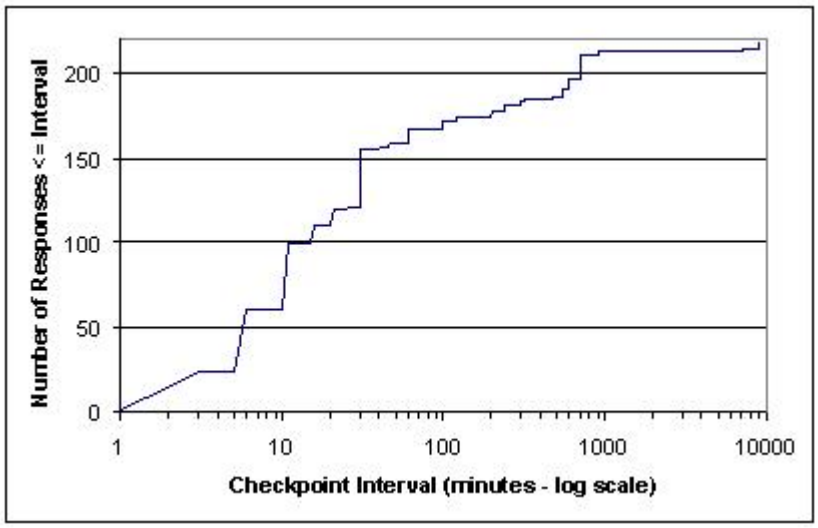

Figure 6.2: Cumulative distribution of users' reported checkpoint intervals.

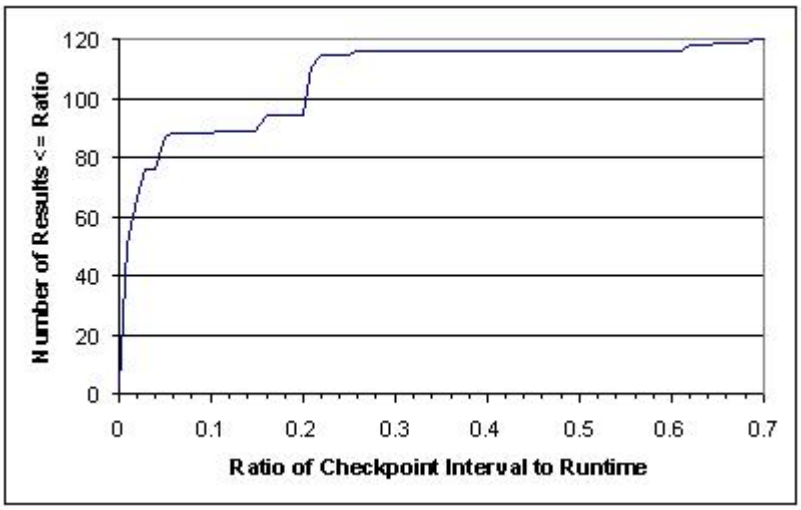

Figure 6.3: Cumulative distribution of users' reported checkpoint intervals.

job's reported checkpoint interval to its actual runtime. (Runtimes are available for 120 of the 219 jobs for which users reported checkpoint intervals.) A majority (67 of 120) of the checkpoint intervals are less than or equal to 1/50th of the runtime of the job, or about 50 checkpoints per run. A full 96% (115 of 120) have ratios of less than 1/4th.

6.5.3 Representativeness of the Survey Respondents

There were 4,203 jobs submitted during the the three-week survey period. Of these, survey results (not including responding by selecting "opt out") were collected for 1,010 jobs. Note that the distribution of in-survey and out-of-survey jobs is not uniform across the two weeks of the survey. In the first two days of the survey, survey responses were collected for the majority of jobs, after which the percent declined as users who had already responded–some several times–opted out. The number of processors and minutes of runtime of each job are used to verify representativeness of the survey sample. The average number of processors in the overall job mix is 7.0 (standard deviation of 17.7), compared to an average of 8.5 (standard deviation of 21.4) for jobs with survey responses. The average runtime in the overall job mix is 208 minutes

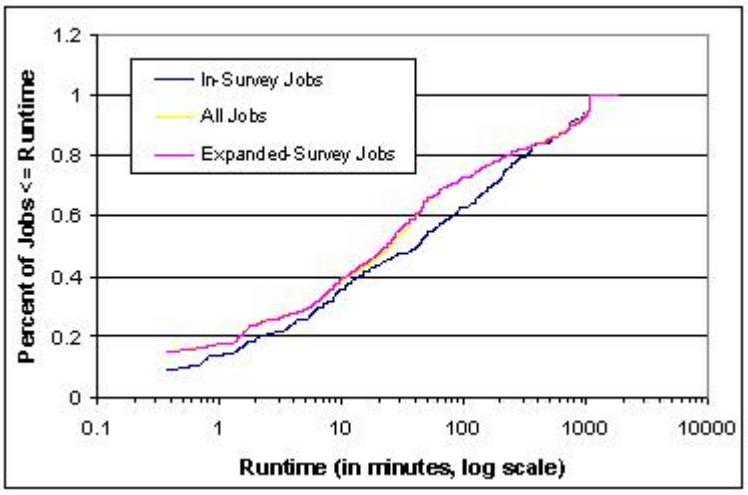

Figure 6.4: Cumulative distribution of jobs' runtimes.

(standard deviation of 412), compared to 212 for in-survey jobs (standard deviation of 317). Figures 6.4 and 6.5 compare the mix of jobs in the survey with the overall mix of jobs, in terms of runtime and number of processors, respectively. Figure 6.5 shows that 1-processor jobs are under-represented in the survey, while 2- through 8-processor jobs are over-represented in the survey. By runtime, the survey respondents are well matched with the overall job mix.

6.5.4 Summary of Results

The findings of the survey are that 16% of jobs in the survey are configured to perform checkpointing, comprising most of the total node-hours of jobs in the survey (57%). Of jobs in the survey that perform checkpointing, most do so with *fine granularity*. This is true whether one defines *fine granularity* to mean a frequency of at least every 15 minutes, or to mean at intervals that are at most 1/50th of the job's runtime.

These findings constitute a baseline level of participation in checkpointing. Another approximately 30% of the surveyed workload (38% of surveyed jobs, or 29% in terms of node-hours) could possibly be persuaded to implement checkpointing if it were

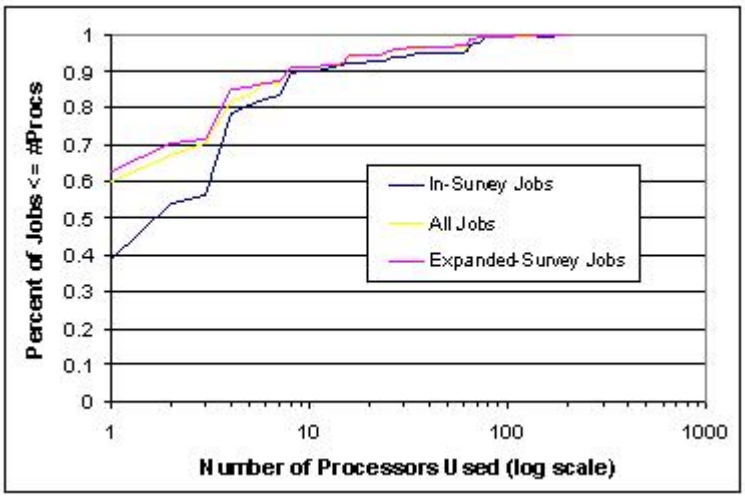

Figure 6.5: Cumulative distribution of jobs' processor counts.

strongly encouraged or required by the supercomputer center. These are the jobs that do not currently perform checkpointing because they "will probably end successfully," or because it "incurs a slowdown."

The greater the prevalence of checkpointing and the finer the granularity of checkpointing, the more conducive a workload is to flexible scheduling via selective termination of running jobs. Thus these results indicate that such an approach to scheduling could be promising. The next sections will discuss using the full workload trace of the system from the period when the survey was conducted to perform simulation trials of such a scheduler.

6.6 Workloads with Checkpoint Information

In order to more fully explore the results in Section 6.5, workload traces augmented with information about the checkpointing behavior each job were produced. These augmented workloads will enable simulation-based scheduler experiments that are grounded in real-world data.

For more information on the contents and format of existing workload traces, to

which this work adds, see the description of the *Standard Workload Format* in Section 2.6.1. The information added to the trace this time will be: first, does the job performs checkpointing, and second, if yes, what is the checkpoint interval. Unlike the workload trace augmentation (with utility functions) described in Section 2.6.1, the workloads produced here attempt to be historically accurate.

Two workloads were produced. The first simply uses the actual survey results for each job in question. For jobs where no data exists (no survey response exists), the conservative assumption was made, *i.e.,* that they do not perform checkpointing.

The second workload attempts to be as historically correct as possible, while also filling in those gaps in the data where reasonable inferences can be made.

What are these inferences? In order to produce the most complete augmented workload data set possible, the checkpoint survey was administered every time a batch job was submitted to the system, excepting users who had previously chosen to opt out. Users very often responded with the same information each time because they were, for example, resubmitting the same code with different input data, or rerunning a Monte Carlo simulation[1].

Consider those users who responded to the survey repeatedly, and consistently, before eventually opting out. It is likely that had additional instances of the survey not been suppressed, their responses would have continued to be be consistent with earlier responses. For other users, variability in their responses may coincide with observable differences between the jobs, and their responses become consistent once these differences are accounted for. Again, one can guess that responses following the opt-out would have been consistent with earlier matching jobs' responses. This is the rationale behind constructing an expanded workload.

The expanded workload was created by pooling all survey responses of like user and number of processors, then selecting randomly from that pool to fill in missing data. In the case of jobs that lack matching user and processor count data, the conservative

[1] Monte Carlo simulations involve many repeated trials using different pseudorandomly generated configurations each time [87].

assumption is made–that they were not performing checkpointing.

The first workload represents a lower bound on the amount of checkpointing. The second workload represents a better guess at the actual amount of checkpointing. The second could overstate the amount of checkpointing if inferences were incorrect. It could also understate the amount of checkpointing if jobs for which there was not adequate data for inferences do in fact perform checkpointing.

6.7 A Checkpoint-Aware Scheduler

6.7.1 Implementation

This section introduces a novel checkpoint-aware scheduler. The scheduler is a straightforward extension of the GA scheduler described in Chapter 3, and illustrates the flexibility and extensibility of the GA scheduler. Added to the space of possible schedules to be considered by the GA algorithm are schedules where currently running jobs have been removed, to be restarted at a later time. The GA algorithm will evaluate the fitness of these schedules alongside the traditional scheduling options.

To implement this feature, both the GA scheduler (Chapter 3) and the scheduler simulator framework (Section 3.3) required modification.

In the GA scheduler, possible schedules are hypothetically considered by simulating removing currently running jobs from their assigned processors and requeuing them as seemingly new jobs. Thus running jobs recompete for their resources at each scheduling round.

When simulating the requeuing of a job inside the GA scheduler, the requested time of the job must be adjusted to account for three factors:

1. The amount of time the job has already been executing.

2. The amount of work lost, *i.e.,* the time elapsed between the time of the last checkpoint and when the job was terminated.

3. The overhead involved in restarting from the checkpoint.

If the scheduler decides to schedule a running job immediately, it is in effect deciding to not terminate it in the first place. Thus no adjustment is required. In cases where the scheduler supplants a previously running (requeued) job with another job, the requeued job's requested time should be adjusted to reflect the work completed, lost work and the restart overhead. The work completed is the amount of time the job had executed prior to requeuing. The adjustment for the amount of lost work is dictated by the checkpointing latency that was reported for that job in the survey (or, in the case of the expanded workload, the guessed latency; see Section 6.6). Models for estimating or predicting restart overheard have been studying elsewhere [64]; here the cost is assumed to be proportional to the processor count.

In the scheduler simulator framework, the interface between the scheduler and the framework had to be modified to allow the scheduler to communicate the fact that it had chosen to terminate and requeue jobs. The simulator framework then frees the processors in question. Recall that the simulator framework operates as a discrete event simulator, and that an event has been enqueued for each running job, representing a timer to mark the expiration of the job's actual runtime. This is separate from the expiration of the job's requested time, as jobs typically spontaneously terminate well before the requested time has expired. For jobs that the scheduler decides to terminate and requeue, these timer events must be invalidated in the discrete event simulator's event queue. New timer events will be enqueued when the job is subsequently restarted. These timer events will reflect the lost work and checkpoint restart overhead adjustments.

The GA scheduler only makes adjustments to the requested time, whereas the simulator framework makes adjustments to both runtime and requested time. This is because the actual runtime of a job is never visible to the scheduler during simulations, replicating real-life scheduling conditions.

Left unchecked, the checkpoint-aware GA scheduler suffered from excessive

requeuing and restarting of jobs, resulting in starvation of jobs and lack of progress in the simulator. Three damping measures are used to prevent thrashing. First, after a requeued job is restarted, it may not be requeued again (a limit of one *bump* per job). Second, when assessing the fitness of possible schedules and calculating the utility of a job's prospective turnaround time, the utility of currently running jobs is increased to give them an advantage and make them less likely to be bumped. The amount of the *boost* is equal to the starting value of the job. Third, is the *airline* policy, which takes inspiration from the policy of many airlines to bump passengers but then guarantee that they will be seated on the very next flight. Recall that the GA scheduler evolves an ordering of jobs in the queue (see Section 3.2). Under the airline policy, bumped jobs have a reserved place at the front of the queue, while GA may only evolve an ordering for the remaining jobs. Limiting to a single bump is always done, while the boosting and airline options may be added separately or together.

6.7.2 Policy Considerations

The requested runtime of jobs that are terminated by the scheduler and restarted from checkpoint is adjusted for work already completed, for time lost, and for restart overhead. In the simulation, restart overhead is estimated and for the purposes of simulation, it is assumed that the estimates are correct. This assumption could prove problematic in a production environment because the overhead cost to restart from the checkpoint may vary widely from job to job. Overhead of user-level checkpointing is particularly difficult to predict, and can vary from near-instantaneous to as long as a system-level checkpoint that saves the entire processor and memory state. Unless the scheduler has information about checkpoint overhead that is specific to a particular job, it is forced to guess.

For jobs where the requested time closely approximates the natural (uninterrupted) runtime, underestimating the restart overhead could cause a restarted job to overrun its requested time and thus be killed by the scheduler (in this case, perma-

nently). On the other hand, an overly generous requested time policy that essentially allows restarted jobs to run forever would undermine the meaning of the requested time. The implications of this include potentially motivating users to deceitfully set the requested time to be too small, hoping the job will be bumped and then face no sanction for underestimating but could benefit by fitting into short backfill spaces.

Another policy consideration is that to the extent that checkpointing is useful to system-wide optimization of aggregate utility, administrators might consider providing incentives to users to perform user-level checkpointing.

6.7.3 Checkpoint-Aware Scheduler Simulations

In 2007, DataStar had 272 nodes, each node consisting of either 8 or 32 processors, for a total of 2,344 processors [71]. The effective machine size for batch queue jobs is reduced by a partition set aside for interactive jobs. Interactive jobs bypass the batch queue submission program used to administer the checkpointing survey and are not included in the workload. For simplicity, in these simulations the machine is modeled as having 260 homogenous nodes.

Figure 6.6 shows the results of running the checkpoint-aware GA scheduler on the checkpoint workload. Each value shown in the graph is the average of 6 simulation trials (utility functions were added to the workload 6 different times). Performance of the standard (non-checkpoint-aware) version of GA, as well as classic scheduling algorithms, are shown for reference. These algorithms do not consider checkpointing nor allow for interruption of jobs.

The checkpoint-aware GA scheduler clearly benefits from measures to curb the tendency to starve jobs via excessive thrashing. Performance is highest–16% improvement over standard GA and 23% improvement over CONS) in the version with the most aggressive damping (a single bump limit, airline policy and boosting; please refer to Section 6.7.1 for a detailed description of the damping methods).

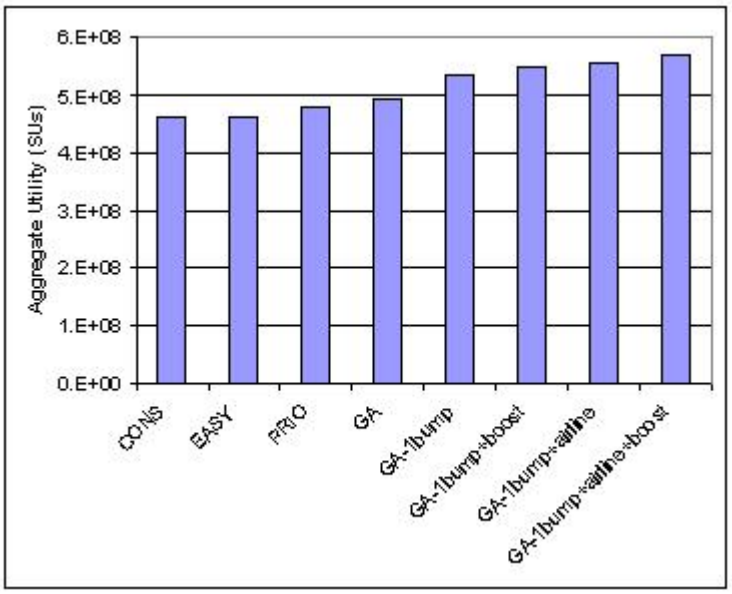

Figure 6.6: Performance of the checkpoint-aware GA scheduler compared to standard algorithms, according to aggregate utility metric.

6.8 Conclusion

A survey of supercomputer users shows that 16% of jobs are configured to perform checkpointing, comprising most of the total node-hours (57%). Of jobs in the survey that perform checkpointing, most do so relatively frequently (at least every 15 minutes, or at intervals at most 1/50th of the job's runtime). Of users who report not using checkpointing in their jobs, about half say it is due to the job being a debug run or the job will probably end successfully. These jobs are very small, comprising just 3% of the total node-hours, as expected for debug runs.

The greater the prevalence of checkpointing and the finer the granularity of checkpointing, the more conducive a workload is to flexible scheduling via selective termination of running jobs. The GA scheduler presented in Chapter 3 was modified to implement this flexibility by allowing the GA scheduler to access the checkpoint survey data for each job. The checkpoint-aware GA scheduler may opt to terminate and requeue running jobs, relying on the most recent checkpoint for later restart.

With these changes, aggregate utility performance was improved by 15% over standard GA, and by 34% over the classic EASY algorithm. This includes penalties for work lost since the last checkpoint and overhead involved in restart. Damping measures are required to prevent excessive thrashing (terminating and requeuing) by the GA scheduler when it is allowed the freedom of terminating jobs. Of the tested damping policies, the more aggressive the damping, the better the performance.

Chapter 7

Conclusion

7.1 Summary

This dissertation identifies and addresses several key issues in the scheduling of scientific computing applications on high-performance computing (HPC) systems, also known as supercomputers. HPC workloads consist of hundreds of jobs each day, and each job has unique resource requirements, including number of processors and runtime. How to best organize the running of these jobs is a mature but active research area. The problem is to map jobs submitted by users onto blocks of time on subsets of the systems' processors. This domain of scheduling is known as parallel job scheduling, or parallel batch scheduling.

If the scheduling task is viewed as, at its core, the task of maximizing satisfaction of the users, then an important prerequisite to scheduling is having information about user satisfaction. In Chapter 2, the desires of the user population were quantified and represented as a utility function, *u(t)*, whose independent variable *t* is turnaround time. A survey-based study of users of a supercomputer system was conducted, showing that the shape of the utility functions was heterogeneous across users and jobs. A model for synthetically generating utility functions to accompany existing workload logs was presented. The model includes three main model subtypes: expected linear decay, expected

exponential decay and step. Applying the synthetic generation model to a standard workload from the Parallel Workloads Archive allowed for evaluation of several classic scheduling algorithms according to the metric *aggregate utility* (Chapter 4). These results include finding that Priority-FIFO, using a crude approximation of user utility, outperforms Conservative and EASY flavors of backfilling (that do not consider utility) on the aggregate utility metric by 18% average.

In Chapter 3, a scheduler optimizing the aggregate utility of all users, using a genetic algorithm heuristic, is demonstrated. Taking a priority queue ordering (a permutation of the list of queued jobs) as an individual in a population, the GA scheduler evolves a queue ordering with the goal of optimizing the aggregate utility of the schedule that results when the fittest individual is input into the EASY scheduling algorithm to produce a scheduling decision. Fitness is defined as an estimate of the aggregate utility of scheduling jobs in that order. The results in Chapter 4 show that the GA scheduler computes scheduling decisions well within the real-time performance constraints of production supercomputers, and outperforms classic scheduling algorithms according a variety of schedule quality metrics including aggregate utility. As noted above, Priority-FIFO outperforms Conservative and EASY flavors of backfilling by 18% average under realistic conditions. The GA approach outperforms Priority FIFO by an additional 13% average.

According to the study in chapter 2, users are willing and able to provide much richer information about their utility than is asked of them by current scheduling systems. In Chapter 5, we find an example of the reverse, namely, scheduling systems universally asking users for data they are unable or unwilling to provide in an accurate manner. It is widely known that users' jobs often use much less time than was requested. The results in Chapter 5 indicate that users do not provide accurate runtime estimates, even absent incentives to add "padding" to their estimates, thus they do not appear able to reliably and accurately provide this information. This suggests the need to design scheduling algorithms so that they do not require accuracy, or find sources of information other than the users themselves (e.g., empirical observation). On the other hand,

users' self-identification of their confidence in their estimates was somewhat predictive of estimate accuracy. This suggests the possibility of designing a tiered system in which some user runtime requests are considered more reliable than others by the scheduling system.

Finally, Chapter 6 raises the possibility of lifting one of the defining constraints of the parallel job scheduling problem–the non-preemptability of running jobs. In order to do this without needing to start the job from scratch, the state of the job needs to be saved in a *checkpoint*. Many users already save checkpoints periodically for other reasons. To investigate the feasibility of relying on these existing checkpoints for preemptive scheduling, it is necessary to know how widespread the practice of checkpointing is, and, when used, how frequently checkpoints are saved. The prevalence and frequency of checkpointing on a major supercomputer system is explored via a survey of users. A workload augmented with this data is used as input to a modified version of the scheduler presented in Chapter 3, that is able to terminate and re-queue running jobs. This checkpoint-aware scheduler balances the overhead of lost work since the checkpoint and the cost of restarting, with the benefits of allowing more urgent jobs potentially immediate access to the resource.

Associated with these contributions was the production of several software and workload artifacts of use to the scheduling research community at large; indeed, some have already been distributed and used by colleagues. These include the following:

1. Script for augmenting workloads with synthetically-generated utility functions
2. Canonical workloads augmented with synthetically generated utility functions (a consequence of 1)
3. Modular scheduler simulator framework compatible with the Standard Workload Format (SWF) file format (python language)
4. Genetic Algorithm based heuristic scheduler (python language, compatible with 3)

5. Checkpoint-aware scheduler (python language, compatible with 3)

6. SWF supercomputer workload for the DataStar system, years 2006 through 2008, prepared for inclusion in the Parallel Workloads Archive (joint effort with Dan Tsafrir), including script for converting IBM LoadLeveler proprietary log format to SWF

7. Subset of the DataStar workload (6) with associated user checkpoint survey responses

8. Subset of the DataStar workload (6) with associated user checkpoint survey responses, and with extrapolation over some jobs not covered in the survey

7.2 Future Work

7.2.1 Genetic Algorithm Scheduler

Offline Optimization Between Iterations

One open question with the GA scheduler is how much efficiency (both in terms of runtime, and quality of the result) is lost due to restarting its optimization from scratch in each scheduling iteration.

The software model employed by the GA scheduler is shared with other supercomputer schedulers, namely the scheduler is invoked each time one of the following three events occurs: a running job terminates, a running job's requested time expires, or a new job is enqueued. Schedulers are stateless between these decision-making iterations. The explanation for this software model is that decision-making cannot begin until all the relevant facts are known, else work would need to be discarded when the facts change due to one of the three events. In the case of the GA scheduler, it may be that previous optimization work retains some value, even in the face of some perturbation to the queue and system conditions, such as the arrival of a new job.

Carrying a small amount of state between scheduling iterations could preserve some of value of previous work. The scheduler could remember the best individual from the previous iteration, in order to minimally modify and re-introduce it into the next iteration of optimization.

Extending this idea, the scheduler could be run continuously between iterations, further optimizing based on the last known state of the system. When called in connection with one of the three scheduling events, it would incorporate the changes in state into this offline-optimized schedule, and do a fixed amount of further optimization before returning. After returning, it would continue by again optimizing based on the last known state. This would possibly increase the quality of the schedule obtained from the fixed amount of immediate optimization in each iteration.

Incorporation of Other Independent Variables into Utility Function

The flexibility of the GA scheduler lends itself to extension to consideration of a variety of concerns that users and site administrators might have. Currently, aggregate utility is only a function of turnaround time of the job. Other factors that could be considered include fairness (minimize variance of turnaround time or expansion factor), predictability (minimize difference between a turnaround time presented to the user when the job is enqueued and the actual turnaround time), or administrator preference for large jobs (suggests a weighting of the aggregate utility).

7.2.2 Job Utility Functions

User Interface Study

In the utility function survey presented in Chapter 2, utility functions were solicited a single point at a time by presenting hypothetical scenarios in interactive prose format. Rethinking this user interface would be an important part of turning the GA scheduler into a production-ready system. Some alternatives include providing the utility function as numeric *(time, value)* pairs in the job script (just as they are stored in

the extended *SWF* format), a graphical user interface (GUI) that allows users to click to specify each point on the curve, or allowing users to build up their utility functions over time by specifying a single point each time as in the survey. These should be evaluated through ethnographic user interface studies.

7.2.3 Checkpointing

Ethnographic Study

The survey presented here provides useful data on current habits, but would benefit from more insight into the circumstances and causes of the reported behavior. An ethnographic study into the details of users' checkpoint habits, such as the threshold at which users determine overhead to be too costly, and whether they possess the programming skills and legal access necessary to add checkpoint capability, could inform further work in area of checkpointing.

It would also be helpful to survey a wide sample of systems at different supercomputing centers to snapshot the current practices in checkpointing. This would provide context for interpreting the representativeness of the DataStar data relative to other systems. Although the survey of user runtime estimate accuracy was similarly limited to a single center and system, data on inaccuracy of user runtime requests is available for many systems worldwide and shows a consistent pattern. This provides some support to the applicability of the results in most cases. No corresponding widely-available data exists for checkpointing.

Non-Uniform Distribution of Hardware Failures

Looking to the future of HPC systems, the demand for ever-increasing parallelism is expected to continue, if not accelerate, due to limitations on further increases in clock speeds. Multi-teraflop systems have processor counts in the tens of thousands, and future petaflop systems are likely to contain hundreds of thousands of nodes [68]. Under these conditions, the frequency of occurrences of a hardware failure somewhere

on the system increases to a rate that can no longer be ignored by scheduling systems [68].

Under current MPI [78] style programming paradigms, simultaneous hardware failures on any subset of a job's processors are equivalent to a failure on all. We assume that any one hardware failure results in needing to restart the entire job from the most recent checkpoint. Using fault-tolerant versions of the MPI software infastructure could result in only the processes on affected processors needing restarting. However, for parallel scientific codes, which require frequent inter-processor communication, these are nearly equivalent because unaffected processes quickly exhaust productive work that can be accomplished until communication with affected processors resumes.

It is the case that the probability of a hardware failure occurring on a subset of size S of the processors may not be equal across all possible subsets of of size S. This is a function of how much non-processor underlying hardware infastructure is shared by the processors. For example, all processors sharing a power supply would be affected by a failure of the power supply. It follows that a job whose S processors all share the same power supply is less likely to experience a failure than one whose processors are spread over many power supplies. Therefore, the cost of restarting from checkpoint, and probability of incurring that cost, are potentially important input data for the scheduler. When does it make sense to start a job immediately on a more failure-prone subset of the processors, and when would delaying the job until a less failure-prone subset of processors becomes available better optimize aggregate utility? An extension of the checkpoint-aware GA scheduler could incorporate failure rate data to answer those questions.

References

[1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing (ICS '04)*, pages 277–286, New York, NY, USA, 2004. ACM.

[2] A. AuYoung, L. Grit, J. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *15th IEEE International Symposium on High Performance Distributed Computing*, June 2006.

[3] A. AuYoung, A. Vahdat, and A. C. Snoeren. Evaluating the impact of inaccurate information in utility-based scheduling. In *Proceedings of 22nd ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis (SC-09)*, November 2009. (Accepted for publication).

[4] D. H. Bailey. A high-performance fast Fourier transform algorithm for the cray-2. *The Journal of Supercomputing*, 1(1):43–60, March 1987.

[5] J. Bashor. Nersc first to reach goal of seamless shutdown, restart of supercomputer. *NERSC*, August 22, 1997.

[6] G. Bell and J. Gray. What's next in high-performance computing? *Commun. ACM*, 45(2):91–95, 2002.

[7] F. B. Berlin. Seymour cray, 1925-1996 [in memoriam]. *Computational Science & Engineering, IEEE*, 3(4):90–92, Winter 1996.

[8] J. Brevik, D. Nurmi, and R. Wolski. Predicting Bounds on Queuing Delay for Batch-scheduled Parallel Machines. In *Proceedings of ACM Principles and Practices of Parallel Programming (PPoPP)*, March 2006.

[9] R. Buehler. Planning, personality, and prediction: The role of future focus in optimistic time predictions. *Organizational Behavior & Human Decision Processes*, 92(1), Septempber 2003.

[10] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger. Economics models for resource management and scheduling in grid computing. In *Grid Computing, Concurrency and Computation: Practice and Experience*, 2002.

[11] R. Buyya and K. Bubendorfer, editors. *Market Oriented Grid and Utility Computing*. Wiley, 2009.

[12] L. Carrington, N. Wolter, A. Snavely, and C. B. Lee. Applying an Automated Framework to Produce Accurate Blind Performance Predictions of Full-Scale HPC Applications. In *Proceedings of the 2004 Department of Defense Users Group Conference*. IEEE Computer Society Press, 2004.

[13] S. J. Chapin, W. Cirne, D. G. Feitelson, J. P. Jones, S. T. Leutenegger, U. Schwiegelshohn, W. Smith, and D. Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In *IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 67–90, London, UK, 1999. Springer-Verlag.

[14] S.-H. Chiang, A. C. Arpaci-Dusseau, and M. K. Vernon. The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In *JSSPP '02: Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 103–127, London, UK, 2002. Springer-Verlag.

[15] B. N. Chun. *Market-based cluster resource management*. PhD thesis, University of California, Berkeley, 2001. Chair-David E. Culler.

[16] 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 *EmNets '05: Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors*, pages 19–28, Washington, DC, USA, 2005. IEEE Computer Society.

[17] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 30, Washington, DC, USA, 2002. IEEE Computer Society.

[18] W. Cirne and F. Berman. Adaptive Selection of Partition Size for Supercomputer Requests. In *IPDPS '00/JSSPP '00: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 187–208, London, UK, 2000. Springer-Verlag.

[19] W. Cirne and F. Berman. A comprehensive model of the supercomputer workload. In *WWC '01: Proceedings of the Workload Characterization, 2001.*

WWC-4. 2001 IEEE International Workshop, pages 140–148, Washington, DC, USA, 2001. IEEE Computer Society.

[20] W. Cirne and F. Berman. A model for moldable supercomputer jobs. In *Proceedings of International Parallel and Distributed Processing Sympsium (IPDPS '01)*, April 2001.

[21] W. Cirne and F. Berman. Using moldability to improve the performance of supercomputer jobs. volume 62, pages 1571–1601, Orlando, FL, USA, 2002. Academic Press, Inc.

[22] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK Benchmark: Past, Present and Future. *Concurrency: Practice and Experience*, 15:803–820, 2003.

[23] J. Dongarra, H. Meuer, H. Simon, and E. Strohmaier. Biannual top-500 computer lists track changing environments for scientific computing. *SIAM News*, 34(9), 2001.

[24] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. High-perofmance computing: Clusters, constellations, mpps, and future directions. *Computing in Science and Engineering*, 7(2):51–59, Mar/Apr 2005.

[25] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. In *SIAM Journal on Discrete Mathematics*, volume 2(4), pages 473–487, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[26] T. Earheart and N. Wilkins-Diehr. San diego supercomputer center. Provided the original workload to the Parallel Workloads Archive [7, 5] of Dror Feitelson et al. The specific Parallel Workoads Archive file version we used is SDSC-BLUE-2000-2.1-cln.swf. (Version 2.1 has a more streamlined representation of jobs17priorities than version 3.1.), 2003.

[27] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 01(2):97–108, 2004.

[28] D. Feitelson. Parallel workloads archive and standard workload format. http://www.cs.huji.ac.il/labs/parallel/workload/, -.

[29] D. Feitelson. On the interpretation of top500 data. *International Journal of High Performance Computing Applications*, 13(2):146–153, 1999.

[30] D. G. Feitelson and L. Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26, London, UK, 1996. Springer-Verlag.

[31] D. G. Feitelson, L. Rudolph, and et al. Parallel Job Scheduling - A Status Report. In *Lecture Notes in Computer Science*, volume 3277, pages 1–16. Springer-Verlag, 2004.

[32] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997.

[33] D. G. Feitelson and L. R. L. Scheduling. Parallel Job Scheduling: Issues and Approaches. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 1–18. Springer, 1995.

[34] E. Frachtenberg and D. G. Feitelson. Pitfalls in parallel job scheduling evaluation. In *Proceedings of Job Scheduling Strategies for Parallel Processing*, volume 3834. Springer, 2005.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[36] R. L. Henderson. Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag.

[37] IBM Redbooks. *Workload Management with LoadLeveler*. IBM, November 29 2001.

[38] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 160–169, Washington, DC, USA, 2004. IEEE Computer Society.

[39] D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. *Lecture Notes in Computer Science*, 2221:87, 2001.

[40] B. Johannes. Scheduling parallel jobs to minimize the makespan. In *Journal of Scheduling*, volume 9(5), pages 433–452. Springer Netherlands, October 2006.

[41] J. P. Jones and B. Nitzberg. Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization. In *IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 1–16, London, UK, 1999. Springer-Verlag.

[42] P. Krueger, T. H. Lai, and V. A. Radiya. Job Scheduling is More Important than Processor Allocation for Hypercube Computers. *IEEE Transactions on Parallel and Distributed Systems*, 5:488 – 497, 1994.

[43] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 1(3):169–182, 2005.

[44] B. G. Lawson and E. Smirni. Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. *SIGMETRICS Perform. Eval. Rev.*, 29(4):40–47, 2002.

[45] C. B. Lee. Parallel job scheduling algorithms and interfaces. Research Exam for the M.S. degree, University of California, San Diego, May 2004.

[46] C. B. Lee, Y. Schwartzman, J. Hardy, and A. E. Snavely. Are user runtime estimates inherently inaccurate? In *In 10th Job Scheduling Strategies for Parallel*, June 2004.

[47] C. B. Lee and A. E. Snavely. On the user-scheduler dialogue: Studies of user-provided runtime estimates and utility functions. *International Journal of High Performance Computing Applications*, 20(4):495–506, 2006.

[48] C. B. Lee and A. E. Snavely. Precise and realistic utility functions for user-centric performance analysis of schedulers. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 107–116, New York, NY, USA, 2007. ACM.

[49] D. J. Lehmann, L. O'Callaghan, and Y. Shoham. Truth revelation in approximately efficient combinatorial auctions. *CoRR*, cs.GT/0202017, 2002.

[50] H. Ligang, S. A. Jarvis, D. P. Spooner, C. Xinuo, and G. R. Nudd. Hybrid performance-oriented scheduling of moldable jobs with qos demands in multiclusters and grids. In *Proceedings of Grid and Cooperative Computing (GCC 2004)*, October 2004.

[51] D. Lovallo and D. Kahneman. Delusions of success. *Harvard Business Review*, 81(7), July 2003.

[52] P. Messina. The concurrent supercomputing consortium: Year 1. *Parallel & Distributed Technology: Systems & Applications, IEEE [see also IEEE Concurrency]*, 1(1):9–16, Feb 1993.

[53] Meuer and H. Werner. The top500 project: Looking back over 15 years of supercomputing experience. *Informatik-Spektrum*, 31(3):203–222, June 2008.

[54] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top500: http://www.top500.org.

[55] A. Mu'alem and D. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *12th Intl. Parallel Processing Symposium*, pages 542–546, April 1998.

[56] A. Mutz, R. Wolski, and J. Brevik. Eliciting honest value information in a batch-queue environment. In *Proceedings of Grid2007*, 2007.

[57] National Energy Research Scientific Computing Center. Bassi System. http://www.nersc.gov/nusers/resources/bassi/.

[58] National Energy Research Scientific Computing Center. Franklin System. http://www.nersc.gov/nusers/systems/franklin/.

[59] C. Ng, P. Buonadonna, B. N. Chun, A. C. Snoeren, and A. Vahdat. Addressing strategic behavior in a deployed microeconomic resource allocator. In *In Proc. 3rd Workshop on Economics of Peer-to-Peer Systems*, pages 99–104, 2005.

[60] P. Norvig. Python code. available at http://aima.cs.berkeley.edu/python/readme.html. Copyright 1998 - 2002. Used according to terms of license., 1998-2002.

[61] D. Nurmi, R. Wolski, and J. Brevik. Probabilistic advanced reservations for batch-scheduled parallel machines. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 289–290, New York, NY, USA, 2008. ACM.

[62] D. C. Nurmi, J. Brevik, and R. Wolski. Qbets: queue bounds estimation from time series. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 379–380, New York, NY, USA, 2007. ACM.

[63] Oak Ridge National Laboratory. Cheetah System. http://www.ccs.ornl.gov/Cheetah/LL.html.

[64] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. *ipdps*, 19:299b, 2005.

[65] D. Perkovic and P. J. Keleher. Randomization, Speculation, and Adaptation in Batch Schedulers. In *Supercomputing 2000*, pages 48–48, 2000.

[66] W. Pfeiffer. Personal interview. San Diego Supercomputer Center of the University of California, San Diego. La Jolla, California., April 9, 2004.

[67] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. pages 213–223, January 1995.

[68] D. A. Reed, C. Lu, and C. L. Mendes. Reliability challenges in large systems. *Future Generation Computer Systems*, 22:293–302, 2006.

[69] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence, Englewood Cliffs, New Jersey, 1995.

[70] San Diego Supercomputer Center. Blue Horizon System. http://www.sdsc.edu/pub/envision/v16.1/bluehorizon.html.

[71] San Diego Supercomputer Center. DataStar System. http://web.archive.org/web/20071009234703/www.sdsc.edu/us/resources/datastar/.

[72] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, 2005.

[73] H. D. Simon, W. T. C. Kramer, and R. F. Lucas. Building the teraflops/petabytes production supercomputing center. pages 61–77, 1999.

[74] G. Singh, C. Kesselman, and E. Deelman. Adaptive pricing for resource reservations. In *8th IEEE/ACM International Conference on Grid Computing (Grid 2007)*, 2007.

[75] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. The easy - loadleveler api project. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47, London, UK, 1996. Springer-Verlag.

[76] A. Snavely and D. Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreading Processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, November 2000.

[77] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *Proceedings of the ACM 2002 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS. 2002)*, pages 66–76, Marina Del Rey, June 2002.

[78] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1995.

[79] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 514–519, 2002.

[80] S. Srinivasan, S. Krishnamoorthy, and P. Sadayappan. Robust scheduling of moldable parallel jobs. *International Journal of High Performance Computing Networks*, 2(2-4):120–132, 2004.

[81] I. Stoica, H. M. Abdel-Wahab, and A. Pothen. A microeconomic scheduler for parallel computers. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 200–218, London, UK, 1995. Springer-Verlag.

[82] E. Strohmaier and H. W. Meuer. Supercomputing: What have we learned from the top500 project? *Computing and Visualization in Science*, 6, 2004.

[83] D. Talby, D. G. Feitelson, and A. Raveh. Comparing logs and models of parallel workloads using the co-plot method. In *IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 43–66, London, UK, 1999. Springer-Verlag.

[84] D. Talby, D. Tsafrir, Z. Goldberg, and D. G. Feitelson. Session-based, estimation-less, and information-less runtime prediction algorithms for parallel and grid job scheduling. Technical Report 2006-77, School of Computer Science and Engineering, the Hebrew University, Jerusalem, Israel, August 2006.

[85] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Modeling user runtime estimates. *Job Scheduling Strategies for Parallel Processing*, 3834:1–35, 2005.

[86] D. Tsafrir and D. Feitelson. The dynamics of backfilling: Solving the mystery of why increased inaccuracy may help. *Workload Characterization, 2006 IEEE International Symposium on*, pages 131–141, Oct. 2006.

[87] S. Ulam and N. Metropolis. The monte carlo method. *Journal of the American Statistics Association*, 44(247):335–341, September 1949.

[88] G. van Rossum. Python Tutorial. Technical report, CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1995.

[89] G. van Rossum and J. de Boer. Linking a Stub Generator (AIL) to a Prototyping Language (Python). In *Proceedings of the Spring 1991 EurOpen Conference*, Tromso, Norway, May 1991.

[90] C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart, and W. S. Stornetta. Spawn: A distributed computational economy. *IEEE Trans. Softw. Eng.*, 18(2):103–117, 1992.

[91] J. Weinberg and A. Snavely. Symbiotic space-sharing on sdsc's datastar system. In *The 12th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '06)*, St. Malo, France, June 2006.

[92] 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.

[93] J. Weinberg and A. Snavely. When Jobs Play Nice: The Case For Symbiotic Space-Sharing. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC-15 '06)*, Paris, France, June 2006.

[94] J. William A. Ward, C. L. Mahood, and J. E. West. Scheduling jobs on parallel systems using a relaxed backfill strategy. In *JSSPP '02: Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 88–102, London, UK, 2002. Springer-Verlag.

[95] R. Wilson. Architecture of power markets. *Econometrica*, 70(4):1299–1340, July 2002.

[96] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. G-commerce: Market Formulations Controlling Resource Allocation on the Computational Grid. In *International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, April 2001. IEEEE.

[97] K. Yoshimoto, P. Kovatch, and P. Andrews. Coscheduling with user-settable reservations. In *In 10th Job Scheduling Strategies for Parallel*, June 2005.

[98] D. Zotkin and P. J. Keleher. Job-Length Estimation and Performance in Backfilling Schedulers. In *IEEE International Symposium on High Performance Distributed Computing*, 1999.